A Balance of Power: Expressive, Analyzable Controller Programming

Tim Nelson
Worcester Polytechnic Institute
tn@cs.wpi.edu

Arjun Guha
Cornell University
arjun@cs.cornell.edu

Daniel J. Dougherty
Worcester Polytechnic Institute
dd@cs.wpi.edu

Kathi Fisler
Worcester Polytechnic Institute
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

ABSTRACT

Configuration languages for traditional network hardware are often fairly limited and hence easy to analyze. Programmable controllers for software-defined networks are far more flexible, but this flexibility results in more opportunities for mis-configuration and greatly complicates analyses. We propose a new network-programming paradigm that strikes a balance between expressive power and analysis, providing a highly analyzable core language while allowing the re-use of pre-existing code written in more complex production languages.

As the first step we have created FlowLog, a declarative language for programming SDN controllers. We show that FlowLog is expressive enough to build some real controller programs. It is also a finite-state language, and thus amenable to many types of analysis, such as model-checking. In this paper we present FlowLog, show examples of controller programs, and discuss analyzing them.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.3 [**Programming Languages**]: Miscellaneous

General Terms

Design, Languages, Verification

Keywords

Software-Defined Networks; OpenFlow; Verification; Network-Programming Languages

1. INTRODUCTION

Networking is now faced with a classic programming language design tradeoff: that between analyzability and expressive power. In the rush to create languages for a new domain, expressive power often wins. This is natural: the first task of a language is to express functionality. Thus, most controller programs are currently written using libraries in general-purpose (henceforth, "full") programming languages like Java and Python. Even tailor-made SDN languages (e.g., Frenetic [5] and NetCore [15]) are embedded in full programming languages to support dynamic policies and state. (We discuss other related work later.)

Because of the power of these languages, analysis of controller programs is non-trivial. In languages with concurrency and state it can be extremely difficult to form useful static summaries of program behavior. Add features like `eval` and dynamic loading, and it can be hard or impossible to determine what the program's full code even *is*. As a result, analyses are either dynamic or use unsound techniques such as symbolic evaluation. Because the controller programs represent infinite-state systems, papers do not even bother mentioning the lack of completeness; any form of reliable static analysis is heavily compromised and tools can usually only offer a "best effort" outcome. Approaches like directly analyzing static forwarding tables [2] cannot avail of the rich semantic knowledge contained in the original controller program. These analyses may be sufficient in some contexts but are not when reliability is a must. We therefore believe it is worth exploring other design strategies for controller programming.

One natural alternative is to program controllers written in languages with limited expressive power, e.g., finite-state languages. This is attractive from an analytic standpoint, but impractical. Controller authors sometimes need to take advantage of the full power of programming, and methodologies that don't provide it will (rightly) be deemed impractical.

The Middle Path.

We propose an alternative SDN programming experience based on the following tenets:

- Controller authors should have the freedom to use both restricted and full languages. Which one they use is a function of the analytic power they need, previously written code they wish to reuse, etc.

- A single controller program should be able to combine elements from both restricted and full languages. This provides the most flexibility and puts the trade-off in the hands of the author. This division appears in many SQL engines, wherein callouts to user-defined functions play the role of the "full" language.
- Nevertheless, because of the difficulty of recovering reasoning power once we permit full expressive power, we prefer that the restricted language reside "outside" and the "full" language play the role of callouts or libraries.

In short, we are aiming for the Alan Kay principle of "Easy things should be easy, while hard things should be possible".

From an analytic perspective, the restricted outer language will be amenable to rich, sound, and complete analyses. In this, we are not interested only in "verification"; for many years we have worked on other rich analyses such as change-impact [16], which let authors understand what impact their edits will have to overall system behavior. Having a restricted language also opens the door to sophisticated forms of synthesis and much else.

But what to make of the embedded full-language components? Our view is that performing the analysis over the restricted language will enable us to automatically infer *constraints*, such as preconditions, postconditions, and invariants, on the behavior of the full language components being invoked. Discharging those constraints becomes a language-specific, task-specific problem: solutions may range from using existing tools for controller languages (such as NICE [3] for Python) to applying richer model-checking techniques to performing dynamic monitoring of invariants.

In this paper we take the first step towards this vision, which is to present FlowLog (section 3), a candidate for the restricted language. FlowLog allows for both reading and writing controller state, provides a concise idiom for stating forwarding policies, and allows for the automatic inference of when packets must be sent to the controller (section 5). Perhaps surprisingly, FlowLog will be finite-state; it can thus be model-checked efficiently (section 4). The more useful test will be whether we can write any interesting controller programs at all in this language; we show that FlowLog alone can express useful controller programs (section 2). When we hit expressive limits, however, our goal is not to keep growing this language—down that path lies `sendmail.cf` and other sulphurous designs—but to call out to full-language code.

2. FLOWLOG BY EXAMPLE

We introduce FlowLog through a pair of examples.

2.1 MAC Learning

We begin with a controller that learns which physical ports have connectivity to which layer-2 (*MAC*) addresses. It also provides forwarding instructions to the switches that take this learning into account.

As the name suggests, FlowLog is inspired by Datalog [1]. FlowLog programs consist of a set of *rules*, each of which has a *head* and a *body* separated by the implication ←. Each rule describes one of two things: a modification of the controller's internal state (e.g., the rules with **+learned** and −**learned** in their head) or a packet-handling action (the rules with **emit** in the head). When the conditions in the body are met, the action specified in the head is performed.

```
  +learned(pkt, sw, pt, mac) ←
    pkt.locSw == sw,
    pkt.dlSrc == mac,
    pkt.locPt == pt;
5 emit(pkt, newp) ←
    learned(pkt.locSw,newp.locPt,pkt.dlDst),
    pkt.locSw == newp.locSw,
    pkt.header == newp.header;
  emit(pkt, newp) ←
10  not learned(pkt.locSw, any, pkt.dlDst),
    pkt.locSw == newp.locSw,
    newp.locPt != pkt.locPt,
    pkt.header == newp.header;
  −learned(pkt, sw, pt, mac) ←
15  pkt.locSw == sw,
    pkt.dlSrc == mac,
    pkt.locPt != pt;
```

Listing 1: MAC Learning in FlowLog

The first rule tells the controller when to learn a port-to-address mapping. The rule's head (line 1) instructs the controller that when it receives a packet `pkt`, it should add any tuples `<sw, pt, mac>` that satisfy the rule body to its `learned` relation. The body simply ensures that the tuple's contents represent the packet's location switch (line 2), source MAC address (line 3), and arrival port (line 4).

The second rule, beginning on line 5, handles forwarding when the controller has learned an appropriate port. It informs the controller that when a packet `pkt` is received, all new packets `newp` that match the rule body should be sent. That is, the new packet must be sent out the port dictated by the `learned` relation (line 6), on the same switch as received it (line 7). All other attributes of the packet are left unmodified (line 8). This last stipulation is necessary because FlowLog will cause *all* packets that match the conditions in the rule body to be produced. This behavior is extremely useful for flooding packets on multiple ports (e.g., in the next rule).

The third rule, beginning on line 9, handles the other forwarding case: when the controller has not yet learned an appropriate port to send a packet on. Barring two differences, it is identical to the second rule: the negative use of the `learned` relation on line 10 enforces that, for the rule to fire, *no* appropriate port has been learned. Line 12 adds a condition on the output port that ensures that the switch will not backflow traffic.

Finally, the rule beginning on line 14 allows the controller to account for *mobility* of hosts by removing outdated port-to-address mappings.

2.2 ARP Cache

Our second example involves the controller acting as a cache for address-resolution protocol responses. ARP allows hosts to discover which datalink-layer (MAC) addresses are currently bound to a specific network-layer (IP) address. The program must, therefore, do several things: allow the dissemination of ARP requests across the network, spy on the replies to those requests in order to update its internal state, intercept requests for which the reply is already known, and spawn reply packets for known requests. Excluding the un-learning of outdated ARP assignments, the entire program is given below:

```
   +learned(pkt, ip, mac) ←
     pkt.dlTyp == 0x0806,
     pkt.nwProto == 2,
     not learned(pkt.nwSrc, any),
 5   ip == pkt.nwSrc,
     mac == pkt.dlSrc;
   emit(pkt, newpkt) ←
     pkt.dlTyp == 0x0806,
     (pkt.nwProto == 2 ||
10     (pkt.nwProto == 1 &&
        not learned(pkt.nwSrc, any))),
     newpkt.locSw == pkt.locSw,
     newpkt.locPt != pkt.locPt,
     newpkt.header == pkt.header;
15 emit(pkt, newpkt) ←
     pkt.dlTyp == 0x0806,
     pkt.nwProto == 1,
     newpkt.dlTyp == 0x0806,
     newpkt.nwProto == 2,
20   learned(pkt.nwDst, newpkt.dlSrc),
     newpkt.loc == pkt.loc,
     newpkt.nwDst == pkt.nwSrc,
     newpkt.dlDst == pkt.dlSrc,
     newpkt.nwSrc == pkt.nwDst,
25   newpkt.otherwise == pkt.otherwise;
```

Listing 2: ARP Cache in FlowLog

In this program, `pkt.dlTyp == 0x0806` checks that the packet is an ARP packet, for which `pkt.nwProto == 1` denotes a request and `pkt.nwProto == 2` a reply.

The first rule updates the controller's state when the network sees an ARP reply for a previously-unknown IP address. The negated condition on line 4 encodes that the IP is indeed not in the current state.

The first **emit** rule, beginning on line 7, allows the propagation of all ARP replies as well as ARP requests for unknown IP addresses. Line 9 floods reply packets, and line 10 stipulates that the IP address must unknown to apply to request packets. Line 13 prevents backflow, and line 14 explicitly preserves the packet header.

The second **emit** rule (starting line 15) spawns reply packets for IP addresses that the controller knows. The generated packet `newpkt` is constrained to be an ARP reply (line 18), while the originating packet must be an ARP request (line 16). Its source MAC address is obtained from the controller's state (line 20). The reply is sent from the same switch and port that observed the request (line 21) but with the packet's destination and source addresses reversed (lines 22–24). All other header fields are preserved in the new packet (line 25). As before, omitting this restriction would cause FlowLog to produce packets to cover all possibilities.

3. FLOWLOG UNDER THE HOOD

These examples show that FlowLog offers a concise means for describing how a controller ought to react to packets. It can govern how those packets are forwarded, cause entirely new packets to be produced and sent, and modify the controller's internal state, by both adding and removing information.

FlowLog programs have a relational notion of state, which the program can both read and update. A relational state is simply a collection of concrete tables (equivalent to a relational database). For example, in our ARP-cache implementation, the controller's knowledge of which IP addresses are mapped to which ethernet addresses is contained in the `learned` relation.

FlowLog syntax is inspired by Datalog [1] with negation, to which we have added a notion of packet fields and slight syntactic sugar. Furthermore, we require that there be no cyclic dependencies between relation names across programs, i.e., FlowLog is *non-recursive.*

FlowLog Semantics.

Let P be a FlowLog program, and let $sig(P)$ denote the state relations used in P. If rules exist to define them, it is natural to consider also the following generated relations, which have special meaning in FlowLog:

1. **emit**, a special binary relation;
2. $\texttt{+R}$, where $\texttt{R} \in sig(P)$ and $arity(\texttt{+R}) = arity(\texttt{R}) + 1$;
3. $\texttt{-R}$, where $\texttt{R} \in sig(P)$ and $arity(\texttt{-R}) = arity(\texttt{R}) + 1$

Together, these relations define the semantics of P. Each `+R` and `-R` relation is effectively a *function* in relation form, mapping incoming packets to sets of tuples to add or remove from `R` in the post-state. Likewise, the **emit** relation produces sets of output packets. The arity increase is what makes this functional behavior possible—by adding a field for the incoming packet. Given a relational state S for $sig(P)$ (i.e., a set of tuples for each state relation), the meaning of P at S is a function that accepts individual packets and produces two things: the next relational state S' and a set of packets to output on the network.

Let S^+ be the interpretation of the **emit**, `+R`, and `-R` relations given by Datalog semantics [1]. We denote the relation `R` in a state S by $\texttt{R}^S$, and tuple-sets resulting from incoming packet *pkt* by subscripting. E.g. $\texttt{+R}^{S^+}_{pkt}$ is the set of tuples to be added to `R` when *pkt* arrives while the controller is in state S. Now the semantics of P is the function $[[P]](S, pkt) = (S', \overline{p}')$ such that:

1. For each $\texttt{R} \in sig(P)$: $\texttt{R}^{S'} = (\texttt{R}^S \setminus \texttt{-R}^{S^+}_{pkt}) \cup \texttt{+R}^{S^+}_{pkt}$
2. $\overline{p}' = \{pkt' \mid \langle pkt, pkt' \rangle \in \textbf{emit}^{S^+}\}$

4. ANALYZING FLOWLOG

A FlowLog program naturally defines a finite-state transducer[1] on controller states. This is easy to see from our operational semantics: each transducer state is a set of relations, and each transition has arriving packets as input and the resulting output packets as output. Due to this fact, many rich analyses of FlowLog programs are not only possible [10], but even efficient. These include change-impact analysis [16], verification, bug-finding, and more; the key is that, regardless of the questions asked or the tools used, FlowLog's simple structure and carefully chosen restrictions ease analysis in general.

As an illustrative example, we describe the application of model-checking below. Model-checking is a sound and complete exploration of a finite state-space. For our experiments we used the SPIN model checker [9]. First we discuss the properties to verify, then examine performance.

[1]A finite-state automaton that reads *and writes* a symbol at each step.

Verification Properties.

We begin by considering MAC-learning. Properties one might want to verify include:

Consistency of State All switch-address pairs are mapped to at most one port.

Preservation of Connectivity If a packet appears at an endpoint, it will eventually exit the network at the appropriate point.

No Loops The algorithm never induces routing loops.

Eventually Never Flood The algorithm will eventually stop flooding.

The "Eventually Never Flood" property is especially interesting. Though it is a behavioral property, it is also equivalent to saying that the second **emit** rule (listing 1, line 9) will always eventually stop firing. In our experience analyzing rule-based languages, we often find it easier to ascribe semantic properties to a small number of syntactic rules, making it easier to understand and debug programs.

ARP-cache shares some properties with MAC-learning and differs on others. The ARP-cache properties we considered are:

Consistency of State One IP address must not map to multiple MAC addresses.

Reliability All ARP requests will be answered.

Cache All ARP requests for known addresses will be stopped and replied to, without propagation in the network.

We have verified each of these properties, as well as caught errors that were artificially injected. Checking against injected errors is important even when a system passes, because all system descriptions—even incorrect ones—will pass (erroneous) vacuously true property statements. Also, it is important to distinguish the times for verification and falsification, since these can be quite different.

Performance of Analysis.

To evaluate the performance of our analyses, we use metrics and network topologies (two to three switches) similar to those of Canini et al. [3]. We note that our rough use of out-of-the-box model-checking tools sees performance similar to theirs, which used a custom-built checker. This is because the restricted nature of FlowLog allows for fairly simple modeling. All tests are run on an Intel Core i5-2400 CPU with 8 GB of memory, running Windows 8 64-bit. We use Spin version 6.2.3, and make the modeling assumption that packets are injected into the network serially. For MAC-learning, we allow host mobility, limiting the number of times hosts can relocate to three. We test all MAC-learning properties on two different network topologies: an acyclic topology with two switches and a cyclic topology with three switches. For ARP-cache, we test without allowing host mobility on a two-switch topology.

Spin verifies **Consistency of State**, **Reliability** and **Cache** in under 20 seconds each, using no more than 500 MB of memory. **Preservation of Connectivity** fails in the presence of host-mobility, and Spin yields an example of this on the 2-switch topology in 140 milliseconds. **No Loops** is correct on the acyclic topology, but Spin finds potential routing loops on the cyclic topology in 40 milliseconds.

Eventually Never Flood fails on both topologies. Spin returns a short illustration of this failure in 160 milliseconds. The property fails for a somewhat subtle reason: address-port mappings are only learned locally. Thus, packets from an unlearned source address to a previously learned destination will not be flooded, preventing other switches from learning the right port for that address. This oversight can be fixed with a single FlowLog rule (not shown in the listing) that floods the *first* (and only the first) packet seen from an unknown source, even if its destination is known.

5. EXECUTING FLOWLOG PROGRAMS

Since FlowLog is a refinement of Datalog, a first option for execution would be using an out-of-the-box Datalog engine. Given a specific packet, running such an engine would certainly yield the set of packets that ought to be emitted, as well as any necessary state edits. However, notifying the controller of every packet arrival places unnecessary overhead on both the network and the controller hardware. A real-world controller must provide flow rules proactively to the switches, and only be informed when a packet arrival might cause the controller's state to be updated or cause behavior that the switch perform do on its own.

Fortunately, there is a way to accomplish this with FlowLog. The NetCore [15] controller provides a runtime environment and a verified compiler [6] from programmatically generated policies to OpenFlow switch tables. The compiler allows controllers to issue rules proactively, making use of wildcarding. We therefore intend to execute FlowLog programs within NetCore, using its verified compiler whenever possible.

FlowLog programs are, however, not just NetCore policies! Instead, FlowLog programs are richer in three ways: they can access controller state, they can modify that same state, and they can make non-constant modifications to packet fields. NetCore policies are stateless, and moreover their ability to modify packet fields is restricted by the nature of OpenFlow 1.0. For instance, a flow table cannot express "swap the source and destination addresses". Thus some pieces of a FlowLog program will not be expressible in a NetCore policy.

To circumvent this problem, we divide a FlowLog program in two: rules which are simple enough to be (modulo the current controller state) compiled to flow tables, and rules which will require direct controller action in some way. Here we find another advantage of our non-recursive, rule-based design: it is easy to characterize the packets that the controller may need to take direct action on. These packets are the ones that can be matched by the bodies of state-modification rules or rules that modify packets in a non-constant fashion. The controller can process these exceptional packets using either a Datalog engine or custom algorithms. By proactively compiling as much of the program as possible to flow tables, we reduce overhead; furthermore, we can statically determine when the overhead will occur, making it easier to tune for performance if the cost of updating switches is found to be expensive. As an added benefit, automatic computation of controller-notifications eliminates the potential for bugs (and inefficiency) in controller notification.

6. RELATED WORK

There have been many Datalog-derived and -related programming languages, especially in the SDN domain. Hinrichs, et al. [8] present the FML language for network management, which, like FlowLog, is based on non-recursive Datalog with negation. FML programs have far more expressivity in their rule conditions than do FlowLog programs, yet they do not provide a way for the program to modify state.

The Declarative networking effort [13] uses declarative languages to program distributed systems, focusing on temporal reasoning and message-passing between network nodes, rather than programming for a centralized controller. While these languages are designed to encourage code correctness, analysis has not been a goal equal to expressive power in their design.

Katta et al. [11] describe a language called Flog which has a similar appearance to ours. Flog does not give the ability to explicitly remove tuples in the next state (as we do with `-R` rules). Instead, Flog's next state is empty unless tuples are explicitly carried over. In terms of expressive power, Flog allows recursion in its rules, but not explicit negation; we allow the opposite. Flog's forwarding policies allow *implicit* negation via rule-overriding, as in their example implementation of MAC learning. Unsurprisingly, the inclusion of recursion gives Flog a significant advantage in expressiveness over FlowLog. However, Flog is not designed with analysis in mind; both are equal considerations in FlowLog.

Field et al. [4] present a declarative language model, designed for web applications, that allows a database to react to external updates. Like FlowLog, their language is inspired by Datalog, but it has significantly more expressive power than FlowLog. It supports recursion, provides numerics, and offers features designed for interaction between processes. Thus, one could view their language as a possible evolution of FlowLog, were we only concerned with expressive power and usability, and not with analysis or the embedding of third-party code. They do not discuss analysis, and the addition of recursion renders some analysis questions undecidable.

Voellmy et al. [19] present Procera, a functional-reactive policy language for SDNs. Procera allows the reactive modification of state as well as packet filtering. It is more expressive than FlowLog, for instance allowing arithmetic, but does not address analysis.

Skowyra et al. [18] compile their rapid-prototyping language to model-checkers in order to verify properties and find bugs in specifications. Their language (VML) is designed for prototyping and not execution, although it is widely applicable outside SDNs. Like us, they use MAC learning as a testbed.

NICE [3] uses a mix of symbolic execution and model-checking to find bugs in NOX controllers written in Python. However, since Python is a full language, symbolic execution is unsound and required implementing a specialized model-checker. We propose a language-design strategy that will increase the available efficiency and soundness of analysis via a finite-state surface language, while still embracing tools like NICE for analysis of arbitrary call-outs, thereby ringfencing the regions of the controller program that may need to rely on unsound methods.

FlowChecker [2] analyzes low-level OpenFlow flow tables. switches. In contrast, FlowLog is an efficiently analyzable language for *controller* programming, and thus operates at a different level of abstraction. Similarly, Anteater [14] uses SAT-solving techniques to analyze the switch forwarding rules on a network. Likewise, Gutz et al. [7] perform analysis of stateless controller policies when analyzing their slice abstractions. In contrast, analysis of FlowLog must take shifting controller state into account, rather than assuming a fixed switch forwarding base. Guha et al. [6] recently developed a machine-verified controller for NetCore that ensures that *static* NetCore policies are correctly translated to OpenFlow messages. In contrast, our work addresses the verification of *dynamic* control programs.

Other analysis tools such as FortNOX [17] and Veriflow [12] act only at runtime; we have focused on designing a language for efficient *static* analysis as well.

7. NEXT STEPS

We have focused so far on the design of a decidable yet expressive language. Because our design is based on permitting external code in full languages, our current work advances on two fronts:

1. Automatically extracting interfaces on callouts. We can exploit the long history of work on interface generation in the verification community, though the nature of composition here—being sequential rather than parallel—has been studied less.

2. Focusing on tasks *other than* traditional verification. Our primary emphasis will be change-impact analysis, but we are also interested in program synthesis approaches.

Our goal is to produce a tractable, expressive, analysis-friendly language that enables rich reasoning to create reliable controllers.

Acknowledgments.

We thank Andrew Ferguson, Vyas Sekar and Seyed Fayazbakhsh for feedback on drafts of this paper. We are grateful for support from the US NSF and Google.

8. REFERENCES

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

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

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

[4] J. Field, M.-C. Marinescu, and C. Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theoretical Computer Science*, 2009.

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

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

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

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

[9] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

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

[12] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Networked Systems Design and Implementation*, April 2013.

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

[14] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

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

[16] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *USENIX Large Installation System Administration Conference*, 2010.

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

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

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