# NetEgg: Programming Network Policies by Examples [*]

Yifei Yuan
University of Pennsylvania
yifeiy@cis.upenn.edu

Rajeev Alur
University of Pennsylvania
alur@cis.upenn.edu

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

## ABSTRACT

The emergence of programmable interfaces to network controllers offers network operators the flexibility to implement a variety of policies. We propose NetEgg, a programming framework that allows a network operator to specify the desired functionality using example behaviors. Our synthesis algorithm automatically infers the state that needs to be maintained to exhibit the desired behaviors along with the rules for processing network packets and updating the state. We report on an initial prototype of NetEgg. Our experiments evaluate the proposed framework based on the number of examples needed to specify a variety of policies considered in the literature, the computational requirements of the synthesis algorithm to translate these examples to programs , and the overhead introduced by the generated implementation for processing packets. Our results show that NetEgg can generate implementations that are consistent with the example behaviors, and have performance comparable to equivalent imperative implementations.

**Categories and Subject Descriptors:** C.2.3 [Computer-Communication Networks]: Network Operations

**General Terms:** Design; Languages; Management

## 1. INTRODUCTION

Recent emergence of software-defined networking (SDN) provides a unified programming interface and offers the flexibility for network operators to program network policies. Major router vendors provide APIs to enable programmability (either vendor specific or tied to OpenFlow), and cloud orchestration systems such as OpenStack also comes with package for service chaining and network virtualization.

However, a key challenge that has yet to be addressed is providing an intuitive abstraction that allows network operators to program their own protocols and policies, hence taking advantage of the new programming interface. In a typical SDN network, centralized controller connects and controls switches in the network using an SDN protocol. A natural approach that has been proposed in recent years uses high-level domain specific languages (e.g. Frenetic [3], Pyretic [11], NetKAT [1], Maple [15], NetCore [10] ,declarative networking [9] ) that raise the level of abstraction and make it easier to program controllers with orders of magnitude reduction in code sizes.

However, we argue that domain specific languages, while useful, are perhaps not the most intuitive approach for network operators, who often times are the consumers of router solutions, and while are experts at configuring routers, may not be trained to program them, particularly in domain specific languages that have higher learning curves.

In this paper, we investigate an alternative approach based on *synthesizing* an implementation automatically from *examples*. Our proposed NetEgg is based on the observation that network operators typically like to work out examples using timing diagrams and topologies, when coming up with new network configurations and designs. Typically, these examples would be generalized into design documents, followed by pseudocode and then finally implementation. NetEgg aims to facilitate the entire process, by generating implementations directly from the examples themselves.

Specifically, this paper makes the following contributions:

**Proof-of-concept design and implementation.** We have developed the NetEgg tool, that 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.Given the scenarios as input, NetEgg automatically generates a controller program that is consistent with example behaviors, including inferring the state that needs to implement the network policy, interested fields associated with the state and rules for processing packets and updating states.

**Validation.** We validate the NetEgg tool to synthesize SDN programs that use the POX controller directly from examples. NetEgg is agnostic to the choice of SDN controllers, and can also be used in non-SDN settings. We demonstrate that NetEgg is able to synthesize network policies using a small number of examples in less than 1 second. The synthesised controller program has little overhead and achieves comparable performance to equivalent imperative programs implemented manually in POX.

[*]This research was partially supported by NSF Expeditions in Computing award CCF 1138996, AFOSR Young Investigator award FA9550-12-1-0327 and NSF CNS-1218066.

*HotNets-XIII,* October 27–28, 2014, Los Angeles, CA, USA.
Copyright 2014 ACM 978-1-4503-3256-9/14/10$15.00
http://dx.doi.org/10.1145/2670518.2673879 .

## 2. ILLUSTRATIVE SCENARIO

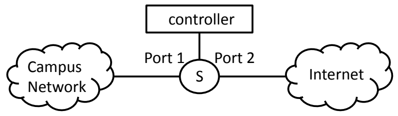

**Figure 1: A stateful firewall example.**

To illustrate the use of NetEgg, we consider the example of a stateful firewall in Figure 1. In this example, we have an SDN-enabled switch S that connects to the campus network via port 1, and the Internet via port 2. The network operator wishes to configure the switch's security policy as follows: (1) all outgoing campus network traffic is allowed to go through, and (2) only selected trusted hosts in the Internet are allowed send traffic to the campus network. A host in the Internet is considered as trusted if some host in the campus network has sent traffic to it before.

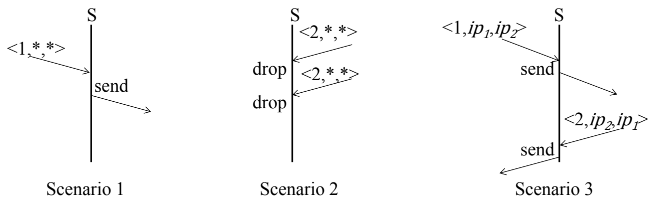

**Figure 2: Three stateful firewall scenarios. In the scenarios, we use a 3-tuple to denote a packet, with the fields being incoming port, source IP and destination IP address. We use $*$ to denote wildcards.**

Figure 2 shows three scenarios described by network operators:

- **First scenario.** The network operator describes the case where a packet arriving on port 1 should be sent out regardless of the source and destination addresses.
- **Second scenario.** A packet arriving on port 2 is dropped if there is no prior knowledge to indicate that a campus host has ever contacted the sender.
- **Third scenario.** The first packet received by the switch is from port 1, and its source/destination IP address is $ip_1$ and $ip_2$. Note that instead of specifying the concrete IP addresses, the network operator uses symbolic constants. A symbolic IP address represents any possible IP address. When the second packet reaches the firewall on port 2, it is sent to the campus network, since the host with $ip_2$ received packets from the host with $ip_1$ and thus is considered as a trusted host.

Given the scenarios in Figure 2, NetEgg generates the program that implements the desired policy. The program can be written as an SDN controller program, or compiled into configuration changes on the switch itself. As part of the program generation, NetEgg automatically generates the data structures and code necessary to implement the policy expressed by these scenarios.

## 3. NETEGG OVERVIEW

In this section, we first present an overview of the NetEgg tool. For ease of presentation, we consider an implementation running on a centralized controller, rather than on the switches. We also assume all packets are sent to the controller for execution. We leave relaxation of the restriction in future work.

From the input scenarios, NetEgg generates a *policy table*, multiple *state tables*, and a controller program. This process is highlighted in Figure 3. The network operator describes scenarios about the desired network policy to NetEgg, which tries to generate an SDN controller program consistent with the behaviors described in the scenarios. The generated program uses a set of *state tables* and a *policy table*. State tables allow the controller program to remember the history of a policy execution, in between arrival of messages and local events. The policy table dictates the actions for processing incoming packets and updates to state tables for various cases. The controller program takes as input incoming packets, looks up the policy table, which will determine state table updates and actions to be applied to the packets.

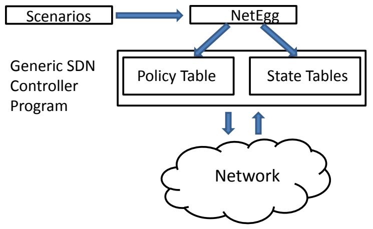

**Figure 3: Tool architecture.**

Revisiting the firewall example from the previous section, we first describe the tables that are generated by our tool, before describing the controller program. We will describe how these tables are generated by the tool in Section 5.

### 3.1 State Tables

In our example, the firewall needs to remember all the destination hosts that have been contacted recently as trusted hosts. Hence, the generated program maintains a state table $ST$ which stores a state for each IP address. NetEgg automatically derives the fact that for a given IP address $ipaddr$, the state of $ipaddr$ in $ST$ is either `0` or `1`, indicating that $ipaddr$ is untrusted or trusted respectively. Initially, the program assigns all states in the table to be `0`. The program accounts for three cases: 1) When it gets a packet from port 1, it sends the packet out, and moreover, it sets the state associated with the destination IP to `1`; 2) When it gets a packet from port 2, it checks whether the state associated with the source IP is `1`. If so, the packet is allowed to get through; 3) If the state associated with the source IP of the packet from port 2 is `0`, it is dropped.

### 3.2 Policy Tables

The above 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 firewall example.

The policy table contains three *rules*, corresponding to the three rows in the table. These three rules correspond to the three cases in the program described above. Every rule has four components: *Match*, *Tests*, *Action* and *Up-*

| Match | Tests | Action | Updates |
|---|---|---|---|
| port=1 | - | send | $ST$(dstip):=`1` |
| port=2 | $ST$(srcip)=`1` | send | - |
| port=2 | $ST$(srcip)=`0` | drop | - |

**Table 1: The policy table for the stateful firewall.**

*dates*. Match specifies the packet fields and corresponding values that a packet should match. Tests check whether the state associated with some fields in a state table is a certain value. For example, $ST$(srcip)=`1` in the second rule checks whether the state associated with the source IP address of the packet is `1` in $ST$. Action defines the action that is applied to matched packets. Updates change the state associated with some fields in a state table to a certain value. For example, $ST$(dstip):=`1` in the first rule changes the state associated with the destination IP address of the packet to `1` in $ST$.

## 3.3 Generic Controller Program

With the policy table, a generic controller program is used for processing incoming packets at the controller. The generic controller program 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 Tests of the rule are satisfied. The first matched rule applies Action to the packet, and state tables are updated according to Updates.

Figure 4 shows an illustrative execution of an incoming packet trace shown in subfigure (a). Initially, all entries in the state table $ST$ are `0`, as shown in subfigure (b). When the first packet $p_1$ is received by the switch, it is matched against each rule in Table 1 in order. The first matched rule is the third rule, since $p_1$ matches the Match (port=2) and the state of the field srcip of $p_1$ in $ST$ is `0`, satisfying the Tests ($ST$(srcip)=`0`) in the rule. Therefore, the rule applies the Action and drops $p_1$. Since the third rule does not have any Updates, the state table remains the same as in subfigure (b). The second packet $p_2$ in the trace matches the first rule, since the rule specifies the Match without any Tests. The program sends $p_2$ through the firewall and also applies Updates ($ST$(dstip):=`1`) of the rule. Thus, the entry indexed by the dstip field of $p_2$ in the state table $ST$ is changed to `1`, as shown in subfigure (c). The third packet $p_3$ is sent, since the state of its source has been changed to `1` and it matches the second rule. Finally, the state of the source of $p_4$ is `0` and thus it matches the third rule and gets dropped.

$p_1$:<port=2, srcip=1.2.3.1, dstip=10.0.0.2>
$p_2$:<port=1, srcip=10.0.0.1, dstip=1.2.3.1>
$p_3$:<port=2, srcip=1.2.3.1, dstip=10.0.0.2>
$p_4$:<port=2, srcip=1.2.3.2, dstip=10.0.0.1>

(a) An example packet trace

| IP | State |
|---|---|
| 1.2.3.1 | 0 |
| 1.2.3.2 | 0 |

(b) The initial state table

| IP | State |
|---|---|
| 1.2.3.1 | 1 |
| 1.2.3.2 | 0 |

(c) The changed state table

**Figure 4: An illustrative execution.**

## 4. FORMAL DEFINITIONS

In this section, we provide a more formal definition of the policy and state tables, which will allow us to describe how the policy tables are generated for each set of scenarios in the next section.

### 4.1 Scenarios

In our programming framework, variables and fields of packets are typed. Examples of base types we use are `bool`, enumerated types such as {`0`, `1`}, `int`[1..$n$] (integers 1 through $n$), `IP_ADDR` (set of IP addresses). We also use `Act` to denote the set of actions for processing packets, and in our examples, `Act` equals {drop, send}. 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 : `int`[1..2], srcip : `IP_ADDR`, dstip : `IP_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`, $*$ representing wildcard value, or a variable $x$ of type `T`. A symbolic packet specifies a symbolic value for each field.

An *event* is a pair $(sp, a)$, where *sp* is a symbolic packet and $a$ is an action in `Act`. A *scenario* is a finite sequence of events. A scenario-based program is a finite set of scenarios.

With this notation, scenario 3 of Figure 2 corresponds to: ($\langle$port=1, srcip=$ip1$, dstip=$ip2$$\rangle$, send), ($\langle$port=2, srcip=$ip2$, dstip=$ip1$$\rangle$, send).

A scenario is called concrete if all the symbolic packets appearing in the scenario have only concrete values. A (symbolic) scenario can be viewed as a short-hand for a set of concrete scenarios. This set is obtained by replacing each wild-card by every possible value of the corresponding type, and each variable by every possible value of the corresponding type. Note that if the same variable appears in multiple symbolic packets in a scenario, then it gets replaced by the same value. Thus, the symbolic scenario 3 of Figure 2 corresponds to $n^2$ concrete scenarios if the type `IP_ADDR` contains $n$ distinct addresses.

### 4.2 State Tables

A $d$-dimensional state table $ST$ is a key-value map of type $\texttt{T}_1 \times .. \times \texttt{T}_d \rightarrow S$, for some base types $\texttt{T}_i$ and an enumerated type $S$. We call $S$ the state set, and state values for the values in $S$. We use $dim(ST)$ to denote the dimension of $ST$. In our example, the only state table maintained is of type `IP_ADDR`$\rightarrow${`0`,`1`}, and its dimension is 1.

The operations we allow to a state table are *tests* and *updates*. A test (update, resp.) of a state table $ST$ of type $\texttt{T}_1 \times .. \times \texttt{T}_d \rightarrow S$ is of the form $ST(\mathrm{f}_1, .., \mathrm{f}_d)$=$s$ ($ST(\mathrm{f}_1, .., \mathrm{f}_d)$:=$s$, resp.), where $\mathrm{f}_i$ is a packet field of type $\texttt{T}_i$ and all $\mathrm{f}_i$'s are different, $s \in S$ is a state value. In our example, $ST$(srcip)=`0` is a test and $ST$(dstip):=`1` is an update of $ST$ with the field srcip and dstip respectively.

To evaluate a state table, we define a valuation $v$ of a state table $ST$ of type $\texttt{T}_1 \times .. \times \texttt{T}_d \rightarrow S$ to be a function mapping a key of type $\texttt{T}_1 \times .. \times \texttt{T}_d$ to a state value in $S$. For example, the initial valuation of the state table in our example maps every IP address to `0`. A test $ST(\mathrm{f}_1, .., \mathrm{f}_d)$=$s$ is true for a packet $p$ under a valuation $v$ iff $v(p.\mathrm{f}_1, .., p.\mathrm{f}_d) = s$, where $p.\mathrm{f}_i$ indicates the value of the field $\mathrm{f}_i$ of $p$. An update $ST(\mathrm{f}_1, .., \mathrm{f}_d)$:=$s$ for a packet $p$ under a valuation $v$ returns a new valuation $v'$ such that $v'(p.\mathrm{f}_1,..,p.\mathrm{f}_d) = s$ and $v'(k) = v(k)$ for other keys. In the example in Figure 4, $ST$(srcip)=`0` is true for $p_1$ under the initial valuation (subfigure (b)) of $ST$. $ST$(dstip):=`1` for $p_2$ under the initial valuation returns the new valuation as shown in subfigure (c).

## 4.3 Policy Tables

Given a set of state tables $\mathcal{T}$, a rule $r = (\rho, \alpha, a, \beta)$ based on $\mathcal{T}$ has four components. $\rho$ is of the form $\langle \mathrm{f}_1 = \mathrm{m}_1, .., \mathrm{f}_k = \mathrm{m}_k \rangle$, where $\mathrm{f}_i$ is a name of a packet field, and $\mathrm{m}_i$ is either a value for the field or a wildcard. A packet $p$ matches $\langle \mathrm{f}_1 = \mathrm{m}_1, .., \mathrm{f}_k = \mathrm{m}_k \rangle$ iff $\mathrm{m}_i$ is wildcard or $p.\mathrm{f}_i = \mathrm{m}_i$, for $i = 1$ to $k$. $a$ is an action in `Act`. $\alpha$ is a conjunction of tests and $\beta$ is a sequence of updates, where each test/update is of some state table in $\mathcal{T}$. The four components $\rho, \alpha, a, \beta$ correspond to Match, Action, Tests, Updates in Table 1 respectively. As an example, each of the last 3 rows in Table 1 is a rule based on the corresponding state table. A *policy table* based on $\mathcal{T}$ is an ordered list of rules, and every rule is based on $\mathcal{T}$.

A valuation $\mathcal{V}$ of a set of state tables $\mathcal{T}$ is all the valuations of each state table $\mathcal{T}$. A packet $p$ matches a rule $(\rho, \alpha, a, \beta)$ under a valuation $\mathcal{V}$ iff $p$ matches $\rho$ and every test in $\alpha$ is true for $p$ under the corresponding valuation in $\mathcal{V}$. Suppose the first matched rule for a packet $p$ under a valuation $\mathcal{V}$ is $r = (\rho, \alpha, a, \beta)$. Then $a$ will be executed on $p$ and every update in $\beta$ will be executed. We denote the execution for packet $p$ as $\mathcal{V} \xrightarrow{p/a}_{PT} \mathcal{V}'$, with $\mathcal{V}'$ the new valuation.

Given a concrete scenario $SC = (p_1, a_1), .., (p_m, a_m)$, a policy table $PT$ is *consistent* with $SC$ iff $\mathcal{V}_{i-1} \xrightarrow{p_i/a_i}_{PT} \mathcal{V}_i$ for $i = 1, .., m$, where $\mathcal{V}_0$ is the initial valuation in which every valuation maps every key to an initial state value in $S$. A policy table is consistent with a symbolic scenario iff it is consistent with all concrete scenarios represented by the symbolic scenario.

## 5. POLICY TABLE GENERATION

Given scenarios describing a stateful policy by the network operator, the goal of NetEgg is to generate a controller program consisting of a set of state tables and a policy table that is consistent with the given scenarios. We refer to this as the *policy learning problem*, formally defined as follows:

**Policy learning problem.** Given scenarios $SC_1, ..., SC_n$ and two natural numbers $d$ and $m$, the policy learning problem seeks a set of state tables $\mathcal{T}$ and a policy table $PT$ based on $\mathcal{T}$, such that (1) $\forall ST \in \mathcal{T}$, if $ST$ is of type $\mathtt{T}_1 \times .. \times \mathtt{T}_t \rightarrow S$, then $t \leq d$ and $|S| \leq m$, (2) all state tables in $\mathcal{T}$ have different types and (3) $PT$ is consistent with all scenarios $SC_i$.

Note that the natural number $d$ is the maximum dimension of state tables and $m$ is the maximum number of states that are used by state tables. The second condition is a natural assumption, since multiple state tables with the same type can be merged into a single state table using a larger state set. NetEgg generates a consistent controller program by solving the policy learning problem repeatedly by increasing the values of $d$ and $m$. NetEgg also allows network operators to provide values for $d$ and $m$.

We first show the hardness of the policy learning problem.

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

PROOF SKETCH. Reduce from 3SAT. □

Now we present a synthesis algorithm to the policy learning problem. At a high level, the algorithm guesses a policy table along with state tables within the space of all potential consistent policy tables, and checks consistency against all the scenarios for the guessed policy table. When terminating, the algorithm either returns a consistent policy table, or reports no such policy table found.

**Guessing a policy table.** In order to be able to guess all possible consistent policy tables satisfying condition (1) and (2) above, we generate a *sketch of policy table* for the input scenarios. A sketch of policy table covers all possible Matches and Tests in order for every concrete packet to match a rule in it, and we leave Action and state values in Updates as ?, meaning undetermined. With the sketch of policy table, guessing a possible consistent policy table reduces to guessing values for ? in the sketch of policy table.

As an example for the scenarios in Figure 2 when $d$ is 1 and $m$ is 2, the Match of a rule in the sketch of policy table is either port=1 or port=2. Since we have no knowledge about the state tables, we consider all state tables with dimension 1, i.e. $ST_1$ :`int[1..2]`$\rightarrow$ $\{$`0`,`1`$\}$ and $ST_2$ :`IP_ADDR`$\rightarrow$ $\{$`0`,`1`$\}$. Besides, we consider all possible state values a test can take. Therefore, every rule has 3 tests $ST_1$(port)=$x \wedge ST_2$(srcip)=$y$ $\wedge ST_2$(dstip)=$z$ , and $x, y, z$ taking state values from $\{$`0`,`1`$\}$. The Action of every rule in the sketch of policy table is represented as ?, meaning undetermined. The Updates of every rule are represented as [$ST_1$(port):=?, $ST_2$(srcip):=?, $ST_2$(dstip):=?][1]. Thus we generate 16 rules in total in the sketch of policy table.

Algorithm 1 shows the algorithm for generating the sketch of policy table, given the scenarios $SC_1, .., SC_n$ and $d$, $m$.

It first generates the set $M$ of all possible Matches (line 1). Algorithm generate_Matches generates a Match $\langle \mathrm{f}_1$=$\mathrm{m}_1$,.., $\mathrm{f}_k$=$\mathrm{m}_k\rangle$ for each (symbolic) packet $\langle \mathrm{f}_1$=$\mathrm{v}_1$,..,$\mathrm{f}_k$=$\mathrm{v}_k\rangle$ appearing in a scenario. If $\mathrm{v}_i$ specifies a concrete value, it sets $\mathrm{m}_i$ to be the same value. If $\mathrm{v}_i$ is $*$ or a variable, we set $\mathrm{m}_i$ to be $*$, since $\mathrm{m}_i$ needs to match a variety of concrete values.

Next, it generates all possible tuples of fields which can be used in Tests (line 2). Algorithm generate_field_tuples returns all $t$-tuples $(\mathrm{f}_1, .., \mathrm{f}_t)$ of fields with $t \leq d$.

Finally, Algorithm 1 generates a list of rules by combining Matches and each conjunction of tests (line 3-12). For each tuple in $F$, a state table of corresponding type is generated (line 6-7). By ranging over all state values for all tuples in $F$ (line 3), the Tests of a rule is generated as a conjunction

[1]For ease of presentation, the Updates here do not cover the case where one entry of $ST_2$ is updated by $ST_2$(dstip):=? and then $ST_2$(srcip):=? in the same rule. This can be achieved by adding another update $ST_2$(srcip):=? to the end of Updates and introducing an auxiliary value - for ?, meaning no change.

of tests of the corresponding state table $ST$ and indexed by a tuple in $F$ (line 8). We leave the Action and state values in Updates in rules as ?.

**Algorithm 1** generate_sketch($[SC_i], d, m$)

```
1:  $M$=generate_Matches($SC_1, .., SC_n$)
2:  $F$=generate_field_tuples($d$)
3:  for all tuples of state values $(v_1, .., v_{|F|})$ s.t. $v_i \leq m$ do
4:      for all $\rho$ in $M$ do
5:          for $i$-th tuple (f$_1$,..,f$_t$) in $F$ do
6:              let $(\mathtt{T}_1, .., \mathtt{T}_t)$ be the corresponding type
7:              build a state table $ST$ of type $\mathtt{T}_1 \times .. \times \mathtt{T}_t \rightarrow S$,
                if no such table exists
8:              $\alpha = \alpha \wedge ST$(f$_1$,..,f$_t$):=$v_i$
9:              append $ST$(f$_1$,..,f$_t$):=? to $\beta$
10:         end for
11:         add the rule $(\rho, \alpha, ?, \beta)$ to the sketch of policy table
12:     end for
13: end for
```

**Checking consistency.** After guessing a policy table, we need to check whether it is consistent with the input scenarios. When all scenarios are concrete, we can simply simulate the execution of the guessed policy table for every concrete scenario. The challenge here is that with symbolic scenarios, we need to ensure that every concrete scenario represented by a symbolic scenario is consistent with the policy table. Clearly, generating all concrete scenarios from given symbolic scenarios is not feasible, since the range for a field is unbounded in the worst case. To address this, we make two assumptions. First, we assume that in a scenario a variable only takes values different from every concrete value of the corresponding type appeared in the scenario, and variables of the same type in a scenario take different values. With this assumption, instead of dealing with all possible concrete scenarios, we can simply replace a variable by any concrete value which is different from both existing concrete values in the scenarios and values for other variables, since all these concrete values for the variable serve as the same role in the scenarios. Second, for every packet field, we assume that either all symbolic packets in all scenarios have wildcards in the field, or all symbolic packets have non-wildcards in the field. With this assumption, we can safely ignore all fields with wildcards in both the Tests and Updates in a rule. Note that, these two assumptions can be achieved by prepossessing the scenarios. For example, the scenario 1 of Figure 2 can be translated to two scenarios: [⟨port=1,srcip=$ip_1$,dstip=$ip_1$⟩,send], [⟨port=1,srcip=$ip_1$,dstip=$ip_2$⟩,send].

While we can simply enumerate values for ? in order to guess a policy table, we optimize the process by checking consistency whenever a new value is assigned to ? in the sketch of policy table. If the check fails, we try the next value for the same ?. When all values for a ? fail the consistency check, we backtrack to last assigned ?. We omit the discussion of other optimizations due to the limit of space.

There are $O(|M|m^{|F|})$ rules in sketch of policy table, and each rule has $|F| + 1$ undetermined values. Thus the worst complexity of our algorithm is $O(2^{|M|m^{|F|}(|F|+1)\log m})$.

## 6. EVALUATION

To validate the NetEgg design, we have implemented a prototype of NetEgg, and used it to generate POX controller implementations. NetEgg itself is agnostic to the choice of the controller – requiring only the implementation of the generic controller program in Section 3.3 within any controllers.

Our experiments are carried out in a virtual machine running Ubuntu 12 (64-bit). The host machine has i7-3520M 2.90GHz processors and 8 GB memory.

### 6.1 Tool Generality

Our first evaluation criteria is on the generality of NetEgg to support a range of network policies. We studied 6 different policies considered in literature [12, 2]. In the experiment, we manually validate that the synthesized program meets the desired policy. Table 2 summarizes our results. We leave a systematic evaluation as future work.

| Policy | # SC | # EV | Time | $d$ | $m$ |
|---|---|---|---|---|---|
| Stateful firewall (Host) | 3 | 1.67 | 2 ms | 1 | 2 |
| Stateful firewall (TCP) | 5 | 3.4 | 17 ms | 2 | 3 |
| Learning switch | 3 | 3.33 | 9 ms | 1 | 3 |
| Flow affinity | 4 | 2.5 | 16 ms | 1 | 2 |
| Resonance | 5 | 4.6 | 180 ms | 1 | 3 |
| TCP handshake | 4 | 2.75 | 6 ms | 2 | 3 |

**Table 2: Network policies generated from examples.# SC is the number of scenarios, # EV is the average number of events per scenarios, Time is the running time for the synthesis algorithm, $d$ is the maximum dimension of state tables, $m$ is the number of states in $S$.**

**Stateful firewall(Host).** This is the motivating example we have shown in Section 2.

**Stateful firewall(TCP).** The stateful firewall(TCP) is similar to the stateful firewall in the motivating example, except that it only allows incoming traffic from Internet in an established TCP connection. The policy requires a 2-dimension state table which maintains connection states for each source-destination pair. When provided the knowledge of the exact dimension of the table, we are able to generate the policy with 5 scenarios and 17 events in total. In case where no exact dimension provided, we are able to generate the policy using 21 scenarios and 60 events in total, within 2.28 s. Additional scenarios are used to rule out programs using small state tables, due to the fact that our heuristic searches small state tables first.

**Learning switch.** The learning switch learns MAC-port mappings in the network. When a packet arrives, a learning switch remembers its source MAC address with the associated incoming port. When the packet's destination MAC is associated with some port, the switch forward it to the port, otherwise flood it. We can generate a learning switch with

2 ports using 3 scenarios with 10 events in total. Note that our abstraction does not support MAC-port mappings, however, we can use the state of a MAC address to represent the incoming port. Thus the policy needs 3 states for a MAC: `0` stands for no port learnt yet, `1(2` resp.) stands for the associated port is 1 (2 resp.). We plan to generalize our abstraction to support arbitrary mappings in the future.

**Flow affinity.** We consider a simple policy that avoids connection disruptions during load balancing. We consider the case where a switch forwards traffic to servers that connect to it. Initially the switch forwards traffic to server A. After getting a signal of shifting traffic, the switch starts forwarding traffic from hosts, which never communicated with any servers, to server B, while keeps forwarding traffic to server A if the sender has communicated to it before. This policy requires two state tables, a 1-dimension state table storing states (communicated with server A or not) for every host, and another 0-dimension state table (i.e. a single entry) storing states about whether the signal is received.

**Resonance.** We generated a simplified version of Resonance [13] using 5 scenarios and 23 events. In our simplified version, a host can be in one of the three states: Registered, Authenticated, Operational. A host can transition from one state to another when the associated server with current state approves. Only the traffic from hosts in operational state is allowed to enter the network.

**TCP handshake.** Our last example considers the state transition of three-way handshake of TCP. The desired policy needs to maintain the TCP state for each connection, and drops any packets that do not satisfy the TCP state transition. Like stateful firewall(TCP), we need a 2-dimension state table to implement desired policy. With the knowledge of the exact dimension, we can generate the desired policy using 4 scenarios and 11 events in total. Without such information, we are still able to generate the desired policy in 238 ms using 12 scenarios and 32 events in total.

We make the following observations from our experiments. First, the number of scenarios are small (ranging from 3-5), and each scenario has 3-5 events on average. NetEgg is able to generate the policy table in only a small fraction of a second. Moreover, we validate that the generated POX implementations are faithful to the example scenarios.

## 6.2 Performance Overhead

Our next goal is to evaluate the efficiency of generated programs. Recall that we have a generic SDN controller program that is customized via the policy and state tables. Unlike a carefully hand-crafted implementation, our program may incur additional overhead.

To quantify this additional overhead, we conduct preliminary evaluations using the learning switch example. For comparison, we have implemented an equivalent imperative version of learning switch in POX. In order to quantify only the overhead of the implementation of the controller program, we forward all traffic to the controller in both programs. We run a network with one switch and two hosts in Mininet [8]. We use ping and iperf to test the latency and throughput from one host to the other, respectively. Table 3 presents the average latency over 100 pings and average throughput over 10 executions of iperf.

| | **Latency** | **Throughput** |
|---|---|---|
| NetEgg-LS | 36.08ms | 4.09Mbps |
| POX-LS | 34.08ms | 4.48Mbps |

**Table 3: Performance.**

The results show that the performance of the automatically generated implementation is comparable to the imperative implementation with equivalent latency and throughput. The overhead, introduced by rule matching, state tests of incoming packets and updates to state tables, is relatively negligible compared to other overheads in POX.

## 7. RELATED WORK

In addition to SDN DSLs described in Section 1, our work is closely related to parallel work in the formal methods community in programming by examples [6, 5, 7]. These work typically implements finite-state reactive controllers from specification of behaviors. A good example is recent work done in Excel, whereby string transformation macros can be generated from input/output examples [4]. [14] uses both symbolic and concrete example to synthesize distributed protocols.

## 8. DISCUSSION

In this paper, we provide an initial feasibility study into a tool that automatically generates network policy implementations from examples. Our initial results are promising. As immediate steps, we are exploring adapting these techniques beyond SDN protocols, particularly protocols that involve multiple switches/devices. In addition, our initial work opens up a number of possible directions for us to explore further. These include:

**Programming environments.** We plan to explore friendly programming environments for network operators to interact with NetEgg, which is able to detect and highlight inconsistency appearing in the input scenarios. Moreover, we plan to explore automatically generating scenarios for network operators, improving the confidence for the synthesized program. Third, we plan to explore automatic verification tools that can check synthesized programs against high level properties. This suggests a possible refinement-based approach, where an operator starts off with a high-level property they have in mind, and NetEgg iterate through scenarios given by the verification tool until the synthesized program is correct.

**Distributing the synthesized implementation.** Our current strawman implementation requires that all traffic is sent to the controller for processing, increasing the overhead due to controller involvement. We are in the process of automatically inferring flow table updates from synthesized actions in the controller policy table to avoid this problem.

## 9. REFERENCES

[1] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. Netkat: Semantic foundations for networks. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2014), ACM, pp. 113–126.

[2] Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., and Valadarsky, A. Vericon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 31.

[3] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: A network programming language. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 279–291.

[4] Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 317–330.

[5] Harel, D. Can programming be liberated, period? *Computer 41*, 1 (2008), 28–37.

[6] Harel, D., and Marelly, R. *Come, lets play: Scenario-based programming using LSCs and the Play-Engine*, vol. 1. Springer, 2003.

[7] Harel, D., Marron, A., and Weiss, G. Behavioral programming. *Communications of the ACM 55*, 7 (2012), 90–100.

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

[9] Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica., I. Declarative Networking. In *Communications of the ACM (CACM)* (2009).

[10] Monsanto, C., Foster, N., Harrison, R., and Walker, D. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices 47*, 1 (2012), 217–230.

[11] Monsanto, C., Reich, J., Foster, N., Rexford, J., Walker, D., et al. Composing software defined networks. In *NSDI* (2013), pp. 1–13.

[12] Moshref, M., Bhargava, A., Gupta, A., Yu, M., and Govindan, R. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'14)* (August 2014).

[13] Nayak, A. K., Reimers, A., Feamster, N., and Clark, R. Resonance: Dynamic access control for enterprise networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking* (New York, NY, USA, 2009), WREN '09, ACM, pp. 11–18.

[14] Udupa, A., Raghavan, A., Deshmukh, J. V., Mador-Haim, S., Martin, M. M., and Alur, R. Transit: specifying protocols with concolic snippets. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 287–296.

[15] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013), ACM, pp. 87–98.