# Tierless Programming and Reasoning for Software-Defined Networks

Tim Nelson Andrew D. Ferguson Michael J. G. Scheer Shriram Krishnamurthi
*Brown University*

## Abstract

We present Flowlog, a *tierless* language for programming SDN controllers. In contrast to languages with different abstractions for each program tier—the control-plane, data-plane, and controller state—Flowlog provides a unified abstraction for all three tiers. Flowlog is reminiscent of both SQL and rule-based languages such as Cisco IOS and JunOS; unlike these network configuration languages, Flowlog supports programming with mutable state. We intentionally limit Flowlog's expressivity to enable built-in verification and proactive compilation despite the integration of controller state. To compensate for its limited expressive power, Flowlog enables the reuse of external libraries through callouts.

Flowlog proactively compiles essentially all forwarding behavior to switch tables. For rules that maintain controller state or generate fresh packets, the compiler instructs switches to send the minimum amount of necessary traffic to the controller. Given that Flowlog programs can be stateful, this process is non-trivial. We have successfully used Flowlog to implement real network applications. We also compile Flowlog programs to Alloy, a popular verification tool. With this we have verified several properties, including program-correctness properties that are topology-independent, and have found bugs in our own programs.

## 1 Introduction

In a software-defined network (SDN), switches delegate their control-plane functionality to logically centralized, external controller applications. This split provides several advantages including a global view of network topology, and the use of general-purpose programming languages for implementing network polices. These general-purpose languages require an interface to the switch hardware, such as OpenFlow [20], which also provides a basic abstraction of the switch's flow tables.

To best use this interface, recent research has produced domain-specific languages like NetCore [21] that can be proactively compiled to flow tables. While this exclusive focus on flow tables simplifies compilation, it hurts expressivity. NetCore, for instance, can describe a forwarding policy, but lacks the ability to reference (let alone change) state on the controller.

Instead, the programmer must write a multi-tier program: a wrapper in a general-purpose language that maintains control-plane state and dynamically creates new, stateless data-plane policies to describe current forwarding goals. The data-plane policies must also specify which packets the switches should send to the controller, but—due to the multi-tier, multi-language nature of these programs—there is no structured connection between policies that describe packets the controller receives, and the arbitrary code in a callback function that consumes them. This gap can lead to bugs in how switches update the controller, resulting in incorrect controller state or network policies. Moreover, if packets are delivered to the controller needlessly, performance suffers.

To better support controller programming, we have created Flowlog, a *tierless* network programming language. As in web-programming, where a program contains multiple tiers such as client-side JavaScript, a server-side program, and a database, an SDN system also has multiple tiers: flow rules on switches, a controller program, and a data-store for controller state. By incorporating all of these tiers, a single, unified Flowlog program describes both control- and data-plane behavior.

Flowlog also provides built-in support for program verification. Because controller programs pose a single point of failure for the entire network, verification tools are invaluable to SDN developers. Prior SDN controller analysis work has often focused on the switch rules themselves, either statically [2, 12, 19, 26] or dynamically, as each update is sent to the switches [25]. However, most SDN analyses focus on trace properties: statements about the end-to-end behavior of packets in the network (e.g., a lack of routing loops). While these analyses are useful, they generally must be performed with respect to a given topology, which limits their flexibility, reusability, and scalability. In contrast, our reasoning focuses on properties independent of network topology.

We have limited Flowlog's expressive power to support both tierlessness and verification, while retaining enough expressivity to be useful for real-world programming. A limited language poses obvious problems for developers, both in expressing their needs and in reusing existing code. Flowlog therefore provides interfaces and abstractions for interacting with external programs. Programmers are free to invoke existing, full-featured libraries as needed, depending on their analysis goals. This is in contrast to most policy languages: in Flowlog, the restricted language itself forms the primary program, calling the external code rather than being called by it. This approach has been successful in SQL, where database queries are in the "limited" language and user-defined functions are in "full" languages. Our work explores such a strategy for network programming. Our contributions are:

1. We present the tierless Flowlog language (Section 3) and demonstrate its expressive power on real-world examples (Section 2). The language includes SQL-like relational state. It also provides abstractions for interaction with external code, via either asynchronous *events* or synchronous *remote tables*. Section 6 describes its implementation.

2. We show how, in spite of Flowlog's tierless merging of data- and control-plane behavior, programs can be proactively compiled to flow table rules (Section 4). The compilation process extends beyond mere packet forwarding; it also filters packets that may trigger state updates or cause event output and notifies the controller only as necessary.

3. We automatically compile Flowlog programs to the Alloy [10] verifier (Section 5). We focus on analyses that are independent of network topology, which are especially helpful when the topology is virtual, in flux, or unknown. We show that this process is aided by Flowlog's tierlessness as well as its limited expressiveness. We are able to verify properties in less than a second with minimal developer input. This verification support has helped us find surprising errors in our own Flowlog programs.

## 2 Flowlog by Example

We introduce Flowlog with illustrative examples. These demonstrate Flowlog's tierless nature, along with its expressive power, concision, and ability to support real-world development needs. We have also designed Flowlog to be amenable to both sound (i.e., no false positives) and complete (i.e., no bugs are missed) verification. To meet these goals, Flowlog bans loops and recursion, and has a logical semantics that we leverage for sound and, in many cases, complete verification (Section 5). Moreover, no recursion means that Flowlog programs always terminate on each incoming event.

**Stolen Laptop Detector** Let us write an application to help campus police track down stolen laptops. It must accept signals from campus police that report a laptop stolen or recovered, and if a stolen laptop is seen sending packets, the program must alert the police, saying which switch the laptop is connected to and when the packet was seen. (For brevity, we do not demonstrate rate limiting of alerts or restriction of alerts to edge-router traffic. Both of these tasks can be accomplished in Flowlog.)

Without a tierless programming language, expressing this program would require many pieces, possibly using multiple languages: a database or data structures to manage controller state; a remote-procedure call (RPC) library, or similar solution, for handling events; and policy-generation code that produces fresh rules on the switches that forward traffic and check for stolen laptops on the network. In Flowlog, all of these components share the same abstraction. Suppose we have a table `stolen` that tracks the MAC addresses of all currently stolen laptops. Then, the heart of the program is just the following rules:

```
1  ON stolen_report(sto):
2    INSERT (sto.mac) INTO stolen;
3  ON stolen_cancel(rec):
4    DELETE (rec.mac) FROM stolen;
5  ON packet_in(p):
6    DO notify_police(sto) WHERE
7      sto.mac = pkt.dlSrc AND
8      sto.time = time AND
9      sto.swid = pkt.locSw AND
10     stolen(pkt.dlSrc) AND
11     get_time(time);
12   DO forward(new) WHERE
13     new.locPt != p.locPt;
```

The program describes several kinds behavior that appear disparate. It: (a) adds addresses to a table when laptops are reported stolen (lines 1–2); (b) removes addresses when laptops are recovered (lines 3–4); (c) notifies police when a packet appears from a stolen laptop (lines 5–11); and (d) floods packets (lines 12–13); this trivial example of forwarding introduces syntax which we will use later. With Flowlog's tierless abstraction, we express all of this in four concise rules.

Every program has similar rules that describe how to handle each packet; these rules are written in a syntax reminiscent of SQL, and the semantics is correspondingly relational. In addition, most programs will have state that must be updated in reaction to packets and other stimuli.

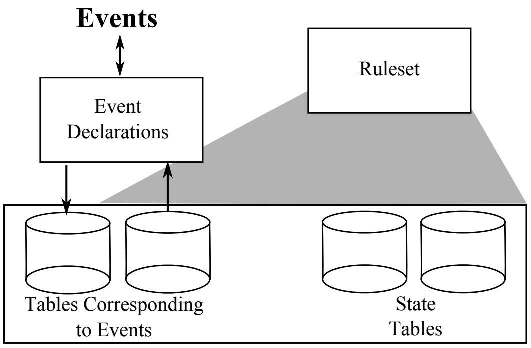

Figure 1: Flowlog system diagram. Events are represented by tuples in event tables. Rules interact with the entire database.

These stimuli are external events, whose implementations can be arbitrary, and whose interfaces are explicitly declared or built in. We now describe the program declarations that support these rules. (Figure 1 shows the different components of Flowlog.)

We have already seen `stolen`, the internal controller state. We expose the current time with an external table `get_time` (which always contains just one entry):

```
TABLE stolen(switchid);
REMOTE TABLE get_time(int);
```

External tables are managed by arbitrary external programs; in our examples, we used OCaml programs running on the controller machine. Each table declares its type, which is used for error-checking and optimization.

Next, we define the shape of incoming and outgoing events. We must handle two kinds of notifications from the police, and send them one:

```
EVENT stolen_report {mac: macaddr};
EVENT stolen_cancel {mac: macaddr};
EVENT stolen_found {mac: macaddr, swid:
    switchid, time: int};
```

Flowlog processes incoming events one at a time. Incoming events are placed in an identically named table, causing dependent rules to be re-evaluated. Outgoing events are also represented as tables, and when the ruleset adds a tuple to such a table, Flowlog sends an event with the tuple's content. These tables are therefore similar to named pipes in Unix. We require one such named pipe to send `stolen_found` events to the police server:

```
OUTGOING notify_police(stolen_found)
  THEN SEND TO 127.0.0.1:5050;
```

If a Flowlog program inserts tuples into the `notify_police` table, these tuples will be transformed into `stolen_found` events and sent to a process listening on `127.0.0.1:5050`. Incoming `stolen_report` and `stolen_cancel` events are automatically inserted into their identically named tables. For instance, the `stolen_report` table will contain arriving `stolen_report` events, and the `packet_in` table will hold incoming packets. Notice we did not declare an outgoing `forward` table. This is because Flowlog automatically creates outgoing tables for common packet-handling behavior, as detailed in Table 1.

The remote table `get_time` is populated by querying `127.0.0.1:9091`, and should be refreshed every second:

```
1 REMOTE TABLE get_time
2   FROM time AT 127.0.0.1:9091
3   TIMEOUT 1 seconds;
```

The `TIMEOUT` field (line 3) is vital for performance and correct proactive compilation. A numeric timeout gives a window during which the results can be cached. Flowlog also provides a `NEVER` keyword, meaning the external callout has no side-effects, and thus its results can be cached indefinitely. A default, empty timeout requires updating the remote table every time the program is evaluated.

The reader may wonder whether this program can be compiled to stateless rules and installed in switches. After all, it contains a rule only the controller can handle, since it involves notifying campus police. But this rule only fires when `stolen(p.dlSrc)` is true, so Flowlog's proactive compiler instructs switches to send packets to the controller only if their source-MAC field has been registered as stolen. Furthermore, Flowlog automatically updates the switches every time a new theft is reported; no code to that effect is needed.

**Network Information Base** Next, we show how Flowlog can be used to compute a network information base, or NIB [15]. We begin with topology discovery, using Flowlog's ability to process timer notifications and emit new packets to run an LLDP-like protocol. First, we react to switch registration to obtain identifiers for every port (omitting table declarations):

```
ON switch_port_in(swpt):
  INSERT (swpt.sw, swpt.pt)
    INTO switch_has_port;
```

Thus, the `switch_has_port` table will hold every switch-port pair that registers. Next, we set up a 10-second event loop (we exclude the timer declaration):

```
ON startup(empty_event):
  DO start_timer(10, "tNIB");
ON timer_expired(timer)
  WHERE timer.id = "tNIB":
  DO start_timer(10, "tNIB");
```

The first rule uses Flowlog's built-in `startup` event to start the loop, and the second rule continues it. The constraint `timer.id = "tNIB"` accounts for situations where multiple timers may be in use. The same timer also causes known switches to issue probe packets from each port:

| INCOMING Table | Corresponding EVENT (with fields) | Description |
|---|---|---|
| packet_in | packet{locSw, locPt, dlSrc, dlDst, dlTyp, nwSrc, nwDst, nwProtocol} | packet arrival |
| switch_port | switch_port{sw, pt} | switch registration |
| switch_down | switch_down{sw} | switch down |
| E | E | Any incoming event E |

| OUTGOING Table | Corresponding EVENT | Description |
|---|---|---|
| emit | packet | Emit a new packet |
| forward | packet | Forward with modifications (triggered by packets only) |

Table 1: Built-in **INCOMING** and **OUTGOING** tables. The locSw and locPt fields denote the packet's (switch and port) location.

```
1 ON timer_expired(timer)
2   WHERE timer.id = "tNIB":
3   DO emit(new) WHERE
4      switch_has_port(new.locSw,new.locPt)
5     AND new.dlTyp = 0x1001
6     AND new.dlSrc = new.locSw
7     AND new.dlDst = new.locPt;
```

We use dlTyp = 0x1001 (line 5) to mark probe packets. The current switch and port IDs are smuggled in the MAC address fields of the probe (lines 6-7). (We omit the rule that initiates the same probe emission process on switch registration.)

We obtain knowledge of the switch topology from probe reception:

```
ON packet_in(p) WHERE p.dlTyp = 0x1001:
  INSERT (p.dlSrc, p.dlDst,
          p.locSw, p.locPt) INTO ucST;
```

The table name ucST denotes *under construction* switch topology; at any point, it contains a topology based on the probes seen so far this cycle. We empty ucST on every cycle, and maintain a switchTopology table that stores the value of the last complete ucST before it is deleted. Flowlog programs routinely use this strategy of building up a helper table over an execution cycle, separating in-progress results from the last complete result set:

```
ON timer_expired(timer)
WHERE timer.id = "tNIB":
  DELETE (sw1,pt1,sw2,pt2) FROM ucST WHERE
     ucST(sw1, pt1, sw2, pt2);
  DELETE (sw1,pt1,sw2,pt2)
    FROM switchTopology WHERE
     switchTopology(sw1, pt1, sw2, pt2);
  INSERT (sw1,pt1,sw2,pt2)
    INTO switchTopology WHERE
     ucST(sw1, pt1, sw2, pt2);
```

Though Flowlog does not allow recursion (Section 3), we can use a similar approach to compute reachability on the network. This program computes a fresh reachability table (ucTC, for under-construction transitive closure) as each probe arrives:

```
1  ON packet_in(p) WHERE p.dlTyp = 0x1001
2    AND dstSw = p.locSw AND srcSw = p.dlSrc:
3    INSERT (srcSw, dstSw) INTO ucTC;
4    INSERT (sw, dstSw) INTO ucTC
5      WHERE ucTC(sw, srcSw);
6    INSERT (srcSw, sw) INTO ucTC
7      WHERE ucTC(dstSw, sw);
8    INSERT (sw1, sw2) INTO ucTC
9      WHERE ucTC(sw1, srcSw)
10     AND   ucTC(dstSw, sw2);
```

The program works as follows: for every probe packet received, it concludes that its source switch and its arrival switch are connected (line 3). It also extends existing reachability in both directions (lines 4–7). Finally, it must account for packets connecting two cliques of reachability (lines 8–10).

It is instructive to compare this algorithm to the standard two-rule Datalog program for transitive-closure [1, p. 274]. The extra rules arise because we are *not* computing transitive-closure in the usual sense. Here, we do not have the luxury of assuming that we possess the entire connection table in advance; we must compute reachability on-the-fly as new information arrives. This difference leads to added complexity. In fact, an initial version of this program lacked the final rule, and so failed to faithfully compute reachability in some cases; we found this bug using our verification tool (Section 5).

Once we have network reachability, we can compute a spanning tree for the network:

```
ON packet_in(p) WHERE p.dlTyp = 0x1001
  AND dstSw = p.locSw AND dstPt = p.locPt
  AND srcSw = p.dlSrc AND srcPt = p.dlDst:
  INSERT (srcSw, srcPt) INTO ucTree
    WHERE NOT ucTC(srcSw, dstSw)
    AND   NOT ucTC(dstSw, srcSw);
  INSERT (dstSw, dstPt) INTO ucTree
    WHERE NOT ucTC(srcSw, dstSw)
    AND   NOT ucTC(dstSw, srcSw);
```

Again, we see unsurprising parallels to distributed protocols. Ordinary spanning tree algorithms have the luxury of working with the entire graph at once, and thus are often able to build a connected tree at every step. We do not have that luxury here: probes may arrive in any order, and we must build a forest of trees that, should links go down, may not even be connected. We also must add a pair of rules, one for each direction of the branch.

```
block ::= ON <id> ( <id> ) [WHERE rformula]
          : rules
rules ::= rule | rule rules
rule  ::= do_act | ins_act | del_act
do_act  ::= DO <id> ( termlist )
            [WHERE rformula] ;
ins_act ::= INSERT ( termlist )
            INTO <id> [WHERE rformula] ;
del_act ::= DELETE ( termlist )
            FROM <id> [WHERE rformula] ;
term ::= <num> | <string> | <id> | <id>.<id> | ANY
termlist ::= term | term , termlist
rformula ::= <id> ( termlist ) | term = term |
             NOT rformula | rformula AND rformula |
             rformula OR rformula | ( rformula )
```

Figure 2: Syntax of Flowlog rulesets. A program is a succession of **ON** blocks. Optional arguments are in square brackets. Capitalized tokens and punctuation are reserved constants.

Of course, this spanning tree is not necessarily the best possible one; we only compute the first such tree to be exposed by probe packets. Better tree-generation algorithms can be written or accessed via external code. Numerous other data, such as the location of connected hosts, can also be gathered, but are omitted for space.

Given a spanning tree for the network—whether it is computed in Flowlog or obtained from external code—we can construct a "smart" learning switch application in Flowlog that does not suffer from the usual issues with cyclic topologies.

**Other Examples** We have implemented additional applications in Flowlog, which are available in our repository.[1] These examples include an ARP proxy, a stateful firewall, and an application (which we use in-house) to facilitate access to Apple TV devices across subnets.

## 3 The Flowlog Language

As seen in Section 2, every Flowlog program contains a declarative *ruleset* that governs controller behavior and a set of *declarations* for the program's state tables and incoming/outgoing interface. Figure 2 gives the concrete syntax of Flowlog rulesets.

**Declarations** A program declares **EVENT**s, state **TABLE**s, and interfaces for **INCOMING** and **OUTGOING** tables. Most **INCOMING** and some **OUTGOING** declarations are made automatically when events are declared. Declaring a table as **REMOTE** informs Flowlog that the table represents a callout to external code, and that the ruleset will not maintain that table's state. Every event declaration is equipped with a set of field names for that event type. Every internal table and interface table is equipped with a type, given as a vector of type names (e.g., "switchid")—one for each column in the table. **REMOTE TABLE** and **OUTGOING** declarations must also be provided with additional information, as we saw in Section 2.

**Rulesets** A ruleset contains a set of **ON** blocks of rules. While we allow multiple rules within the same **ON** block for conciseness, without loss of generality we will pretend that every rule has its own **ON** block. Each rule indicates an action to be taken when the **ON**-specified trigger is seen: either to **INSERT** or **DELETE** a tuple from the controller state, or to **DO** an action such as forwarding a packet. Finally, rules and triggers have an optional **WHERE** clause, which adds additional constraints; these are always an expression involving only the tables declared in **TABLE**s, never those declared as **INCOMING** or **OUTGOING**. These rules determine a function that maps controller state and incoming events to a new state and set of outgoing events.

Each rule defines a logical implication stating that, should its body be satisfied, its action should be as well. Figure 3 shows how we arrive at this *rule clause* for each rule. If the rule's action is **DO**, then the resulting clause inserts tuples into the outgoing table directly. If the rule's action modifies an $n$-ary table $R$ via the **INSERT** or **DELETE** keywords, the clause uses $n$-ary helper tables $R_{add}$ or $R_{del}$, which hold the tuples to be added to and removed from the controller state after an event is processed.

If $S$ is a controller state, let $S^{\uparrow}$ represent the unique least expansion of $S$ that satisfies all rule clauses. That is, while $S$ contains a table for each **TABLE**, $S^{\uparrow}$ also contains ephemeral $R_{add}$ and $R_{del}$ tables for each **TABLE** as well as tables for each **OUTGOING** declaration. These **OUTGOING** tables are consumed by Flowlog and dictate which outgoing events it should send. The ephemeral tables for each state table $R$ dictate its value in the next state as follows:

$$R_{next} = (R^{S} \setminus R_{del}^{S^{\uparrow}}) \cup R_{add}^{S^{\uparrow}}$$

In other words, **INSERT** overrides **DELETE** in Flowlog—the next state's $R$ contains the pre-state's $R$, minus $R_{del}$, plus $R_{add}$.

## 4 Proactive Compilation for Flowlog

While Flowlog programs receive many kinds of input—both packets and external notifications—packets remain the most common and time-sensitive stimuli. No software-defined network can scale to the level required by large networks if it sends every arriving packet to the controller for instructions. This complicates the implementation of a tierless SDN language; rather than simply have all packets be processed by the controller in accordance with the program, the Flowlog runtime is forced to solve three related, but distinct, challenges:

[1] http://cs.brown.edu/research/plt/dl/flowlog/

| | |
|---|---|
| **ON** `IN(in)` **DO** `OUT(o1, ..., ok)` **WHERE** `rf` | $\forall in, o_1, ...o_k\ \exists e_1, ..., e_k\ OUT(o_1, ...o_k) \leftarrow IN(in) \wedge T_{fmla}(\texttt{rf})$ |
| **ON** `IN(in)` **INSERT** `(o1, ..., ok)` **INTO** `R` **WHERE** `rf` | $\forall in, o_1, ...o_k\ \exists e_1, ..., e_k\ R_{add}(o_1, ...o_k) \leftarrow IN(in) \wedge T_{fmla}(\texttt{rf})$ |
| **ON** `IN(in)` **DELETE** `(o1, ..., ok)` **FROM** `R` **WHERE** `rf` | $\forall in, o_1, ...o_k\ \exists e_1, ..., e_k\ R_{del}(o_1, ...o_k) \leftarrow IN(in) \wedge T_{fmla}(\texttt{rf})$ |

(All rules existentially quantify variable occurrences that are free and not in $\{in, out_1, ..., out_k\}$; hence the $e_i$s.)

| | | |
|---|---|---|
| $T_{fmla}$(**NOT** `f`) | $=$ | $\neg T_{fmla}(\texttt{f})$ |
| $T_{fmla}$(`f1` **AND** `f2`) | $=$ | $T_{fmla}(\texttt{f1}) \wedge T_{fmla}(\texttt{f2})$ |
| $T_{fmla}$(`t1 = t2`) | $=$ | $T_{term}(\texttt{t1}) = T_{term}(\texttt{t2})$ |
| $T_{fmla}$(`P(t1, ..., tk)`) | $=$ | $\exists x_1, ..., x_k\ P(T_{term}(\texttt{t1}), ..., T_{term}(\texttt{tk}))$ <br> $x_1$, ..., $x_k$ are the fresh variables introduced by `ANY`s in the original formula. |

| | | |
|---|---|---|
| $T_{term}$(`c`) | $=$ | $c$ |
| $T_{term}$(`x`) | $=$ | $x$ |
| $T_{term}$(`x.fld`) | $=$ | $fld(x)$ |
| $T_{term}$(`ANY`) | $=$ | $x_{fresh}$ |

Figure 3: Rule-formula to formula ($T_{fmla}$) and Rule-term to term ($T_{term}$) transformation functions. Without loss of generality, we provide every rule with its own **ON** trigger and assume that disjunction in rule bodies has been removed, resulting in multiple rules. $x_{fresh}$ denotes a fresh variable.

1. It must compile a program's *forwarding behavior* to equivalent OpenFlow [20] rules whenever possible, including references to both local and remote state;

2. it must discern which packets can trigger non-forwarding behavior, such as emission of an event or a state change, and produce OpenFlow rules that send those packets—and only those packets—to the controller; and,

3. if a rule uses features that are not supported by switch tables (we detail these cases later), the compiler determines the class of packets that must be sent to the controller for correct handling. This situation applies to both forwarding and non-forwarding rules.

As Section 3 demonstrated, Flowlog rules can involve equality between packet fields, negation, database state, and numerous other features not supported by OpenFlow. Moreover, rules can contain existentially quantified variables that, at first glance, require searching and backtracking in the state to properly handle. Simply put, Flowlog rules are strictly more powerful than OpenFlow 1.0 flow rules. It is therefore reasonable to wonder: can a non-trivial amount of Flowlog really be compiled faithfully? This section will show it can be. Figure 4 shows the compilation dataflow.

Our proactive ruleset compiler has three stages:

1. First (Section 4.1), it simplifies each rule and identifies the compilable forwarding rules. Both non-forwarding rules (state updates, emission of fresh packets, etc.) and non-compilable forwarding rules require switches to send packets to the controller; fortunately, these notifications can be extensively filtered based on the program's structure.

2. Second (Section 4.2), it partially evaluates the ruleset at the current controller state, producing a new ruleset that has no references to state tables. Since the original ruleset defines a function that accepts a state and an incoming tuple and returns a set of outgoing tuples, the resulting ruleset depends only on the incoming tuple.

3. Finally (Section 4.3), it compiles the new ruleset automatically to flow table rules in two steps. First, it converts to NetCore [21], a stateless forwarding policy language for OpenFlow. Second, it applies NetCore's compiler to produce flow table rules.

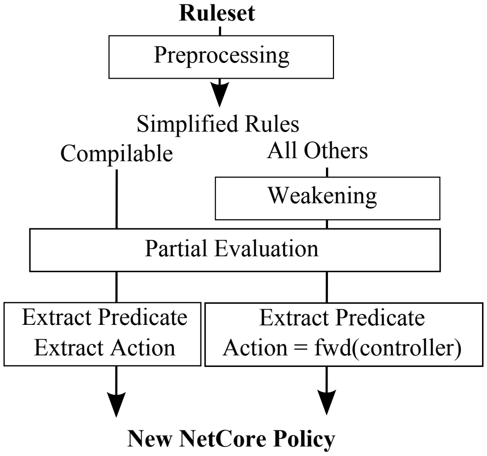

Figure 4: Flowlog's compilation process. Rules are first pre-processed before being checked for compilability, then (if un-compilable) weakened before being partially evaluated in the current state. After partial evaluation, the rule is re-written as a stateless NetCore policy.

**Example: Forwarding** For intuition into the compilation process, consider the following example rule.

```
ON packet_in(p):
  DO forward(new) WHERE
    learned(p.locSw,new.locPt,p.dlDst);
```

This rule says: "Forward `p` on a port corresponding to its location and destination, provided the controller has learned that correspondence". It compiles to the following rule clause:

$$\begin{aligned}\forall p, new\,.\,forward(new) &\Longleftarrow \\ &learned(locSw(p), locPt(new), dlDst(p)) \\ &\wedge packet\_in(p)\end{aligned}$$

Suppose the current state contains $learned = \{\langle 1,2,3\rangle, \langle 1,3,2\rangle, \langle 1,4,3\rangle\}$. Then the *learned* expression in the above clause is equivalent to:

$$\begin{aligned}&((locSw(p) = 1 \wedge locPt(new) = 2 \wedge dlDst(p) = 3) \vee \\ &\;(locSw(p) = 1 \wedge locPt(new) = 4 \wedge dlDst(p) = 3) \vee \\ &\;(locSw(p) = 1 \wedge locPt(new) = 3 \wedge dlDst(p) = 2))\end{aligned}$$

Re-written as a NetCore policy, this is just:

```
(filter (locSw = 1 and dlDst = 3);
  fwd(2) | fwd(4)) +
(filter (locSw = 1 and dlDst = 2); fwd(3))
```

This policy can remain in place in flow tables until such time as the `learned` table changes.

**Example: State Change** Flowlog provides a "see-every-packet" abstraction. For instance, the following program appears to execute entirely on the controller:

```
ON packet_in(p):
  INSERT (p.locSw, p.locPt, p.dlSrc)
    INTO learned WHERE
    NOT learned(p.locSw, p.locPt,
                p.dlSrc);
```

With the exception of Maple [32], existing languages with this abstraction require the programmer to carefully maintain separate logic for packet forwarding and controller notifications. In contrast, the Flowlog runtime handles controller notification automatically; the only packets the controller needs are those that provably alter the controller state. Flowlog's compiler automatically builds and deploys a NetCore policy that applies to all such packets. For example, suppose the current state is $learned = \{\langle 1,2,3\rangle, \langle 1,3,2\rangle\}$. The following NetCore policy ensures the controller sees the packets it needs to, and no more:

```
if not ((locSw = 1 and locPt = 2 and dlSrc = 3) or
        (locSw = 1 and locPt = 3 and dlSrc = 2))
   then fwd(controller)
```

## 4.1 Simplification and Compilability

Before compiling a ruleset, Flowlog removes unnecessary variables. For example, if `p` is the incoming packet, the condition `learned(p.locSw, y, p.dlSrc)` **and** `x=p.locPt` **and** `y=x` would be rewritten as `learned(p.locSw, p.locPt, p.dlSrc)`. This process eliminates hidden dependencies, simplifying compilation.

Each rule is then subjected to a compilability check. Table 2 lists the conditions under which a rule cannot be compiled. If a rule fails one or more tests, it either triggers an outright error or must be handled by the controller. For instance, a rule that compares the incoming packet's layer-2 source and destination fields is easily expressed in Flowlog as `p.dlSrc = p.dlDst` and can be checked reactively by the controller, but is not supported by OpenFlow 1.0 forwarding tables.

Finally, to reduce the number of packets that must be sent to the controller, Flowlog *weakens* the **WHERE** condition of each uncompilable rule to obtain a compilable overapproximation. A rule clause is a conjunction of literals (i.e., positive or negative assertions about state or equality), and weakening removes objectionable literals (Table 2) from the clause. Removing parts of a conjunction yields a new formula that is implied by the original, so it is a sound overapproximation.

## 4.2 Partial Evaluation

Partial evaluation removes references to state tables within each rule, replacing them with simple equalities involving only constants and variables. Figure 5 defines the partial evaluation function ($T_{pe}$), and other transformation functions used below. Once partial evaluation is complete, the compiler distributes out any disjunctions introduced by partially evaluating *positive* literals, resulting in a new set of clause formulas. This is done so new equalities constraining the outgoing packet, if any, are immediately available at the top level of the conjunction. (Disjunctions coming from negative literals are left in place; this is safe since outgoing packet fields that occur in negated table references are forbidden.) Any clauses that were partially evaluated to a contradiction are removed.

## 4.3 Extracting NetCore Policies

The policies that the proactive compiler produces have two parts: a stateless filtering condition on packets (the predicate) and the set of actions to apply when the predicate matches. NetCore predicates support the essential Boolean operators—`or`, `and`, `not`—as well as *filters* over header fields and switch identifiers.

An equivalent NetCore policy for each clause is created using the $T_{pred}$ (extract predicate) and $T_{act}$ (extract action) functions defined in Figure 5. Predicate extraction only involves the incoming packet; other literals map to the trivially true predicate `all`. Non-forwarding rule clauses are always assigned the send-to-controller action. For each forwarding rule clause, the compiler extracts an action assertion such as "forward on port 3". Since

| Condition | Example | Explanation |
|---|---|---|
| (a) Forbidden new-packet field assignment | `new.nwProto = 5` | Not allowed in OpenFlow 1.0 |
| (b) Different fields in old-to-new assignment | `new.dlSrc = old.dlDst` | Not allowed in OpenFlow 1.0 |
| (c) Negatively constrained new-packet field | `new.dlSrc != 5` or **not** `R(new.dlSrc)` | Forbid packet avalanche |
| (d) Reflection on incoming packet in equality | `old.dlSrc = old.dlDst` | Not allowed in OpenFlow 1.0 |
| (e) Non-assignment condition of new packet | `new.locPt = new.dlSrc` | For compilation speed |
| (f) Multi-way join on state tables | `R(3,X)` **and** `R(X,4)` | For compilation speed |

Table 2: Situations that cause a rule to be weakened and dealt with by the controller. In a forwarding rule, (a–b) are forbidden at compile time. The "flood" condition, `new.locPt != p.locPt`, is the sole exception to (c); other forms would cause a plethora of outgoing packets. (d–f) are allowed at compile time, but force weakening of forwarding rules. By eliminating complex join conditions from compilation (e–f), we avoid the necessity of solving a search problem to compile rules; after preprocessing, existential variables appear in compiled rules only as placeholders for "don't-care" positions in rule formulas.

Partial Evaluation: *States* × *Formulas* → *Formulas*

$$\begin{array}{rcl}
T_{pe}(S,R(t_1,...,t_n)) & = & \bigvee_{\langle c_1,...,c_n\rangle \in R^S} (t_1 = c_1 \wedge ... \wedge t_n = c_n) \\
T_{pe}(S,t_1 = t_2) & = & t_1 = t_2 \\
T_{pe}(S,\neg\alpha) & = & \neg T_{pe}(S,\alpha) \\
T_{pe}(S,\beta \vee \gamma) & = & T_{pe}(S,\beta) \vee T_{pe}(S,\gamma) \\
T_{pe}(S,\beta \wedge \gamma) & = & T_{pe}(S,\beta) \wedge T_{pe}(S,\gamma)
\end{array}$$

Predicate Extraction: *Rule Clauses* → *Pred*

$$\begin{array}{rcl}
T_{pred}(oldpkt.fld = c) & = & \texttt{fld = c} \\
T_{pred}(t = c) & = & \texttt{all} \\
T_{pred}(\neg\alpha) & = & \texttt{not } T_{pred}(\alpha) \\
T_{pred}(\beta \wedge \gamma) & = & T_{pred}(\beta) \texttt{ and } T_{pred}(\gamma)
\end{array}$$

Action Extraction: *Rule Clauses* → $2^{Action}$

$$\begin{array}{rcl}
T_{act}(newpkt.locPt = c) & = & \{\texttt{fwd(c)}\} \\
T_{act}(newpkt.fld = c) & = & \{\texttt{set(fld, c)}\} \\
T_{act}(\neg\alpha) & = & \emptyset \\
T_{act}(\beta \wedge \gamma) & = & T_{act}(\beta) \cup T_{act}(\gamma)
\end{array}$$

Figure 5: Transformation functions used during compilation.

contradictions were removed (Section 4.2), only one such assertion is made per clause. Clauses containing inequalities of the form `new.locPt != old.locPt` are added to the predicate in a final pass after the forwarding action is extracted. Once a predicate and action has been obtained for each clause, the compiler assembles the final policy by generating a sub-policy for each action that filters on the disjunction of all matching predicates, and then taking the union of those sub-policies.

## 5 Verification

To verify Flowlog programs, we use the Alloy Analyzer [10]. Alloy has a first-order relational language, which makes it a good match for Flowlog's first-order relational semantics. In addition, Alloy is automated and generates counterexamples when properties fail to verify.

We have created a compiler from Flowlog rulesets to Alloy specifications. The conversion is fully automated, although users must provide types (e.g., IP address) for constants used in the original program; this type information is used for optimization. By default, the compiler abstracts out the caching process and treats **REMOTE TABLE**s as constant tables. If analysis goals involve remote state, axioms about the behavior of remote code (e.g., "the routing library always gives a viable path") can be added manually.

Because of tierlessness, Alloy models created from Flowlog programs need not consider the eccentricities of the OpenFlow protocol or individual switch rules, and so reasoning benefits from the illusion that all packets are processed by the controller. This simplifies the resulting Alloy models (and improves analyzer performance), and also makes it easier for users to express properties across tiers. Ordinarily, for example, checking dependencies between forwarding behavior and state change would involve expressing the desired behavior for both packets that reach the controller and packets handled by switches; when reasoning about Flowlog, this split is unnecessary.

**Inductive Properties** An important class of program properties, which we call *inductive*, take the form: "If *P* holds of the controller state, then no matter what packet arrives, *P* will continue to hold in the next state." This property serves to prove that *P* always holds in any reachable state, so long as it holds of the starting state. Many desirable goals can be expressed in this way, and they are often independent of network topology.

To illustrate the power of this class of properties, consider our NIB example (Section 2). A piece of the NIB program gradually computes the transitive closure of the network topology. But does it really compute transitive closure faithfully? As probe packets arrive, the `ucTC` table needs to contain the transitive closure of the graph defined by all the links seen *so far*. Figure 6 shows how to encode this property in Alloy.

Running this analysis on an older version of the NIB program revealed a missing rule: we had failed to account for the case in which two mutually unreachable sub-networks become connected by an incom-

```
all st: State, st2: State, ev: EVpacket |
  transition[st, ev, st2] and
  ev.dltyp = C_0x1001 and
  (st.uctc = ^(st.uctc)) implies
    st2.uctc =
      ^(st.uctc + (ev.dlsrc -> ev.locsw))
```

Figure 6: Example Alloy property: "For all states (`st`) with `ucTC` transitively closed, the program only transitions to states (`st2`) with a transitively closed extension of `ucTC` by the arriving probe packet (`ev`)'s src/dst". Recall that the source switch ID is in the packet's `dlSrc` field. The `^` operator denotes transitive closure in Alloy. We chose `C_` to prefix constant identifiers.

| **Property** | **Time(ms)** | **B** |
|---|---|---|
| *NIB* | | |
| Reachability computed correctly (4sw) | 40 | |
| Reachability (with bug) | 35 | |
| Spanning tree never has cycles | 58 | ✓ |
| Timer correctly updates persistent tables | 23 | ✓ |
| Correctly capture host location changes | 65 | ✓ |
| *Stolen Laptop* | | |
| Only police can un-flag a laptop | 4 | ✓ |
| *Learning Switch* | | |
| $\leq$ 1 port learned per host per switch | 14 | ✓ |
| Only switch failure can restart flooding | 14 | ✓ |

Table 3: Example properties with time to verify or find a counterexample. The **B** column shows whether sufficient bounds could be established, as described in Section 5. Alloy 4.2/3.1 GHz Core i5/8 GB RAM.

ing probe packet. That is, we were missing this rule:

```
ON packet_in(p) WHERE p.dlTyp = 0x1001
  AND dstSw = p.locSw AND srcSw = p.dlSrc:
  INSERT (sw1, sw2) INTO ucTC
    WHERE ucTC(sw1, srcSw)
    AND   ucTC(dstSw, sw2);
```

This rule is necessary due to the subtle nature of computing reachability from each probe in succession, rather than having access to the entire table and applying recursion. Alloy was able to demonstrate this bug on a network of only four switches.

Using this method, we have successfully verified properties of (or found bugs in) multiple Flowlog programs including the NIB, the stolen laptop alert program, and a MAC learning switch. Table 3 lists several along with the time they took to verify: well under a second.

**Completeness** Alloy performs *bounded* verification: it requires a size bound for each type, which it uses to limit the search for a counterexample. For instance, for our NIB verification we might instruct Alloy to search up to four switches (irrespective of the wiring between them), three distinct MAC addresses, etc. Because of its bounded nature, Alloy is not in general *complete*: it will fail to find counterexamples larger than the given bound. Since individual properties are a result of purpose-specific program goals, properties and their associated bounds must be entered manually.

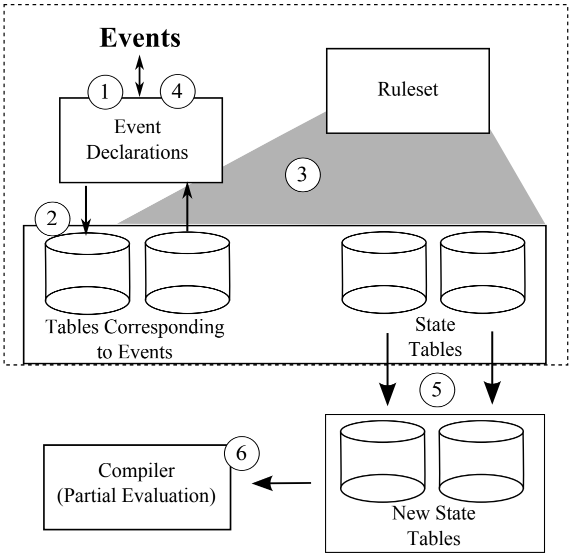

Figure 7: Flowlog's workflow for responding to events. The boxed portion of the diagram appeared as Figure 1.

Fortunately, for many common types of analyses, we can exploit prior work [23] to compute (small) size bounds that are sufficient to find a counterexample, should one exist. All but one of the properties we verified is amenable to this technique; the exception is reachability, because the technique does not support transitive-closure (Table 3). Yet broad experience with Alloy indicates that many bugs can be found with fairly small bounds (e.g., four switches for our transitive-closure bug). Moreover, bounds on other objects (e.g., non-switches) can still be produced for all the inductive properties that we tested.

## 6 Implementation and Performance

The current Flowlog implementation uses OpenFlow 1.0 [20] and Frenetic [5] for packet-handling, Thrift RPC (`thrift.apache.org`) for orchestrating events and remote state, and the XSB [28] Prolog engine for evaluation. Flowlog is implemented in OCaml.

Figure 7 sketches the controller's workflow. When an event arrives (1) the controller converts it into a tuple and places it in the appropriate input table via XSB's `assert` command (2). Then, for each outgoing and state-modification table, the controller queries XSB to obtain a set of outgoing tuples (3) which are converted to events (4). Then each state-modification tuple is `asserted` or retracted to result in the new state (5). Finally, proactive compilation (6) is performed on the new state, producing a NetCore policy.

**External Events** Flowlog evaluation is triggered by a general set of events; the runtime must watch for more than just packet arrivals. For instance, the runtime sends an event to the controller whenever a switch registers or goes down, and as seen in Section 2, external applications may also interact with Flowlog through events. Our hypothetical campus police-officer informs Flowlog to register a stolen laptop by using a small application (around 100 lines, most of which is boilerplate) that uses Thrift to send an asynchronous message to Flowlog.

**Remote Tables and Caching** Flowlog rules reference a database of relational facts. As seen in Section 3, tables can be declared either as local or remote. A local table is managed internally by the controller (via assert and retract statements to XSB), while a remote table is merely an abstraction over callouts to external code. Like events, these callouts use Thrift RPC to interact with external code. Unlike events, callouts are synchronous. Callouts have the form of an ordinary state-referencing formula, $R(t_1,...,t_n)$, but each $t_i$ must be either a constant value or a variable. After Flowlog queries the correct external application, the reply contains a set of tuples of constants—one constant for each variable in the query.

Although the rules see no distinction between local and remote tables, in practice it would be impractical or impossible to obtain *entire* remote tables (such as an infinitely large table that represents the addition of numbers). Therefore, Flowlog obtains tuples from external code only when they are needed by a rule. A naïve implementation could simply obtain remote tuples every time they were required; however, that would mean forwarding rules could not be compiled if they referred to external code. Instead, we cache remote tuples for the declared time-to-live. Since we maintain the remote cache in XSB, when it comes time to react to an event, the controller handles remote and local state in exactly the same way: via XSB.

When an event arrives, we invalidate cached tuples whose time-to-live has expired. If the expired tuples were used in a compiled policy, we force an update of the cache and provide switches with the new policy. External programs are expected to not update their internal state or otherwise provide inconsistent results within the TIMEOUT values of their Flowlog definitions.

**Handling Overlapping Rules** In some SDN applications, switches will forward a packet on the data plane and also send it to the controller. If we kept the pre-compilation ruleset unmodified on the controller, this could lead to packet duplication due to the compiled forwarding rules: packets would be forwarded once by the switch tables, and forwarded again by the same rule's action on the controller.

Since the compilability of a rule is independent of controller state, we can determine which rules these are at program startup. We then leave these rules out of what we pass to XSB, and disallow the controller from taking duplicate actions. However, since there is a delay between the controller's state change and corresponding rules being installed on the switches, this is not a perfect solution: packets may be in-transit during deployment of the new policy. This issue is not unique to Flowlog, and has been noted by others [26].

**Performance and Scalability** Flowlog is proactively compiled to switch rules whenever possible. From the traffic forwarding perspective, therefore, Flowlog is largely dependent on what is supported in switch hardware. We have confirmed experimentally that, as one would hope, the controller receives no unnecessary packets. For instance, a Flowlog learning switch never sends the controller a packet once that packet's source location is learned, and eventually the controller is not burdened at all. Packet-counts were confirmed by both `ifconfig` counters and packet events seen by Flowlog. We used `ping` packets and a Mininet [16]-hosted virtual network to simulate network traffic. We tested on tree topologies with 3 and 7 switches as well as a cyclic 3-switch topology to test controller robustness, and have begun testing on larger topologies as well.

Because forwarding in Flowlog is as fast as hardware allows, one scalability question remains: since Flowlog compiles the controller state into NetCore policies, how does this scale as the controller's database grows? The size of the NetCore policy we produce depends on how table references appear in the ruleset. The compiler produces a policy fragment for each rule, whose size is proportional to the number of clauses generated by partial evaluation (Section 4.2). Partial evaluation replaces state table references in each rule with the disjunction of every matching tuple in that table. The largest number of clauses produced by a rule that references tables $R_1$ through $R_k$ is $|R_1| \times \cdots \times |R_k|$, which is the best achievable in the worst case. (This ensues because we don't need to lift negated disjunctions, a technical detail that we lack space to describe.) We also simplify the resulting policies, which further reduces their size in practice. To convert policies to flow-table rules, we rely upon NetCore's optimizing compiler [21].

To evaluate the quality of the NetCore policies our compiler produces, we ran a Flowlog learning-switch program, using `dpctl dump-flows` to count the maximum number of table entries it produced on per switch. We then did the same with the OCaml learning-switch module in the Frenetic repository. The Frenetic example produced a maximum of 25 rules per switch for the 3-switch tree and 81 rules per switch for the 7-switch tree. Our program initially produced 40 rules and 108 rules respectively. The increased number of rules was because Flowlog did not make use of OpenFlow's built-in flood action, whereas the OCaml program did. After adding an optimization to our learning-switch program that forced the use of the flood action, we saw the same number of table entries as with the Frenetic version. This indicates that our compiler can match the scalability of existing programs that use NetCore.

## 7 Related Work

Our work here draws on a prior workshop paper [24]. The previous version mentioned, yet did not describe or implement, external events and state. Our proactive compiler, conversion to Alloy, topology-independent verification, and implementation are also new. We compare other SDN programming languages side-by-side with Flowlog in Table 4, and discuss each in detail below.

FML [9] provides a stateful, rule-based idiom for forwarding policies. It too disallows recursion and admits negation. FML can read from, but not modify, the underlying system state. It responds to new flows reactively, whereas Flowlog proactively compiles to switch tables whenever possible. FML does not provide abstractions for external code, and does not address verification.

Frenetic [5] is a functional-reactive (FRP) language that responds to fine-grained network events. While Frenetic was originally reactively compiled, it has since been extended with proactive compilation for stateless NetCore [21] policies. State and interaction with external code must be managed by OCaml wrapper applications. Frenetic also includes switch-based events and rich query constructs; Flowlog lacks abstractions for queries, yet provides more general events. Verification of full Frenetic programs has not been addressed, although Gutz, et al. [8] use model-checking to prove slice isolation properties of NetCore policies. Our runtime is currently implemented atop Frenetic's OCaml library and uses NetCore's optimizing compiler. Guha et al. [7] have created a verified compiler for a subset of NetCore; our compiler has only been tested, not proven correct.

Pyretic [22] implements NetCore [21] in Python, and introduced sequential composition of network programs. Though we do not address program composition explicitly, sequential and parallel composition are roughly analogous to Flowlog's relational join and union, respectively. Since its initial publication, Pyretic has been extended with proactive compilation. Verification of Pyretic programs has not been discussed.

Flog [11] is stateful and rule-based. It allows recursion but not explicit negation in rule bodies; negation is implied in some cases by rule-overriding. Flog has no notion of callouts or external events unrelated to packets, and the paper does not address verification. Flog works at the microflow level, whereas Flowlog is proactive.

Procera [31] is another FRP SDN language. As Procera is embedded in Haskell, programs have access to general state and external callouts. Procera allows programs to react to external events, but does not directly support issuing events or external queries. Like Frenetic, Procera provides query functionality that Flowlog does not. To our knowledge, Procera programs have not been verified, and it is unclear whether the flow constraints they generate are proactively compiled.

Nlog [14], a rule-based configuration language, is part of a larger project on network virtualization. When system state changes, an Nlog program dictates a new virtual forwarding policy. Nlog's inability to modify controller state means that it is not "tierless". Like Flowlog, Nlog is also proactively compiled to flow tables, and our relational abstraction for callouts is similar to Nlog's. Verification of Nlog programs has not been discussed.

Maple [32] is a controller platform that unifies control- and data-plane logic; Flowlog goes further by also integrating controller state, creating a tireless abstraction. Unlike in Flowlog, Maple programs are compiled reactively, and have not been verified.

A number of other rule-based languages also merit discussion, although they were not built for SDNs and do not compile to flow tables. NDLog [17] and OverLog [18] are declarative, distributed programming languages. In these languages, each tuple in the relational state resides on a particular switch. This is in contrast to the single controller state assumed by Flowlog. These languages support recursion as in ordinary Datalog. Wang, et al. [33] verify NDLog programs via interactive theorem provers, and some of the properties they verify are topology-independent. Our compiler to Alloy requires far less user effort and is simplified by Flowlog's lack of recursion. Alvaro, et al. [3] present Dedalus, a variant of Datalog with a notion of time. Our treatment of state change is similar to theirs, except that theirs is complicated by recursion. Active networking [30] is a forerunner of SDN where packets carry programmatic instructions for switches. Like SDN, active networking has inspired language design efforts. One such is ANQL [27], a SQL-like language for defining packet filters and triggering external code. Flowlog echoes ANQL's view of packets as entries in a database, but supports more general external stimuli. Verification of ANQL has not been discussed.

There is also a rich landscape of related SDN verification. Canini, et al. [4] find bugs in Python controller

| Language | Type | State | Rec? | Neg? | Compilation | Reasoning? | Callouts |
|---|---|---|---|---|---|---|---|
| Flog [11] | Rule-Based | ✓ | ✓ | ✗ | Reactive | ✗ | ✗ |
| FML [9] | Rule-Based | ✓ | ✗ | ✓ | Reactive | ✗ | ✗ |
| Frenetic [5] | FRP | ✓$_{pol}$ | ✗ | ✓ | Reactive | ✗ | ✗ |
| Frenetic OCaml Environment | Functional | ✓$_{PL}$ | ✓$_{PL}$ | ✓ | via NetCore | ✗$_{PL}$ | ✓$_{PL}$ |
| NetCore [21] | DSL | ✗ | ✗ | ✓ | Proactive | ✓ | ✗ |
| Nlog [14] | Rule-Based | ✓ | ✗ | ✗ | Proactive | ✗ | ✓ |
| NOX [6] | Imperative | ✓$_{PL}$ | ✓$_{PL}$ | ✓ | Manual | ✗$_{PL}$ | ✓$_{PL}$ |
| Procera [31] | FRP | ✓ | ✗ | ✓ | Unclear | ✗ | ✗ |
| Pyretic [22] | Imperative | ✓$_{pol}$ | ✓$_{PL}$ | ✓ | Proactive | ✗ | ✓$_{PL}$ |
| Flowlog | Rule-Based | ✓ | ✗ | ✓ | Proactive | ✓ | ✓ |

Table 4: SDN language comparison. **Rec?** and **Neg?** mean recursion and negation, respectively. A ✓ means that a feature is present and a ✗ that it is not; ✓$_{PL}$ denotes that a feature's presence is due to embedding in a Turing-complete programming language. In the **State?** column, ✓$_{pol}$ indicates that maintaining a stateful forwarding policy is possible, but that general state requires wrappers in a Turing-complete language. In the **Reasoning?** column, a ✗$_{PL}$ indicates that *sound* reasoning is made non-trivial by Turing-completeness, and a ✗ means that verification has not been attempted.

programs. Their work required the creation of a purpose-built model-checker. Although their tool successfully finds bugs, it is limited by the undecidability of Python code analysis. Flowlog's expressivity is deliberately limited to avoid this concern. Skowyra et al. [29] use model-checking to find bugs in SDN programs sketched in their prototyping language, but focus solely on verification, not execution. Other reasoning tools [2, 8, 13, 19, 25, 34] analyze a fixed, stateless forwarding policy, either statically or at runtime. Including transitions between states, as we do, is necessarily more complex.

## 8 Discussion and Conclusion

To our knowledge, Flowlog is the first tierless SDN programming language, the first stateful rule-based language for SDNs that proactively compiles to flow-table rules, and the first such language to provide rich interfaces to external code. Tierlessness simplifies the process of SDN programming and simultaneously enables cross-tier verification of the SDN system.

Since tierlessness precludes manual handling of flow-table rules, automatic flow-table management is necessarily a key part of any tierless SDN language. There are other such strategies besides proactive compilation; a prototype version of Flowlog simply sent all packets to the controller. However, a proactive approach minimizes controller interaction and thus shows that a tierless language can be performant.

In order to support efficient, proactive compilation and verification, we opted to limit Flowlog's expressive power. Even with these limitations, we have built non-trivial applications. Moreover, events and remote tables allow Flowlog programs to access, when necessary, external code in languages of arbitrary power.

**Future Work** It is possible to strengthen Flowlog without abandoning our limitations on expressive power. We plan to migrate to a newer version of OpenFlow soon, which removes several uncompilable constructs in Table 2. Flowlog's general event framework could support a query system like that seen in Frenetic [5] and other languages. Flowlog's relational idiom supports the addition of new features. For instance, we have recently added address masking (e.g., matching packets coming from `10.0.0.0/24`) to Flowlog by taking advantage of the fact that masks are simply relations over IP addresses.

There are also several promising directions to take verification in Flowlog. For instance, we suspect that Flowlog's restrictions could enable the sound and complete use of techniques like symbolic execution for verifying trace properties. Another important analysis, change-impact—which describes the *semantic* consequences of a program change—is undecidable for general-purpose languages, yet is decidable for Flowlog.

**Acknowledgments** We thank the anonymous reviewers for their comments. We are grateful to Daniel J. Dougherty, Kathi Fisler, Rodrigo Fonseca, Nate Foster, Arjun Guha, Tim Hinrichs, Jonathan Mace, Sanjai Narain and others at Applied Communication Sciences, Joshua Reich, and David Walker, for discussions and feedback. We are grateful to Jennifer Rexford for several enlightening conversations, for shepherding this paper, and for her trenchant analysis of drinking styles. We thank the Alloy, Frenetic, and XSB teams for excellent software that we could build upon. This work is partially supported by the NSF. Andrew Ferguson is supported by an NDSEG Fellowship. Michael Scheer was supported by a Brown University Undergraduate Research Award.

# References

[1] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Al-Shaer, E., and Al-Haj, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Workshop on Assurable and Usable Security Configuration* (2010).

[3] Alvaro, P., Marczak, W. R., Conway, N., Hellerstein, J. M., Maier, D., and Sears, R. Dedalus: Datalog in time and space. In *Datalog Reloaded 2010* (2010), pp. 262–281.

[4] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE way to test OpenFlow applications. In *Networked Systems Design and Implementation* (2012).

[5] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: A network programming language. In *International Conference on Functional Programming (ICFP)* (2011).

[6] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: Towards an operating system for networks. *ACM Computer Communication Review 38*, 3 (July 2008), 105–110.

[7] Guha, A., Reitblatt, M., and Foster, N. Machine-verified network controllers. In *Programming Language Design and Implementation (PLDI)* (2013).

[8] Gutz, S., Story, A., Schlesinger, C., and Foster, N. Splendid isolation: A slice abstraction for software-defined networks. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[9] Hinrichs, T., Gude, N., Casado, M., Mitchell, J., and Shenker, S. Practical declarative network management. In *Workshop: Research on Enterprise Networking (WREN)* (2009).

[10] Jackson, D. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, April 2006.

[11] Katta, N. P., Rexford, J., and Walker, D. Logic programming for software-defined networks. In *Workshop on Cross-Model Design and Validation (XLDI)* (2012).

[12] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *Networked Systems Design and Implementation* (2012).

[13] Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[14] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Gude, N., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.-H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J., Thakkar, P., Wendlandt, D., Yip, A., and Zhang, R. Network Virtualization in Multi-tenant Datacenters. In *Networked Systems Design and Implementation* (2014).

[15] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. Onix: a distributed control platform for large-scale production networks. In *Operating Systems Design and Implementation* (2010).

[16] Lantz, B., Heller, B., and McKeown, N. A network in a laptop: Rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks* (2010).

[17] Loo, B. T., Condie, T., Garofalakis, M. N., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. Declarative networking. *Communications of the ACM 52*, 11 (2009), 87–95.

[18] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. Implementing declarative overlays. In *Symposium on Operating Systems Principles* (2005).

[19] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with Anteater. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (2011).

[20] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communication Review 38*, 2 (Mar. 2008), 69–74.

[21] Monsanto, C., Foster, N., Harrison, R., and Walker, D. A compiler and run-time system for network programming languages. In *Principles of Programming Languages (POPL)* (2012).

[22] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software-defined networks. In *Networked Systems Design and Implementation* (2013).

[23] Nelson, T., Dougherty, D. J., Fisler, K., and Krishnamurthi, S. Toward a more complete Alloy. In *International Conference on Abstract State Machines, Alloy, B, and Z* (2012).

[24] Nelson, T., Guha, A., Dougherty, D. J., Fisler, K., and Krishnamurthi, S. A balance of power: Expressive, analyzable controller programming. In *Workshop on Hot Topics in Software Defined Networking* (2013).

[25] Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., and Gu, G. A security enforcement kernel for OpenFlow networks. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[26] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for network update. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (2012).

[27] Rogers, C. M. ANQL - an active networks query language. In *International Working Conference on Active Networks* (2002).

[28] Sagonas, K., Swift, T., and Warren, D. S. XSB as an efficient deductive database engine. In *International Conference on the Management of Data* (1994).

[29] Skowyra, R., Lapets, A., Bestavros, A., and Kfoury, A. Verifiably-safe software-defined networks for CPS. In *High Confidence Networked Systems (HiCons)* (2013).

[30] Tennenhouse, D. L., Smith, J. M., Sincoskie, W. D., Wetherall, D. J., and Minden, G. J. A survey of active network research. *IEEE Communications Magazine* (1997).

[31] Voellmy, A., Kim, H., and Feamster, N. Procera: A language for high-level reactive network control. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[32] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (2013).

[33] Wang, A., Basu, P., Loo, B. T., and Sokolsky, O. Declarative network verification. In *Practical Aspects of Declarative Languages* (2009).

[34] Xie, G. G., Zhan, J., Maltz, D. A., Zhang, H., Greenberg, A., Hjalmtysson, G., and Rexford, J. On static reachability analysis of IP networks. In *IEEE Conference on Computer Communications* (2005).