Scenario-based Programming for SDN Policies [*]

Yifei Yuan Dong Lin Rajeev Alur Boon Thau Loo
University of Pennsylvania

ABSTRACT

Recent emergence of software-defined networks offers an opportunity to design domain-specific programming abstractions aimed at network operators. In this paper, we propose scenario-based programming, a framework that allows network operators to program network policies by describing representative example behaviors. Given these scenarios, our synthesis algorithm automatically infers the controller state that needs to be maintained along with the rules to process network events and update state. We have developed the NetEgg scenario-based programming tool, which can execute the generated policy implementation on top of a centralized controller, but also automatically infers flow-table rules that can be pushed to switches to improve throughput. We study a range of policies considered in the literature and report our experience regarding specifying these policies using scenarios. We evaluate NetEgg based on the computational requirements of our synthesis algorithm as well as the overhead introduced by the generated policy implementation. Our results show that our synthesis algorithm can generate policy implementations in seconds, and the automatically generated policy implementations have performance comparable to their hand-crafted implementations.

CCS Concepts

•**Networks** → **Programming interfaces;** *Network management;*

[*]This research was partially supported by NSF Expeditions in Computing award CCF 1138996, CNS-1218066, CNS-0845552, and CNS-1117052.

CoNEXT'15 December 01-04, 2015, Heidelberg, Germany
© 2015 ACM. ISBN 978-1-4503-3412-9/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2716281.2836119

Keywords

Software defined networking; programming by examples; policy synthesis

1. INTRODUCTION

Software-defined networking (SDN) holds the promise of extensible routers that can be customized directly by network operators. Major router vendors now provide APIs (OpenFlow or vendor specific) that provide various forms of extensibility for traffic steering, on-demand network virtualization, security policies, and dynamic service chaining. The enhanced programming interface of SDN offers an opportunity to design domain-specific programming abstractions for network operators to take advantage of the flexibility to program network policies.

To take advantage of this new wave of innovation, recently proposed domain-specific languages or DSLs (e.g. declarative networking [14], Frenetic [6], Pyretic [16], NetKAT [2], NetCore [15], FlowLog [18], Merlin [20], FatTire [19]) make it easier to program controllers with orders of magnitude reduction in code sizes by raising the level of abstraction.

A key challenge that has yet to be addressed is providing an intuitive programming abstraction that allows network operators even with little programming experiences to program their own protocols and policies, hence taking advantage of the new programming interface.

Motivated by recent work on programming by examples [10, 9, 11], we investigate an alternative approach aiming at providing network operators intuitive programming interfaces. Our approach is based on *synthesizing* an implementation automatically from *example scenarios* and providing a platform whereby operators can observe the synthesized implementation at runtime, and then tweak their input scenarios to refine the synthesized program.

Our proposed approach is based on the observation that network operators typically like to use examples such as timing diagrams to design new network configurations and policies. In most cases, these examples would be generalized into design documents, followed by pseudocode and then finally implementation. We aim to facilitate the entire process by generating implementations directly from the examples themselves, hence giving the power of network programmability to all network operators. While the focus of this paper is on SDN settings, the approach is general and can be applied to any network protocol design and implementation.

Specifically, this paper makes the following contributions:

Scenario-based programming framework. We propose the framework of scenario-based programming (Section 4), which allows network operators to specify network policies using example behaviors. Instead of implementing a network policy by programming, the network operator simply specifies the desired network policy using *scenarios*, which consist of examples of packet traces, and corresponding actions to each packet.

Proof-of-concept design and implementation. We have developed the NetEgg tool, including a synthesis algorithm (Section 5) and an interpreter for executing policies. Given the scenarios as input, our synthesizer automatically generates a controller program that is consistent with example behaviors, including inferring the state that needs to implement the network policy, relevant fields associated with the state and rules for processing packets and updating states. The interpreter executes the generated policy program for incoming network events on the controller, as well as infers rules that can be pushed onto switches (Section 6).

Validation. We validate the NetEgg tool by synthesizing SDN programs that use the POX controller directly from examples. Our tool is agnostic to the choice of SDN controllers, and can also be used in non-SDN settings. We demonstrate that using our approach, we are able to synthesize a range of network policies using a small number of examples in seconds (Section 7). The synthesized controller program has low performance overhead and achieves comparable performance to equivalent imperative programs implemented manually in POX (Section 8). Moreover, the example scenarios are concise compared to equivalent imperative programs.

NetEgg should be viewed as enabling a rapid prototyping platform where users can iterate through example scenarios, observe runtime behavior to determine correctness, and tweak their scenarios otherwise. NetEgg is correct and consistent with respect to the input scenarios. Synthesizing the input scenarios themselves based on high-level correctness properties is an avenue of future work.

2. ILLUSTRATIVE EXAMPLE

To illustrate the use of NetEgg, we consider the example where a network operator wants to program a learning switch policy supporting migration of hosts on top of the controller for a single switch. The learning switch learns incoming ports for hosts. For an incoming packet, if the destination MAC address is learnt, it sends this packet out to the port associated with the destination MAC address; otherwise it floods the packet. To support migration of hosts, the learning switch needs to remember the latest incoming port of a host.

To program the policy, the network operator simply describes example behaviors of the policy in representative scenarios, in the form of network timing diagrams. Figure 1 shows a scenario described by network operators.

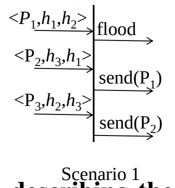

Figure 1: **A scenario describing the learning switch. In the scenario, a packet is denoted by a 3-tuple: ⟨incoming port, source MAC, destination MAC⟩.**

The scenario. In this scenario, the network operator describes example behaviors of the policy using three packets. The first packet arriving on port P_1 with source MAC address h_1 and destination MAC address h_2 is flooded by the switch, since no port has been learnt for h_2. The second packet from h_3 to h_1 should be sent directly to the port P_1, according to the port learnt from the first packet. The third packet from h_2 to h_3 should be sent to the port P_2, since the second packet indicts that h_3 is associated with port P_2. Note that instead of using real port numbers and MAC addresses in the packet, the network operator uses *variables* for each field. The variables stand for a variety of concrete values.

Given this scenario, NetEgg automatically synthesizes the desired program. The synthesized program can be executed on the SDN controller, as well as install flow table rules onto switches. As part of the program generation, NetEgg automatically generates the data structures and code necessary to implement the policy.

Network operators may further test the synthesized program using existing verification and testing techniques, and refine the program if needed. As part of refinement, network operators simply illustrate new scenarios (e.g. obtained from counter examples) to NetEgg, and NetEgg automates the refinement by synthesizing a new program from the new set of scenarios. We will demonstrate more use cases in Section 7.

3. TOOL OVERVIEW

Figure 2 provides a high-level overview of NetEgg. The network operator describes example behaviors about the desired network policy in representative scenarios (network timing diagrams, which we refer to as scenarios) to NetEgg. These network timing diagrams are written in a configuration language, which we will describe in Section 4. The tool first checks whether there exist conflicts among the scenarios. If two scenarios conflict with one another, NetEgg displays the conflict to the network operator. After the operator resolves all conflicts, NetEgg tries to generate a policy described in the scenarios.

The generated policy uses a set of *state tables* and a *policy table*. State tables are used to remember the history of a policy execution. The policy table dictates the actions for processing incoming network events and updates to state tables for various cases.

When executing the policy, the interpreter, sitting on top of the controller, looks up the policy table for incoming network events (e.g. packetin, connectionup and other events), which will determine state table updates and actions to be applied to the network events. Moreover, NetEgg automatically infers rule updates to the data plane from current state of the policy execution, thus reducing controller overhead and network delay.

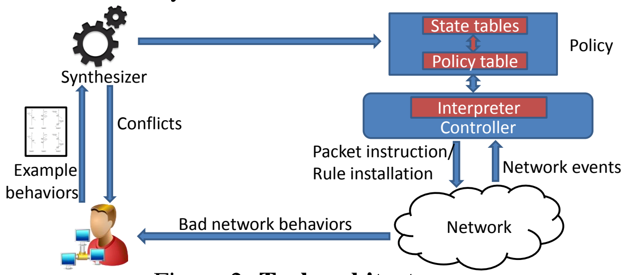

Figure 2: **Tool architecture.**

While NetEgg is general to handle any network events, we focus on packetin events in this paper in order to simplify our presentation.

Revisiting the learning switch example from the previous section, we first describe the state tables that are generated by our tool, before describing the policy.

3.1 State Tables

In our example, the learning switch needs to remember whether a port is learnt for a MAC address, and if learnt, which port is associated with the MAC address. Hence, the generated policy maintains a state table ST, which stores a state and a value (in this case a port number) for each MAC address. An example of the snapshot of ST is shown below.

MAC	state	value
A	1	2
OP...	...	...

Our tool automatically derives the facts that for a given MAC address *macaddr*, the state of *macaddr* in ST is either 0 or 1, indicating that the port associated with *macaddr* is unknown yet, or learnt, respectively. ST also stores a port for MAC addresses with state 1. Initially, the program assigns all states in the table to be 0. The program accounts for two cases: 1) When the destination port is unknown, it floods the packet through all ports; 2) When the incoming packet's destination port is known, it sends the packet out through the port associated with the destination MAC address. In both cases, the state associated with the source MAC address is set to be 1, and the incoming port for the source MAC address is remembered.

3.2 Policy Tables

The state table is manipulated by rules implementing the desired policy. These rules are captured in a policy table, as shown in Table 1 for the learning switch example. We delay the discussion of its generation to Section 5.

match	test	actions	update
$*$	ST(dstmac) .state=0	flood	ST(srcmac) :=(1, port)
$*$	ST(dstmac) .state=1	send(ST(dstmac) .value)	ST(srcmac) :=(1, port)

Table 1: **The policy table for the learning switch.**

The policy table contains two *rules*, represented as the two rows in the table, corresponding to the two cases in the program described above. Every rule has four components: match, test, actions and update. The match specifies the packet fields and corresponding values that a packet should match. In this example, no matches need to be specified and we use $*$ to denote the wildcard. The test is a conjunction of checks, each of which checks whether the state associated with some fields in a state table equals a certain value. For example, the test in the second rule has one check ST(dstmac).state=1, which checks whether the state associated with the dstmac address of the packet is 1 in ST. The actions define the actions that are applied to matched packets. In this example, the action in the first rule floods the matched packet to all ports and the action send(ST(dstmac).value) in the second rule first reads the value (in this case, the port) stored in ST for the dstmac address of the matched packet, and sends the packet to that port. The update is a sequence of writes, each of which changes the state and value associated with some fields in a state table to certain values. For example, the write ST(srcmac):=(1,port) changes the state associated with the srcmac address of the packet to 1 in ST, and stores the value associated with the srcmac address of the packet to the port of it.

3.3 Interpreter

The interpreter processes incoming packets at the controller using the policy table. The pseudocode of the interpreter is shown in Figure 3. The interpreter matches each incoming packet against each rule in the policy table in order. A rule is matched, if the packet fields match the match and all checks in the test of the rule are satisfied. The first matched rule applies actions to the packet, and state tables are updated according to the update of the rule. Moreover, NetEgg automatically infers the rules that can be installed on the data plane from the latest configuration of state tables. The corresponding function is update_flowtable in the pseudocode. We will describe policy execution in more detail in Section 6.

Example. Figure 4 shows an illustrative execution for the incoming packet trace in subfigure (a). Since the purpose of this example is to illustrate how a policy table is executed, we assume that every packet is processed on the controller. Initially, all states in the state table ST are 0, and all values are $\perp$, meaning unknown, as shown in subfigure (b). The first packet p_1 is matched against each rule in Table 1 in order at the controller. The first matched rule is

```
Input: a packet $p$
for $i = 1$ to $n$ do
    if rule $r_i$ matches $p$ then
        execute the actions and update of rule $r_i$ on $p$
        update_flowtable($p$)
        return
    end if
end for
apply default actions to $p$
```

Figure 3: **The interpreter.**

the first rule, since p_1 matches the match ($*$) and the state of the field dstmac of p_1 in ST is 0, satisfying the check (ST(dstmac).state=0) in test of the rule. Therefore, the rule applies the action which instructs the switch to flood p_1, and updates the state table as in subfigure (c). The second packet p_2 in the trace matches the second rule in the policy table, since the state of its dstmac is 1. The program sends p_2 out to port 2, which is stored in the state table associated with MAC address A. Applying the update of the rule, we get the state table as in subfigure (d). The third packet p_3 matches the second rule in the policy table, and the updated state table remains the same and thus not shown here. The last packet p_4 suggests that the host with MAC A has migrated to port 3, and it matches the second rule in the policy table and gets sent to port 1. Subfigure (e) shows the state table after applying the update.

p_1:<port=2, srcmac=A, dstmac=B>
p_2:<port=1, srcmac=B, dstmac=A>
p_3:<port=2, srcmac=A, dstmac=B>
p_4:<port=3, srcmac=A, dstmac=B>

(a) An example packet trace

MAC	state	value
A	0	⊥
B	0	⊥

(b) The initial state table

MAC	state	value
A	1	2
B	0	⊥

(c) The state table after p_1

MAC	state	value
A	1	2
B	1	1

(d) The state table after p_2

MAC	state	value
A	1	3
B	1	1

(e) The state table after p_4

Figure 4: **An illustrative execution.**

4. NetEgg MODEL

In this section, we first describe the scenario-based programming model of NetEgg, and explain how this model allows the operator to describe example network behaviors in representative scenarios. Second, we define the policy model, which includes the model of state tables and policy tables. We will show how to generate a policy from scenarios in the next section.

4.1 Programming Language

Packet-type	P	::=	$\langle \mathrm{f}_1 : \mathtt{T}_1, .., \mathrm{f}_k : \mathtt{T}_k \rangle$
Symbolic packet	*sp*	::=	$\langle sv_1, .., sv_k \rangle$
Action	a	::=	drop|flood|send(port)
			|modify(f,v)|..
Event	e	::=	$sp \Rightarrow [a_1, .., a_i]$
Scenario	*sc*	::=	$[e_1, .., e_j]$
Program	prog	::=	$\{sc_1, .., sc_n\}$

Figure 5: **Scenario-based programming model.**

NetEgg provides a configuration language for expressing network timing diagrams. In this language, variables and fields of packets are typed. Examples of base types we use are `bool`, `PORT`, `IP_ADDR` (set of IP addresses), `MAC_ADDR` (set of MAC addresses). A packet-type consists of a list of names of fields of the packet along with their types. In our example, the packet-type consists of three fields and is given by $\langle$port : `PORT`, srcmac : `MAC_ADDR`, dstmac : `MAC_ADDR`$\rangle$.

A (concrete) packet specifies a value for each field of type corresponding to that field. A symbolic value of a type `T` is either a concrete value of type `T`, or a variable x of type `T`. A symbolic packet specifies a symbolic value for each field.

We use `Act` to denote the set of action primitives for processing packets. For the action primitives with parameters, the user can use either concrete values or variables of the corresponding type.

In NetEgg, we provide a library that supports standard packet fields and actions such as drop, flood, send(port) (send to a port), modify(f,v) (modify the value of field f to v). Our tool also supports user-defined packet-type using customized field names and types, as well as user-defined actions. One can generalize it by providing handlers for user-defined fields and action primitives.

An *event* is a pair of a symbolic packet *sp* and a list of actions $[a_1, ..a_l]$, denoted as $sp \Rightarrow [a_1, .., a_l]$. A *scenario* is a finite sequence of events. A scenario-based program is a finite set of scenarios.

With the notation, the scenario of Figure 1 corresponds to:

$$\begin{array}{l} P_1, h_1, h_2 \Rightarrow \text{flood} \\ P_2, h_3, h_1 \Rightarrow \text{send}(P_1) \\ P_3, h_2, h_3 \Rightarrow \text{send}(P_2) \end{array}$$

A scenario is *concrete* if all the symbolic packets and actions appearing in the scenario have only concrete values. A scenario-based program with symbolic scenarios can be viewed as a short-hand for a set of concrete scenarios. This set is obtained by replacing each variable by every possible value of the corresponding type with the following requirements. First, a variable can only take values that have not appeared in the scenario-based program. Second, if the same variable appears in multiple symbolic packets and actions in the program, then it gets replaced by the same value. Third, different variables in a program get replaced by different values. Thus, the symbolic scenario of Figure 1 corresponds to $\Pi_{i=0,1,2}(n-i)(l-i)$ concrete scenarios if the type `MAC_ADDR` and `PORT` contain n and l distinct values, respectively.

The language itself is simple and can be viewed more as a configuration language rather than a general-purpose programming language. One can even build a visual tool that takes as input scenarios drawn as actual network timing diagrams, and generate the configuration.

4.2 Policy Model

A policy consists of a *policy table* along with *state tables* that store the history of policy execution.

State Tables. A state table is a key-value map that maintains states and values for relevant fields. Let $\mathtt{T}_{i_j}$ be some base type appearing in the packet-type, S be a state set with finitely many states, and the packet-type be $\langle \text{f}_1 : \mathtt{T}_1,..,\text{f}_k : \mathtt{T}_k\rangle$. A d-dimensional state table ST stores a state in S and a value of type $\mathtt{T}_{i_{d+1}}$, for all keys of type $\mathtt{T}_{i_1} \times .. \times \mathtt{T}_{i_d}$

The operations we allow on a state table are reads, checks and writes.

Let ST be a state table of type $\mathtt{T}_1 \times .. \times \mathtt{T}_d \rightarrow S \times \mathtt{T}_{d+1}$, let f$_1$,..f$_d$ be field names of type $\mathtt{T}_1$,..,$\mathtt{T}_d$, respectively. A read of ST indexes some entry in ST, and is of the form ST(f$_1$,..,f$_d$). A check of ST checks whether the state associated with some key is a particular state. Syntactically, it is a pair of a read and a state, written as ST(f$_1$,..,f$_d$).state=s, where $\mathtt{s} \in S$ is a state. In our example, ST(dstmac).state=0 is a check with the field dstmac. In contrast to a check, a write of a state table changes the state along with the value associated with some key. A write of ST is of the form ST(f$_1$,..,f$_d$):=(sv,fv). Here, sv is either a state, or - representing no change. fv is either a concrete value of type $\mathtt{T}_{d+1}$, - representing no change, or a field name of type $\mathtt{T}_{d+1}$. In our example, ST(srcmac):=(1,port) is a write of ST with the field srcmac.

We use the term configurations for the snapshots of state tables. For example, the initial configuration of the state table in our example maps every MAC address to (0,$\perp$) as shown in figure 4(b). Here, we use $\perp$ to represent the fact that no value is stored. A read ST(f$_1$,..,f$_d$) for a packet p at a configuration c returns the state-value pair stored in ST for the key $(p.\text{f}_1, .., p.\text{f}_d)$ at c. We use ST(f$_1$,..,f$_d$).state and ST(f$_1$,..,f$_d$).value to denote the state and value in the returned pair. A check ST(f$_1$,..,f$_d$).state=s is true for a packet p at a configuration c if the state read from ST at the configuration c is s. In the example in Figure 4, ST(dstmac).state=0 is true for p_1 at the initial configuration (subfigure (b)) of ST. A write ST(f$_1$,..,f$_d$):=(sv,fv) for a packet p writes the state-value pair to the corresponding entry indexed by the read. Note that if sv(fv, resp.) is -, the write does not write any state(value, resp.) to ST, and if fv specifies a field name, the value of p.fv should be written.

Policy Tables. Given a set of state tables $\mathcal{T}$, a rule r based on $\mathcal{T}$ has four components, namely, match, test, actions and update. match is of the form $\langle$f$_1$=v$_1$,..,f$_k$=v$_k\rangle$, where f$_i$ is a name of a packet field, and v$_i$ is a concrete value or a wildcard. A packet p matches $\langle$f$_1$=v$_1$,..,f$_k$=v$_k\rangle$ iff v$_i$ is a wildcard, or $p.\text{f}_i = \text{v}_i$ for all $i = 1$ to k. The actions is a list of actions using action primitives in Act. In the case where an action primitive accepts parameters, the parameters can be concrete values or values read from state tables in $\mathcal{T}$ using reads. test is a conjunction of checks and update is a sequence of writes, where each check/write is of some state table in $\mathcal{T}$. As an example, last two rows in Table 1 are two rules. A *policy table* based on $\mathcal{T}$ is an ordered list of rules, and every rule is based on $\mathcal{T}$.

A configuration $\mathcal{C}$ of a policy consists of all the configurations of each state table in $\mathcal{T}$, on which the policy table is based. A packet p matches a rule at a configuration $\mathcal{C}$ iff p matches match and every check in test is true for p at the corresponding configuration in $\mathcal{C}$. Suppose the first matched rule for a packet p at a configuration $\mathcal{C}$ is r. Then actions of r will be executed on p and every write in update of r will be executed. We denote the execution for packet p as $\mathcal{C} \xrightarrow{p/as}_{PT} \mathcal{C}'$, with $\mathcal{C}'$ the new configuration, and as the actions applied to p.

5. POLICY GENERATION

Given a set of scenarios describing a policy, our synthesizer first checks if there are conflicts among scenarios. This process is straightforward and thus omitted due to the space constraint. In this section, we focus on how to generate a policy, consisting of a set of state tables and a policy table, given a set of scenarios without conflicts. We start this section by discussing the objective policies NetEgg aims to generate. Then we present the synthesis algorithm in steps.

5.1 The Policy Learning Problem

First, we note that, since the input scenarios describe the behaviors of the desired policy in representative scenarios, the generated policy should be *consistent* with all the behaviors described in all scenarios.

DEFINITION 1 (CONSISTENCY). *Given a concrete scenario $SC = [sp_1 \Rightarrow as_1, .., sp_k \Rightarrow as_k]$, a policy table PT is* consistent *with SC iff $\mathcal{C}_{i-1} \xrightarrow{sp_i/as_i}_{PT} \mathcal{C}_i$ for $i = 1, .., k$, where $\mathcal{C}_0$ is the initial configuration in which every state table maps every key to the initial state 0 and a value of $\perp$. A policy table is consistent with a scenario-based program, iff it is consistent with all the concrete scenarios represented by the scenario-based program.*

As an example, the policy given in Table 1 is consistent with the scenario in Figure 1. However, the following policy is not consistent with the scenario, since it floods the third packet in the scenario instead of sending it to P_2.

match	test	actions	update
*	ST(dstmac) .state=0	flood	ST(srcmac) :=(1, port)
*	ST(dstmac) .state=1	send(ST(dstmac) .value)	ST(srcmac) :=(-, -)

Table 2: **An inconsistent policy table.**

In addition to consistency, NetEgg also aims to generate a generalized policy from input scenarios. For this, we aim to generate a consistent policy with minimal number of rules. To see how this heuristic can help to generate a general policy, let us consider the 3-rule policy table in Table 3. It can be verified that the policy is consistent with the scenario in Figure 1. However, this policy overfits the input scenario and will not generalize to a fourth packet such as $\langle P_1, h_4, h_2\rangle$, because this packet would be flooded by the policy. On the other hand, the desired policy in Table 1 only uses two rules, and can handle the fourth packet mentioned above correctly.

match	test	actions	update
$*$	ST(dstmac) .state=0	flood	ST(srcmac) :=(1, port)
$*$	ST(dstmac) .state=1	send(ST(dstmac) .value)	ST(srcmac) :=(2, port)
$*$	ST(dstmac) .state=2	send(ST(dstmac) .value)	ST(srcmac) :=(-, -)

Table 3: **A consistent yet restrictive policy table.**

We summarize the major computational problem as the following *policy learning problem.*

Policy learning problem. Given scenarios $SC_1, ..., SC_n$, the policy learning problem seeks a set of state tables $\mathcal{T}$ and a policy table PT based on $\mathcal{T}$, such that (1) PT is consistent with all scenarios SC_i. (2) PT has the smallest number of rules among all consistent policy tables.

A simple reduction from DFA learning problem [7] shows the following theorem.

THEOREM 1. *The policy learning problem is NP-hard.*

In the following sections, we break our synthesis algorithm into steps, and discuss each step separately.

5.2 Policies without tests

First, let us consider the simplest case where the desired policy table does not have tests. In this case, each rule in the policy table only consists of a match and actions. Thus, it suffices to generate an ordered list of matches, together with the corresponding list of actions.

Generate ordered match list. We first describe the algorithm which generates a list of matches from the input scenarios, shown in Algorithm 1. As defined in our scenario-based programming model, a symbolic packet represents a set of concrete packets, which is obtained by replacing symbolic values by concrete field values. Therefore, Algorithm 1 generates a match for each symbolic packet in the scenarios, by replacing symbolic values by $*$ (line 3). Moreover, to ensure that the generated match does not mismatch unrepresented packets, the algorithm inserts the generated match to the list, such that no match under it is completely covered (line 4). Note that for two generated matches which are overlapping with each other, we can order either one above the other. We will explain this through an example later in this subsection.

Search for actions. With the ordered list of matches, it is straightforward to search for the actions for every match in the list: we can simply search the first matched match in the list for each symbolic packet and check consistency between the actions with the packet and actions with the match. If actions for the match are not set, we can set them to the actions associate with the symbolic packet. A consistent policy table is returned when all actions in every symbolic events

Algorithm 1 generate_ordered_match_list($[SC_i]$)

```
1: $L = \emptyset$
2: for all packet $sp = \langle f_i = v_i \rangle$ in every scenario $SC_i$ do
3:     let m = $\langle f_i = m_i \rangle$, where $m_i = v_i$ if $v_i$ is a concrete value
       else $*$
4:     insert m to $L$
5: end for
6: return $L$
```

are consistent.

Examples. Consider a scenario describing a firewall, shown in Figure 6a. Here a symbolic packet is expressed by $\langle$srcip, dstip$\rangle$.

$A, ip1 \Rightarrow$ send(1)
$ip2, B \Rightarrow$ send(2)
$A, C \Rightarrow$ drop

(a) A scenario.

match	actions
srcip=A, dstip=C	drop
srcip=A, dstip=$*$	send(1)
srcip=$*$, dstip=B	send(2)

(b) The policy table.

Figure 6: A firewall example.

The generated policy table is shown in Figure 6b. The match column of the policy table is generated by Algorithm 1. We note that if the second rule is swapped with the third rule in the policy table, the resulted policy table is still consistent with the scenario. The reason is that by definition the variables $ip1$ and $ip2$ only represent values other than A, B, C. Thus, the packet $\langle A, B\rangle$ is not represented by any symbolic packets in the scenario.

5.3 Policies with tests

Now we consider synthesizing policies that use tests. As an example, consider the scenario for the learning switch in Figure 1. One can verify that a consistent policy table for the scenario requires tests. In fact, the only match in the match list generated by Algorithm 1 is $\langle * \rangle$, and every packet in the scenario matches it. However, their actions differ from each other. Therefore, in addition to the only match $\langle * \rangle$, a consistent policy table must have tests. In this subsection, we will assume that all tests in the policy table are given, and show the algorithm to synthesize the policy. We will relax this assumption in the next subsection.

Sketch. Suppose the tests used in the policy for learning switch are ST(dstmac).state=0 and ST(dstmac).state=1. Composing the match and test, we know that the policy table has the form as shown in Table 4.

match	test	actions	update
$*$	ST(dstmac) .state=0	ax_1	ST(srcmac) :=(sx_1, ux_1) ST(dstmac) :=(sx_2, ux_2)
$*$	ST(dstmac) .state=1	ax_2	ST(srcmac) :=(sx_3, ux_3) ST(dstmac) :=(sx_4, ux_4)

Table 4: **A policy table sketch for the learning switch.**

Table 4 is a symbolic representation of policy tables. It represents actions and update in rules using variables. We call such symbolic representations *policy table sketches* (or *sketches* in short). In this sketch, we derive all possible writes in update from the state tables that are used in each rule's tests. In this example, since the only state table accepts keys of type `MAC_ADDR`, the state table can be updated using either srcmac or the dstmac of a packet. Therefore, we have two writes per rule in its update, and sx's range over all states (in this example, {`0`,`1`}) plus a special symbol - meaning no update, and ux's range over all field names plus -. For ease of presentation, we intentionally ignore potential overwrittings due to the two writes in each rule's update. In practice, we need to consider other orders of writes as well. We use variables ax's to represent possible actions for each rule. In this example, ax's can range from {flood, send(ST(srcmac)), send(ST(dstmac))}, which are obtained from actions appearing in the scenario. To distinguish variables appearing in sketches from variables appearing in the scenarios, we will call these variables appearing in a sketch as sketch variables.

Algorithm 2 shows the algorithm of generating sketches given the match list L and tests. The set C contains all possible reads to the state tables appearing in $TESTS$ (line 1-line 4), and this set is used to derive writes in each rule (line 7). The sketch is generated by composing each match and test in $TESTS$ (line 5-line 6), and every rule has all possible writes derived from reads in C (line 7).

Algorithm 2 generate_sketch(L, $TESTS$)

```
 1: let $C = \emptyset$
 2: for all state table $ST$ appearing in $TESTS$ do
 3:     add all reads $ST$(f$_1$,..,f$_k$) to $C$, where f$_i$ is a field name of
        the corresponding type.
 4: end for
 5: for all match $match$ in $L$ in order do
 6:     for all test $test$ in $TESTS$ do
 7:         construct a rule $r = (match, test, ax, update)$, where
            $ax$ is a new variable for actions, and $update =$
            $[read_i{:=}(sx_i,ux_i)]$,$\forall read_i \in C$, $sx_i$, $ux_i$ fresh variables.
 8:         add rule $r$ to $sketch$
 9:     end for
10: end for
```

Search for sketch variables. Using the sketch, we can search concrete values for sketch variables, with the goal that the obtained policy table is consistent with all scenarios. To search for a consistent policy table, we perform a simple backtracking search algorithm over all sketch variables. The algorithm is shown in Algorithm 3.

The algorithm maintains a stack of sketch variables together with the values assigned to them. Whenever a sketch variable is assigned a value, it ensures that the sketch variable is pushed to the stack (line 5, 7). For each symbolic event in every scenario, the algorithm checks consistency of the first matching rule's actions (line 6). Whenever inconsistency encountered (line 9), it performs standard backtracking procedure on the stack (line 10-11); and when it is consistent, the algorithm executes the update of the rule (line 8) and carry on to the next symbolic event.

Algorithm 3 search_sketch($[SC_i]$, $sketch$)

```
 1: $stack = []$
 2: for all scenarios $SC_i$ do
 3:     for all events $e_j$ in $SC_i$ do
 4:         let $r$ be the first matching rule
 5:         initialize actions of $r$ and push the sketch variable to
            $stack$, if $r$'s actions are not set
 6:         if actions of $r_i$ are consistent then
 7:             initialize any sketch variables in $r$'s update and push
                them to $stack$, if they are not assigned values yet
 8:             apply update of $r$
 9:         else
10:             backtrack($stack$)
11:             return FAILED if $stack$ is empty else restart from
                line 2
12:         end if
13:     end for
14: end for
15: return $sketch$
```

5.4 Putting it together

Now we describe the overall synthesis algorithm (Algorithm 4), using the procedures described in previous subsections. At a high level, the synthesis algorithm enumerates sketches by increasing the number of rules, in order to generate a consistent policy table using as few rules as possible. For each sketch, it invokes Algorithm 3 to search for a consistent policy using the sketch.

Algorithm 4 synthesize($\{SC_i\}$)

```
 1: $L$ = generate_ordered_match _list($[SC_i]$)
 2: let $A$ contain all possible reads
 3: for all ($c$, m) with ascending order of m$^c$ do
 4:     for all subset $B \subset A$ with size $c$ do
 5:         $TESTS = \{\bigwedge_i read_i.state{=}v_i | \forall read_i \in B, \forall v_i \in$
            $[0,..,\mathtt{m}-1]\}$
 6:         generate_sketch($L$, $TESTS$)
 7:         if search_sketch($sketch$) returns a consistent policy table
            then
 8:             return the policy table
 9:         end if
10:     end for
11: end for
```

Example. We use the learning switch example in Figure 1 to illustrate how Algorithm 4 works. For the scenario in Figure 1, the set A contains $2^{|fields|} = 2^3 = 8$ possible reads, each of which corresponds to a combination of field names, and a state table with the corresponding type. Because two state tables with the same type can be merged into a single state table with larger states, we only construct one state table per type. Then the algorithm constructs a sketch, where each rule has c checks, and its state ranges from `0` to `m-1`. Thus the generated sketch has $\mathtt{m}^c|L|$ rules. Note that, when `m` is `1` or c is 0, the generated sketch does not use tests essentially, hence enumeration of the pair $(c, \mathtt{m})$ can start from $(1, 2)$. When picking reads from A (line 4), we pick reads with high dimension first. As an example for the learning switch, the first generated set B with size 1 in line 4 is $\{ST$(port,srcmac,dstmac)$\}$, followed by other reads with dimension two.

5.5 Additional Heuristics

In addition to the basic synthesis algorithm described above, the synthesizer has implemented other heuristics.

Lazy initialization. Algorithm 3 initializes sketch variables and pushes them to the stack as soon as applying update of the matching rule. This eager initialization could push irrelevant sketch variables to the stack and increase the search depth. For example, the variables sx_2, ux_2 in Table 4 are not used when checking consistency for any symbolic packet in Figure 1, and hence irrelevant to the consistency checking. Thus, the synthesizer takes a lazy initialization heuristic. That is, only when an uninitialized sketch variable is read from state tables, the synthesis algorithm initializes it and pushes it to the stack.

Post processing. After synthesizing a consistent policy, the synthesizer applies additional post processing to the policy table in order to simplify the policy table. These includes: (1) If a rule in the policy table is not matched by any symbolic packet in the input scenarios, this rule can be removed; (2) The synthesizer removes writes in each rule's update, if they do not change the state table; (3) When multiple rules can be merged into one without causing inconsistency, the synthesizer will merge these rules.

6. POLICY EXECUTION

Given the synthesized policy, our tool uses the interpreter to process packets on the controller. As described in Section 3.3, the interpreter simply iterates through all rules in the policy table and picks the first matched rule for the incoming packet. Then it updates all state tables based on the update of the matched rule, and instructs the switch to apply the action of the rule to the packet.

While processing packets on the controller is sufficient for executing the policy, it is not practically efficient and degrades the performance of the network. In this section, we show how the tool infers flow table rules which can be installed onto switches, thus reducing the overhead of controller and delay of packet delivery.

Our key observation is the following theorem.

THEOREM 2. *A packet can be handled on switches if and only if handling this packet on the controller does not change any state tables.*

Indeed, if a packet p is handled on switches, the controller will not be aware of the packet and thus the state tables remain unchanged. On the other hand, if p is sent to the controller for execution and the updated state tables remain the same as before, we know handling p on switches would not affect future packets execution. Therefore, it is sufficient and necessary to install rules on switches for the packets whose execution will not change current configuration of state tables.

Based on this observation, we have implemented a reactive installation approach which installs flow table rules that only match necessary fields. Moreover, to keep the installed rules up to date, we update installed rules when the policy configuration changes, and remove invalid rules on switches. Note that, one can also infer flow table rules in a proactive way based on this observation. We leave the implementation of proactive approaches to future work.

Algorithm 5 update_flowtable(p)

```
1: let rule $r$ be the matched rule for $p$ in the policy table
2: if $r$ does not update state tables, or the updated state tables remain unchanged then
3:     $match \leftarrow \langle f_{i_1}=p.f_{i_1},..,f_{i_k}=p.f_{i_k}\rangle$, for all field $f_{i_j}$ appearing in the policy table
4:     add $match \rightarrow r$.actions to the flow table, if the actions $a_j$ applied to $p$ by $r$ is supported by the switch
5: end if
6: for all installed rule $match' \rightarrow [a'_1, .., a'_l]$ in the flow table do
7:     let $p'$ be a packet matches $match'$
8:     let rule $r$ be the matched rule in the policy table for $p'$
9:     if $r$ does not update state tables or the updated state tables remain unchanged then
10:        update the installed rule to $match' \rightarrow r$.actions, if the actions $a_j$ applied to $p$ by $r$ is supported by the switch
11:    else
12:        remove the installed rule from the flow table
13:    end if
14: end for
```

Algorithm 5 shows the installation strategy. First the algorithm checks whether the matched rule r for p will change the configuration of state tables. The rule r will not change the configuration, if r does not have writes, or the updated states and values remain the same as the old ones (line 2). If executing p would not change the configuration, the algorithm installs a flow table rule $match \rightarrow [a_1, .., a_l]$ onto the switch, where $match$ specifies the values for fields related to the policy, and a_j's are the actions that should be applied to p (line 3-4). The algorithm also needs to check whether previously installed rules are still correct. For this, the algorithm repeats a similar process for each installed rule (line 6-14).

Example. Revisit the example run in Figure 4. By the interpreter's algorithm shown in Figure 3, the first packet is processed on the controller, and the state table is updated to the one shown in subfigure (c). Applying Algorithm 5, the matching rule r for p_1 would be the first rule in the policy table shown in Table 1. Since port 2 is already remembered for the srcmac A, r would not change the state table. Therefore, a flow table rule $fr_1 = \langle$port=2,srcmac=A,dstmac=B$\rangle \rightarrow$flood, which matches the port, srcmac and dstmac of p_1 is pushed down to the switch. After processing the second packet p_2, the state table is updated as in subfigure (d) and a flow table rule matching p_2 can be pushed down. Moreover, the algorithm checks the installed flow table rule fr_1. Since now p_1 would match the second rule in the policy table, and the applied action to p_1 is different from the installed flow table rule, the action of fr_1 is updated to send(1).

7. USE CASES

In this section, we demonstrate scenario-based programming for four policies. For each policy, we will show the packet-type we use, the scenarios that can be used to synthesize the desired policy, and the policy table generated from the scenarios. To this end, we manually validate that the synthesized policy is the correct policy. One can also formally verify the correctness of the generated policy against logical specifications using control plane verification tools such as Vericon [3] and Nice [5]. We plan to explore light-weight verification tools for the custom policy abstraction in the future.

7.1 Learning Switch

First, we revisit our motivating example. Recall that we can program the learning switch application for a single switch using a scenario in Figure 1. Now we show how to adapt the scenario to program the learning switch for a network. That is, the policy needs to maintain the port of each switch for hosts. To program this policy, we need a field specifying which switch the packet is located. Therefore, we use the packet-type ⟨switch : `SWITCH`, port : `PORT`, srcmac : `MAC_ADDR`, dstmac : `MAC_ADDR` ⟩. For the scenario, we simply add the switch field to each symbolic packet in the scenario in Figure 1. This modified scenario suffices for NetEgg to synthesize the network-wide learning switch policy. The scenario and synthesized policy table is shown in Figure 7 and Table 5.

scenario 1:
s_1,P_1,h_1,h_2⇒ flood
s_1,P_2,h_3,h_1⇒ send(P_1)
s_1,P_3,h_2,h_3⇒ send(P_2)

Figure 7: **Scenario-based program for the learning switch.**

match	test	actions	update
*	ST(switch, dstmac) .state=0	flood	ST(switch, srcmac) :=(1, port)
*	ST(switch, dstmac) .state=1	send(ST(switch, dstmac) .value)	ST(switch, srcmac) :=(1, port)

Table 5: **The policy table for the learning switch.**

7.2 Stateful Firewall

Now, we show how to use scenarios to program stateful firewall policies inductively.

First firewall. First, we consider a stateful firewall which protects hosts connecting to port 1 by blocking untrusted traffic from port 2. The firewall should allow all outbound packets from port 1, and only allow inbound packets from port 2 if the sender of the packet has received packets from the receiver before. For this policy, we use the packet-type ⟨port:`PORT`, srcip:`IP_ADDR`, dstip:`IP_ADDR`⟩. We start by giving two of the most intuitive scenarios shown in Figure 8. In the first scenario, the switch blocks the traffic from port 2, and the second scenario demonstrates the case where the firewall allows the traffic from port 2. It turns out that these two scenarios are sufficient to generate the desired policy, shown in Table 6.

scenario 1:
2,h_2,h_1⇒ drop

scenario 2:
1,h_1,h_2⇒ send(2)
2,h_2,h_1⇒ send(1)

Figure 8: **Scenario-based program for the first stateful firewall.**

match	test	actions	update
port=1	True	send(2)	ST(dstip,srcip) :=(1, -)
port=2	ST(srcip, dstip).state=0	drop	-
port=2	ST(srcip, dstip).state=1	send(1)	-

Table 6: **The policy table for the first stateful firewall.**

Second firewall. Now suppose we want to specify a policy such that it allows inbound traffic if the sender has received packets from any protected hosts before. One may notice that the policy should maintain a state for each host, instead of a pair of hosts. Using the scenario-based programming, we can simply adapt scenarios from Figure 8 and change the dstip of the second packet in scenario 2, as following:

modified scenario 2:
1,h_1,h_2⇒ send(2)
2,h_2,h_3⇒ send(1)

Figure 9: **The modified scenario for the second stateful firewall.**

The synthesized policy maintains a 1-dimension state table, and is shown in Table 7.

Third firewall. While we mostly focus on packetin events, NetEgg can be generalized to handle arbitrary events. In this use case, we will demonstrate how to use fields in symbolic packets to handle user-defined network events. Suppose we want to further implement a policy such that inbound traffic is allowed until a timeout event indicates the sender expires. For the policy, we need to handle a timeout event, and the expired host ip specified in the event. We can use a packet-type ⟨event:`EVENT`, eventip:`IP_ADDR`, srcip:`IP_ADDR`, dstip: `IP_ADDR`⟩. Here, the field named event specifies the type of the network event, and the field named eventip specifies the expired host. These two fields are set by the corresponding field handlers. For this policy, we can add one more scenario exhibiting the behavior of timeout, as in Figure 10. The first symbolic packet is similar to above, but since this is

match	test	actions	update
port=1	True	send(2)	ST(dstip) :=(1, -)
port=2	ST(srcip).state =0	drop	-
port=2	ST(srcip).state =1	send(1)	-

Table 7: **The policy table for the second stateful firewall.**

a packetin event, its eventip field is not applicable (we use - to denote its value). The second symbolic packet is the timeout event, which specifies that host h_2 is expired. Since the controller does not need to apply any actions to this event, we use nop for its action. The third packet from host h_2 now gets dropped. Scenario 1 and Scenario 2 can be adapted similarly from Figure 8 and Figure 9 respectively.

scenario 3:
packetin,-,1,h_1,h_2⇒ send(2)
timeout,h_2,-,-⇒ nop
packetin,-,2,h_2,h_3⇒ drop

Figure 10: **The added scenario for the third stateful firewall.**

Given the three scenarios, the desired policy can be synthesized, as in Table 8.

match	test	actions	update
event= packetin, port=1	True	send(2)	ST(dstip) :=(1, -)
event= packetin, port=2	ST(srcip) .state=0	drop	-
event= packetin, port=2	ST(srcip) .state=1	send(1)	-
event=timeout	True	nop	ST(eventip) :=(0,-)

Table 8: **The policy table for the third stateful firewall.**

7.3 TCP Firewall

In this use case, we use scenarios to program the TCP firewall that tracks the state transition of TCP handshake protocol, and only allows packets that follow the protocol. We use the packet-type that contains 5 fields: ⟨flag:`TCP_FLAG`, srcip:`IP_ADDR`, dstip: `IP_ADDR`, srcport: `TCP_PORT`, dstport: `TCP_PORT`⟩.

We first specify two scenarios describing two allowed packet traces by the TCP firewall in Figure 11. A trivial policy which allows all packets would be generated. Next, we add two scenarios describing packets which should be denied by the firewall. Checking the policy, we find an undesired behavior of the generated policy, which allows the second packet in scenario 5. We add the correct behavior as in scenario 5, and the synthesizer generates the desired policy. The generated policy table is shown in Table 9, and the state table maintains states for each tuple of srcip,dstip,srcport and dstport.

scenario 1:
SYN,ip_1,ip_2,$port_1$,$port_2$⇒ allow
ACK,ip_2,ip_1,$port_2$,$port_1$⇒ allow

scenario 2:
SYN,ip_1,ip_2,$port_1$,$port_2$⇒ allow
SYNACK,ip_2,ip_1,$port_2$,$port_1$⇒ allow
ACK,ip_1,ip_2,$port_1$,$port_2$⇒ allow

scenario 3:
ACK,ip_1,ip_2,$port_1$,$port_2$⇒ deny

scenario 4:
SYNACK,ip_2,ip_1,$port_2$,$port_1$⇒ deny

scenario 5:
SYN,ip_1,ip_2,$port_1$,$port_2$⇒ allow
ACK,ip_1,ip_2,$port_1$,$port_2$⇒ deny

Figure 11: **Scenario-based program for the TCP firewall.**

match	test	actions	update
flag=SYN	True	allow	ST(dstip, dstport,srcip, srcport) :=(1,-)
flag= SYNACK	ST(srcip, srcport, dstip, dstport) .state=0	deny	-
flag= SYNACK	ST(srcip, srcport, dstip, dstport) .state=1	allow	ST(dstip, dstport,srcip, srcport) :=(1,-)
flag=ACK	ST(srcip, srcport, dstip, dstport) .state=0	deny	-
flag=ACK	ST(srcip, srcport,dstip, dstport) .state=1	allow	-

Table 9: **The policy table for the TCP firewall.**

7.4 ARP Proxy

In this use case, we use user-defined action primitives to program the ARP proxy in scenarios. An ARP proxy caches MAC addresses associated with IP addresses, and responds to ARP requests when the requested MAC is known. We use the packet-type ⟨ srcmac:`MAC_ADDR`, arpop:`ARP_OP`, srcip:`IP_ADDR`, dstip:`IP_ADDR`⟩ to specify this use case. The first scenario we provide is similar to the scenario for the learning switch example. In addition, we provide another scenario which describes learning srcmac from ARP reply messages. Note that in the scenarios, we use the user-defined action primitive reply, which should construct an ARP reply message with the requested MAC address.

scenario 1:
h_1,request,ip_1,$ip_2$$\Rightarrow$ flood
h_3,request,ip_3,$ip_1$$\Rightarrow$ reply(h_1)
h_4,request,ip_4,$ip_3$$\Rightarrow$ reply(h_3)

scenario 2:
h_2,reply,ip_2,$ip_1$$\Rightarrow$ flood
h_3,request,ip_3,$ip_2$$\Rightarrow$ reply(h_2)

Figure 12: **Scenario-based program for the ARP proxy.**

match	test	actions	update
arpop= request	ST(dstip) .state=0	flood	ST(srcip) :=(1,srcmac)
arpop= request	ST(dstip) .state=1	reply(ST(dstip) .value)	ST(srcip) :=(1,srcmac)
arpop= reply	True	flood	ST(srcip) :=(1,srcmac)

Table 10: **The policy table for the ARP proxy.**

8. EVALUATION

We have developed a prototype of NetEgg written in Python. We evaluate NetEgg along two dimensions: (1) the feasibility of NetEgg in its ability to implement a range of SDN policies [3, 17, 12], (2) the performance and overhead of the synthesized policies, and finally, (3) correctness of the flow table rule installation strategy.

8.1 Feasibility

We explore two aspects of feasibility of NetEgg. First, is the policy generation process efficient in terms of execution time? Second, is NetEgg easy to use, in terms of the number of input scenarios required and lines of configuration code?

Table 11 summarizes our findings in terms of execution time and scenario size. We report the total number of events in the scenarios used to program each policy, and the number of scenarios. We also report the computation time of the synthesizer to generate the policy from scenarios.

	#EV	#SC	Time
maclearner1	3	1	11 ms
maclearner2	3	1	15 ms
auth	3	2	13 ms
gardenwall	5	3	52 ms
ids	3	2	15 ms
monitor	3	2	13 ms
ratelimiter	10	5	147 ms
serverlb	7	3	143 ms
stateful firewall1	3	2	12 ms
stateful firewall2	3	2	16 ms
stateful firewall3	6	3	107 ms
trafficlb	7	3	402 ms
ucap	3	2	13 ms
vmprov	3	2	24 ms
TCP firewall	9	5	64 ms
ARP proxy	5	2	49 ms

Table 11: **Network policies generated from scenarios. #SC is the number of scenarios used to synthesize the policy, #EV is the total number of events in scenarios, Time is the running time of the synthesizer.**

We make the observation that most of the scenarios are expressed in less than 5 events, with some outliers requiring up to 10 events. NetEgg is also efficient, and in all examples, requires no more than 402 ms.

We further perform a "code size" comparison, by counting the number of lines in each of our NetEgg example configurations, and compare with corresponding policies that we implemented in Pyretic and POX. Table 12 summarizes our results. We observe that NetEgg is more concise, achieving a 4× and 10× reduction in code size compared with Pyretic and POX.

	NetEgg	Pyretic	POX
maclearner2	3	17	29
stateful firewall1	3	21	58
TCP firewall	9	24	68

Table 12: **Lines of code to implement policies in different programming abstractions. We report the total number of events in scenarios for NetEgg, lines of code for Pyretic and POX implementations.**

8.2 Performance Overhead

NetEgg uses the policy table as the policy abstraction, and a generic interpreter to execute the policy table. Unlike hand-crafted implementations which can be customized to policies, generic execution of our abstraction of policies may incur additional overhead. We evaluate the generic execution engine of NetEgg using a combination of targeted benchmarks and end-to-end evaluation.

8.2.1 Cbench Evaluation

We first use the Cbench [21] tool to evaluate the response time of three policy implementations.

Experiments. We emulate one switch in Cbench, which sends one packet-in request to the controller as soon as it receives a reply for last sent request. The response time corresponds to the time between sending out a request and receiving its reply, which hence includes the execution time of policy implementations. For comparison, we also evaluate the policies' implementations in POX.

Results. Figure 13 shows the response time for the policy implementations in POX and NetEgg. We note that in all cases, the differences in response times between the POX and NetEgg versions are within 12%. In the case of MAC learning and stateful firewall, the differences are negligible (<1%). We observe that the response time between implementations in POX and NetEgg is comparable, which suggests our policy abstraction incurs reasonably small overhead on execution.

8.2.2 End-to-end Performance

Our next set of experiments aim to validate that the synthesized implementation closely matches the hand-crafted implementation on end-to-end performance for network applications such as HTTP.

Experiments. We emulate a fattree topology [1] in Mininet,

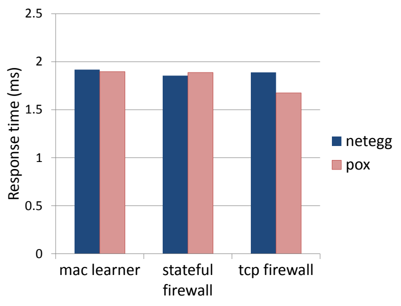

Figure 13: **Response time for POX and NetEgg implementation**

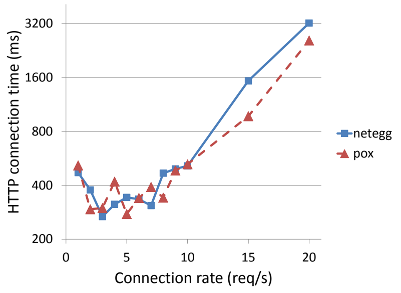

Figure 14: **HTTP connection time.**

which consists of 20 switches and 16 hosts. We setup a HTTP server on one host, and run httperf on all other hosts as clients. Httperf sends HTTP requests from the clients to the server, and measures the HTTP connection time for each request, which is the time between a TCP connection is initiated and it is closed. We run httperf with different rates of sending requests, and the same number of connections (e.g. at rate 5 request/second, httperf issues 5 requests per client). Each run starts from the initial network state. On the controller side, we run the MAC learner policy using two implementations: POX and NetEgg.

Results. Figure 14 reports the average connection time over all 15 clients. The x-axis is the rate of HTTP requests issued by the clients. As expected, the connection time under the NetEgg implementation matches closely to that under hand-crafted POX implementation. These results suggest our synthesized implementation is able to achieve comparable end-to-end performance as hand-crafted implementations. This also further verifies that execution of our policy abstraction incurs small overhead, and our flow table rule installation is efficient.

8.3 Rule Installation

To achieve realistic performance, our interpreter infers and installs flow table rules. We validate the correctness of our rule installation strategy using emulation-based experiments.

Experiments. We run the synthesized MAC learner policy on the controller, and emulate a simple topology with a single switch connected with 300 hosts in Mininet [13]. We partition these hosts into two groups, with 150 hosts per group. Every host in a group sends 100 ping messages to another host in the other group with 1 message per second. For comparison, we run the set of experiments under two settings, one with flow table installation and one without.

Results. We plot the average RTT for all ping messages over time in Figure 15. The red line corresponds to the policy implementation without installing flow table rules. This implementation has a high RTT consistently over time, due to the fact that every packet is sent to the controller. The blue line corresponds to the case with installation. We observe that only the first message experiences high latency, and subsequent messages has significantly smaller RTT below 0.1 ms. This fact suggests that our installation strategy is able to infer flow table rules from the first incoming packet-in event, and correctly install the rules onto the switch. Hence, subsequent packets are all processed by the switch.

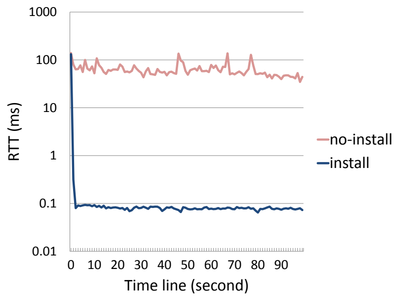

Figure 15: **Effects of flow table rule installation.**

9. RELATED WORK

This work builds upon our position paper [24] in several significant ways. The NetEgg model presented in Section 4 is an extension of the earlier simplified model in our position paper. We have added new extensions that automatically push flow table rules from the synthesized policies to the data plane. Moreover, we have developed a complete prototype, added several use cases and experiments to validate the NetEgg tool.

SDN programming languages. SDN domain-specific languages [6, 16, 23, 15, 2, 18, 20, 19] make programming policies easier using high-level abstractions. Our approach is different – we target at designing intuitive abstractions for network operators who can take advantages of their domain expertise and generate examples for our tool.

Programming by examples. Our work is motivated by related work in the formal methods community in programming by examples. [10, 9, 11] implement finite-state reactive controllers from specification of behaviors. [8] generates string transformation macros in Excel from input/output string examples. [22] uses both symbolic and concrete example to synthesize distributed protocols.

Our work is similar in spirit to above works, but technically different. Our input examples and target program are designed specific to the SDN domain, and have different characteristics, which require different synthesis algorithms.

Policy abstractions. Recent work proposes new abstractions of policies based on state machines [17, 4, 12]. These work shows the state machine abstraction benefits from fast execution on data plane [17, 4], and conciseness of programming [12]. Our abstraction of policy tables is similar in spirit to these state machine abstractions and thus can benefit from the advantages of previous work. But however, our work focuses on providing an intuitive programming framework which can generate policies directly from examples.

10. CONCLUSION

In this paper, we explore the design and implementation of scenario-based programming that automatically generates network policy implementations from example scenarios.

In our evaluation, we observe that NetEgg is expressive and can support a wide range of policies at low overhead compared to imperative implementations. Using NetEgg, we are able to implement several policies in an intuitive and concise manner that is significantly more compared than alternative approaches. This approach lends itself naturally to rapid prototyping and shortening the design/implementation iteration cycle.

NetEgg is designed for state-oriented policies, and does not suit well for objective-oriented policies, such as shortest-path routing and traffic engineering. We leave the exploration of programming such policies using scenarios to future work.

Moving forward, we plan to carry out a user study to gather feedback from a larger pool of users. We also observe that NetEgg is slightly cumbersome for supporting policies that depend on stateful aggregate values, for example, take a particular action if a threshold is met. We plan to explore the combination of imperative languages with NetEgg, or using NetEgg with a database query language for enabling such complex policies. We plan to explore the use of formal verification techniques to check scenarios against high-level properties. Finally, while the paper focuses on SDN polices, the programming model is not restricted to SDN, and we plan to apply this approach to other settings, for example Internet and wireless routing policies.

11. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM CCR*, 2008.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *POPL*, 2014.

[3] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *PLDI*, 2014.

[4] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *SIGCOMM CCR*, 2014.

[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, et al. A nice way to test openflow applications. In *NSDI*, 2012.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, 2011.

[7] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 1978.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, 2011.

[9] D. Harel. Can programming be liberated, period? *Computer*, 2008.

[10] D. Harel and R. Marelly. *Come, let's play: Scenario-based programming using LSCs and the Play-Engine*, volume 1. Springer, 2003.

[11] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *CACM*, 2012.

[12] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *NSDI*, 2015.

[13] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*, 2010.

[14] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. In *CACM*, 2009.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, 2012.

[16] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, et al. Composing software defined networks. In *NSDI*, 2013.

[17] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *HotSDN*, 2014.

[18] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

[19] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN*, 2013.

[20] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, 2014.

[21] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-defined networks. In *HotICE*, 2012.

[22] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: specifying protocols with concolic snippets. In *ACM SIGPLAN Notices*, 2013.

[23] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying sdn programming using algorithmic policies. In *SIGCOMM*, 2013.

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