Procera: A Language for High-Level Reactive Network Control

Andreas Voellmy
Yale University
51 Prospect ST
New Haven, CT, U.S.A.
andreas.voellmy@yale.edu

Hyojoon Kim
Georgia Tech
School of Computer Science
266 Ferst Drive
Atlanta, GA, U.S.A.
joonk@gatech.edu

Nick Feamster
University of Maryland
Dept. of Computer Science
College Park, MD, U.S.A.
feamster@cs.umd.edu

ABSTRACT

Our previous experience building systems for implementing network policies in home and enterprise networks has revealed that the intuitive notion of network policy in these domains is inherently dynamic and stateful. Current configuration languages, both in traditional network architectures and in OpenFlow systems, are not expressive enough to capture these policies. As a result, most prototype OpenFlow systems lack a configurable interface and instead require operators to program in the system implementation language, often C++. We describe Procera, a control architecture for software-defined networking (SDN) that includes a declarative policy language based on the notion of functional reactive programming; we extend this formalism with both signals relevant for expressing high-level network policies in a variety of network settings, including home and enterprise networks, and a collection of constructs expressing temporal queries over event streams that occur frequently in network policies. Although sophisticated users can take advantage of Procera's full expressiveness by expressing network policies directly in Procera, simpler configuration interfaces (e.g., graphical user interfaces) can also easily be built on top of this formalism.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Centralized networks*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network Management*; C.3 [**Special Purpose and Application-Based Systems**]: []; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages, Haskell*

Keywords

OpenFlow, Software-defined Networking, Network Configuration, Functional Reactive Programming, Haskell

1. INTRODUCTION

Network operators and researchers have often discussed the need for a network configuration language that can express high-level network policies, in contrast to the status quo, whereby configuration is low-level, mechanism-focused, and vendor specific [5,7,18]. Conventional methods result in systems that are complex, error-prone, and hard to manage [6,9,14,19]. Software defined networking (SDN) offers the opportunity to make networks easier to configure by providing richer configuration methods, and indeed such systems have been proposed. As an example, Flow Management Language (FML) [13] provides a simple rule-based formalism for controlling OpenFlow [21] networks.

Many network systems implement policies that are inherently dynamic and depend on temporal conditions defined in terms of external events such as measurements of bandwidth use of hosts, intrusion detections, or specific time events. For example, intrusion detection and prevention systems detect certain sequences of events and trigger actions [23]; load balancing systems can choose a server based on load [12]; and campus networks can deny access when certain complex temporal bandwidth usage conditions occur [1].

We present Procera, a controller architecture and high-level network control language that allows operators to express the kinds of policies that we have described above, without resorting to general-purpose programming of a network controller. We have designed Procera to be *reactive* because many realistic network policies react to dynamic changes in network conditions. Procera also incorporates events that originate from sources other than OpenFlow switches, allowing it to express policy that reacts to conditions such as user authentications, time of day, bandwidth use, or server load. Procera is both expressive and extensible, so users can easily extend the language by adding new constructs.

Procera has the following features:

- Procera controllers output *flow constraint functions* that a lower-level network controller uses to constrain its own behavior, thereby allowing the high-level control to be simple and to cleanly separate lower-level network issues, such as routing and flow-table management.
- Procera applies the principles of *functional reactive programming (FRP)* [26], which provides a declarative, expressive, and compositional framework for describing reactive and temporal behaviors. Procera includes a collection of domain-specific temporal operators that allow users to easily express the values of time-varying sets and dictionaries in terms of event histories.
- Procera can be customized with a collection of primitive event streams, but does not by default have access to OpenFlow events, such as flow request events. This func-

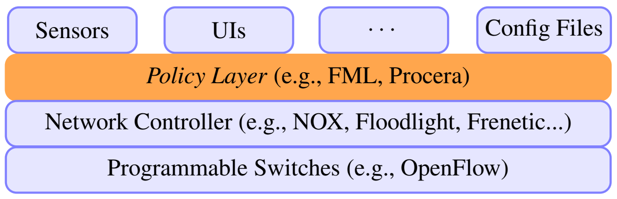
Sensors
UIs
· · ·
Config Files
Policy Layer (e.g., FML, Procera)
Network Controller (e.g., NOX, Floodlight, Frenetic...)
Programmable Switches (e.g., OpenFlow)

Figure 1: System Architecture.

tional division permits scalable implementations that process OpenFlow events in parallel, even while processing non-OpenFlow events sequentially.

This paper illustrates the need for a high-level, reactive policy language; presents Procera, a language that permits operators to express high-level, reactive network policies; and illustrates how Procera can help operators express these policies through a series of examples that are motivated from our experience with configurations in both campus networks and home networks. We are developing a version of Procera that can instantiate policies in our underlying Lithium controller [16].

The rest of this paper is organized as follows. Section 2 defines the notion of a *policy layer* that enables a network controller to respond to higher-level events, and explains Procera in the context of existing network control technologies. Section 3 motivates the need for a reactive policy layer with several examples and shows that existing languages and control models do not allow network operators to easily express even simple reactive network policies. Sections 4 and 5 describe the Procera language and its use, through a series of examples. Section 6 describes related work, and Section 7 concludes with a summary and agenda for future work.

2. CONTROL SYSTEM ARCHITECTURE

Figure 1 shows our system architecture. The lowest layer consists of programmable OpenFlow [21] switches that ultimately perform forwarding actions on the flows. On top of this layer, the network control layer exchanges messages with OpenFlow switches to configure flow tables and gather statistics. There are many examples of network controllers today, for example NOX [11], Maestro [27], Floodlight [2], Nettle [25], and Frenetic [10]. This layer is responsible for performing network-related tasks, such as routing and topology discovery.

The *policy layer* resides above the network control layer and is the focus on this paper. The policy layer acts as a supervisor, reacting to signals about relevant network events and out-of-band signals from users, administrators, and other sensors such as intrusion detection devices. In turn, the policy layer provides guidance and directives to the network controller. This guidance should be declarative to keep the supervisor simple and abstract so that the network layer may implement the guidance efficiently and flexibly. We envision that this policy layer may in fact be a policy engine that can be customized with user-specified supervisor code. FML is an example of a language that resides at the policy layer: it generates a data structure used by the network controller to handle flow requests. Similarly, Procera allows network operators to define how a network controller should react to higher-level network events. Procera could act as a language and runtime component of the Lithium event-based OpenFlow controller, which we describe in an accompanying technical report [16].

3. THE NEED FOR REACTIVE POLICY

We describe some simple examples that illustrate the need for a declarative, reactive, operator-facing network programming language, We do this by highlighting the problems that arise when using an existing policy language, FML, to specify and implement policies that arise in real-world systems.

FML provides a high-level declarative policy language based on logic programming. An FML program consists of a collection of inference rules that collectively defines a function that assigns a set of flow constraints to each packet. The constraints typically include "allow" and "deny" constraints, but may include others as well. The logic is simple, permitting neither recursion nor arithmetic constraints or functions. The FML engine maintains some state, including the users associated with particular devices, and FML policy may predicate packet constraints based on this state. This state space is fixed by the FML implementation and therefore not modifiable by the author of the network policy.

Consider the following examples, which illustrate the limitations of FML's logic. The first policy is "ban a device if its usage over the last five days exceeds 10 GB". Unfortunately, we cannot write this policy in FML directly because FML does not provide device usage data or allow arithmetic constraints. Assume for the moment, though, that we have extended FML with arithmetic inequalities and a predicate `usage(D,T,B)` which holds when device `D` has consumed `B` gigabytes of bandwidth over the last `T` days. Then we could write the above policy in FML as:

```
deny(Us, D, As, Ut, Ht, At, P, R) <- over(D).
over(D) <- usage(D,T,B), T=5, B > 10.
```

These rules should be read as implications from right to left. For example, the first line, expresses the implication that whenever a device `D` satisfies predicate `over`, then `deny(...)` holds with device `D` in the source Ethernet address field of a packet; that is, all packets from `D` should be denied.

Expressing even a minor variation on this policy, such as "*permanently* ban the device as soon as its usage over the last five days exceeds 10 GB", is much more difficult. The difficulty arises from the fact that the policy requires that the violating user to be banned from the network, even if later his five-day usage drops below the maximum allowed (which will certainly occur, since the user is banned). In other words, the policy requires that when the condition becomes true, the policy itself must change. We could try to work around this, by adding a predicate `usageOnceExceeded(Hs,Ds,Amt)`, so that we could write

```
over(D) <- usageOnceExceeded(D,T,B), T=5, B=10.
```

Such a function is fairly special-purpose for a general network policy language, but if it were the only such exception, we could live with it. Unfortunately, another minor variation to the policy causes yet more difficulties: "ban a device as soon as its last five-day usage exceeds 10 GB, and remove their ban when and only when an adminstrator resets that device". This time, a different temporal condition changes the state of the system and affects the desired policy. We could again extend FML with a predicate `usageOnceExceededAndNoResetSince(D,T,B)`, but it should clear at this point that there is no end to these variations, and that we need a more general notation for denoting any of these variations.

The key feature missing from FML is the ability for the user-defined policy logic to *affect* the state. Instead, in FML, the policy logic is passive and can only observe the state. While there are approaches, such as McCarthy's frame calculus [20] and Kowalski's event calculus [17], to enhance the logic programming approach with state-changing actions, these models substantially complicate the logic.

We instead pursue a different path, based on *functional reactive programming (FRP)*. Using this approach, users describe time-varying values by describing how their current value depends on event histories and current and prior values of other quantities. In particular, we instantiate FRP with temporal operators that produce *event histories* and define commonly occurring functions that determine the value of sets and dictionaries in terms of these event histories. For example, we will be able to define a dictionary mapping devices to their five-day usage simply by writing the following:

```
proc world → do
  recent ← since (daysAgo 5) ─≺ add (usageEvents world)
  usageTable ← group sum ─≺ recent
  returnA ─≺ usageTable
```

FRP is well-established and has been applied to a wide range of domains, including user interfaces, robotics, and network control [10, 25]; many efficient and practical FRP implementations exist. These languages allow the user to specify and invoke arbitrary functions, so we can easily support policies that involve arithmetic constraints and other functions, which would require substantial extensions to FML.

4. PROCERA: REACTIVE POLICY

The key features of the Procera policy language are (1) a core language based on functional reactive programming, (2) event comprehensions to manipulate event streams, (3) windowing and aggregation signal functions, (4) the use of function values to represent high-level policy. Each of these ideas borrows heavily from previous systems: the core reactive language is based on the ideas of Yampa [8], a domain-neutral FRP language, event comprehensions are inspired by list and monad comprehensions in Haskell, and the windowing and aggregation constructs are inspired by streaming database systems [4].

Procera is an embedded domain-specific language (EDSL) in Haskell. This allows Procera to reuse various general-purpose datatypes and constructs, such as numbers, lists, and functions. It also allows users to mix Haskell expressions with Procera constructs, provided the expressions satisfy appropriate type constraints.

4.1 Signals, Signal Functions, and Events

The main reactive concepts of Procera are *signals* and *signal functions*. A signal is conceptually a function of continuous time into some range. Often the range of a signal is numeric, but in Procera signals can range over any data type. Although the domain of a signal is conceptually continuous time, discrete time signals are supported by allowing missing values. Values have an option type, which is either present with a value or missing.

On the other hand, signal functions (also known as "systems" in other contexts) transform signals. The name emphasizes the fact that signal functions are in fact mathematical functions and therefore describe deterministic transformations of input signals to output signals. By acting on signals, signal functions can implement time-varying behavior, and in particular, can exhibit hysteresis. A simple example of this behavior is an edge detector, which outputs an event whenever its boolean input signal rises from $False$ to $True$. This signal function exhibits hysteresis because the value of the output signal does not depend solely on the value of input signal at the same time; in other words, the signal function $edge$ has internal state.

In Procera, the programmer defines signal functions using the *arrow syntax* [22], which provides a syntactic analog of signals and systems diagrams consisting of boxes and wires. For example,

```
proc x → do
  y ← integral ─≺ x
  z ← integral ─≺ y
  returnA ─≺ z + y + 1
```

defines a signal function that integrates the input signal (named x) twice and scales the result (this example is only illustrative: $integral$ is not in Procera). The notation allows users to bind variables to stand for instantaneous values of signals, as in x, y, and z above, and these can be used in expressions to denote other instantaneous values (e.g., $z + y + 1$ above) and to wire-up to signal functions (e.g., y is fed into $integral$). Note that the instantaneous values are all synchronous; for example, the output of the signal function at time t above involves adding the values of z and y taken at that same time t.

As we mentioned earlier, Procera supports discrete signals by having the signals carry values of the $Event$ data type. Procera provides an *event algebra* that allows operations for filtering, transforming, merging, and joining event streams. For example, merging of two event streams e_1 and e_2 is written $e_1 .|. e_2$ and denotes the events that occur in either of the event streams. Most of this algebra is accessed most conveniently using *event comprehensions*, a notation very similar to list comprehensions that supports transforming, filtering and joining event streams. For example, if e stands for the instaneous values of an event stream carrying natural numbers, we could write:

$$[n + 1 \mid n \leftarrow e, prime\ n]$$

to stand for the instantaneous values of an event stream which keeps just those events from e that carry prime numbers, and adds one to the values of the filtered event stream.

4.2 Windowing and Aggregating

We introduce a collection of signal functions and abstract data types that allow users to conveniently describe commonly occurring reactive policies. Table 1 enumerates these constructs.

The constructs fall into two groups. The members of the first group produce windowed event histories. In particular, $since\ dt$, $limit\ size$, and $limitBy\ attribute\ size$ implement time-based, count-based, and partitioned-count-based windowing. The $clockResetWindow\ \ next$ construct accumulates a window until the next clock event given by $next$, when the window is cleared. Each of these constructs consumes and produces event histories, and therefore can be composed. For example, $since$ and $limit$ can be used together to define a windowed history that includes only the last few events that occur within a given time window. We will provide example uses of these constructs in Section 5.

The second group of constructs includes $accumSet$, $accumList$, and $group$. These signal functions take event histories as input and compute some value of the input history. For example, the value of the output signal of $accumSet$ at any time is the set whose members are currently in the input history. The $group\ \ op$ construct outputs a time-varying dictionary from an event history of key-value pairs using the given operation op to aggregate values of the same key.

History signals carry their history incrementally, in the form of additions and removals from the history. This design allows window and aggregation operations to incrementally compute re-

since dt	Windows a history to the past *dt* seconds
limit size	Windows a history to the last *size* number of events.
limitBy attribute size	Limits the input history by keeping only the last *size* number of events for each value of the attribute.
clockResetWindow next	Windowed history that is cleared at the time indicated by *next*.
accumList	Accumulates a sequence of events from a windowed history.
accumSet	Accumulates a set of events from a windowed history.
group op	Accumulates a dictionary from a history of key-value pairs, using *op* to combine values for the same key.
last1PerGroup	Accumulates a dictionary from a history, mapping keys to the last occurring value for the key.
add e,*remove e*, $ar_1 \oplus ar_2$	Event additions, removals, and combinations thereof.

Table 1: Key Procera constructs.

allow	Allow the packet
deny	Do not allow the packet
rateLimit rate	Rate limit the flow to the given rate
redirect host	Redirect the flow to the specified host
$x \odot y$	Constrain according to both x and y

Table 2: Constraints.

sults. It also allows the same aggregation operations to act on input signals of event additions and removals other than those produced by window operators. Section 5 will show example uses of these operations. Procera includes an abstract data type *AddRemoveEvent a*, and functions *add*, *remove*, and $\oplus$ to insert and remove events and to merge two *AddRemoveEvent a* streams.

4.3 Input Signals & Flow Constraints

The input to the main Procera signal function is a *world signal* whose instantaneous values have the abstract *World* type. While the exact details of this data type are application-specific, it is typically a record of attributes of the environment, and includes components that indicate the presence or absence of external events. For example, in the following section we write *authEvents world* to denote the authentication events that might be present in the current *world* value.

The output of a Procera controller is a signal carrying *flow constraint functions*. A flow constraint function determines the constraints that are applied to a flow. The constraints will typically include constructs to allow or deny flows, but may include other constraints according to the needs of the particular system. Table 2 lists some constraints useful in home and campus network management systems. For convenience, we include a *constraint combination* operation denoted with $\odot$, which hides the detailed representation of the constraint data type.

5. PROCERA IN ACTION

We now present several Procera controllers, beginning with simple ones that explain the syntax. We then proceed to describe more complex examples that demonstrate the expressive power of the language.

Example: Static Policy. Our first example is the "hello world" of network access control examples. It simply allows all network traffic flows.

```
proc world → do
  returnA −≺ λreq → allow
```

Here we simply ignore the *world* signal and always output the same flow constraint function (the expression $\lambda req \to allow$ is a lambda-expression, i.e. an anonymous function, with parameter *req*). The flow constraint function itself ignores its argument and simply allows all requests. A variant of this example is one that denies all flow requests:

```
proc world → do
  returnA −≺ λreq → deny
```

Many networks use static access control policies which limit access to some devices or services on the network. For example, a policy might allow only http traffic to IP addresses in subnet `128.36.5.0 / 24`:

```
proc world → do
  returnA −≺
    λreq → if destIP req ‘inSubnet‘ ipAddr 128 36 5 0 // 24
           then allow else deny
```

Many more static policies, for example accessing other packet fields, can be written in Procera. We omit the details here due to space limitations.

Example: Device registration. In many home and campus networks, network access is limited to some set of registered or authenticated devices, and in many networks, the set of authenticated devices varies over time, as a result both of administrative actions (e.g., through user interface actions or configuration file edits), or through user actions (e.g., authenticating with a portal).

To express policy related to administrative device authentication events, we add these events to the *World* data type and add functions *authEvents* and *deAuthEvents* to refer to authentication events. Then we can, for example, define a set whose members at any time are exactly those devices which have a more recent authentication than deauthentication, and use that in our policy function:

```
proc world → do
  authDevs ← accumSet −≺ add (authEvents world) ⊕
                          remove (deAuthEvents world)
  returnA −≺
    λreq → if srcEthAddr req ‘member‘ authDevs
           then allow else deny
```

We feed a stream of *AddRemoveEvent* values into *accumSet* which accumulates the set arising from the addition or removal of values. We create the stream of *AddRemoveEvent*s using *add*, *remove*, and $\oplus$ functions. The policy function varies over time, because it refers to the time-varying set of authenticated devices.

Example: Device Usage Caps. We now consider policies inspired by the uCap system [15], which allows home network administrators to limit the network access to devices and users based on maximum bandwidth usage settings, or caps. In uCap, users configure monthly usage caps via a user interface and the system collects device usage logs from routers every few seconds. A (simplified) uCap controller ensures that any device whose usage for the current month exceeds its monthly cap is barred from using the network until the following month.

To express this logic, we add events to our *World* data type for cap settings and usage reports. These events are denoted by *usageEvents world* and *capSetEvents world*, respectively. We can then track the cap settings by keeping track of the last setting per device, using the *last1PerGroup* signal function, which outputs a dictionary mapping keys to the value of the last event for that key:

```
capTableSF =
  proc world → do
    last1PerGroup ─≺ add (capSetEvents world)
```

(The arrows notation allows the last statement to be either **returnA** as in earlier examples, or a signal function, as in this example.) The controller can then track the monthly usage by using the *clockResetWindow nextMonth* signal function, which accumulates events until the next month, and then groups by the key and sums values for the same key:

```
useTableSF =
  proc world → do
    recent ← clockResetWindow nextMonth
               ─≺ add (usageEvents world)
    group sum ─≺ recent
```

nextMonth, whose straightforward definition we omit here, is a function which maps a given date into the beginning of the month following the date. Other functions could be used here to reset usage in other ways, for example by week or by time of day.

We can then define our overall uCap controller by using the above signal functions (we omit the straightforward definitions of *lookupUse* and *lookupCap*):

```
proc world → do
  useTable ← useTableSF ─≺ world
  capTable ← capTableSF ─≺ world
  let policy req =
    let src = srcEthAddr req
        use = lookupUse src useTable
        cap = lookupCap src capTable
    in if use < cap then allow else deny
  returnA ─≺ policy
```

Example: Per-User Usage Caps. A variant of the above controller keeps track of usage by user rather than by device. Procera can also accomodate this policy, despite the input signals only including usage by device. The policy correlates the usage events with the authentication table, by associating the device of each usage event with the user id given in the authentication table. The following example demonstrates how this can be done (we assume that *authTableSF*, which provides the authentication table, has been defined previously):

```
useTableByUserSF =
  proc world → do
    authTable ← authTableSF ─≺ world
    let evs = [(lookupUser ddev authTable, use)
              | (dev, use) ← usageEvents world]
    recent ← clockResetWindow nextMonth ─≺ add evs
    group sum ─≺ recent
```

In this example, the event comprehension attaches the usage amount of each usage event to the user who is currently authenticated at the device at the time of the event. These events are then windowed and grouped as in the previous example.

Example: Usage Caps on a Campus Network. We return to the policy mentioned in Section 3 and which is inspired by the publicly available campus network policies from Carnegie Mellon [1]. A simplified version of the policy states that a device is permanently barred from the network if it exceeds a five-day sliding cap, until an adminstrator reinstates the device. This policy differs from the previous example in two ways: (1) usage is aggregated over a sliding window and (2) devices can be explicitly reinstated. To express this policy, we add another event source to the *World* data type for administrative resets, letting *adminResets world* denote these events. We then express the policy in two steps, first writing the signal function to track usage over the sliding window and second writing the signal function which tracks users that have exceeded their cap and have not yet been reinstated:

```
useTableSF = proc world → do
  recent ← since (daysAgo 5) ─≺ add (usageEvents world)
  group sum ─≺ recent
overSetSF =
  proc world → do
    useTable ← useTableSF ─≺ world
    capTable ← capTableSF ─≺ world
    let devEv = usageEvents world .|. capSetEvents world
    accumSet ─≺
      add [dev | dev ← devEv,
                 let cap = lookupCap dev capTable,
                 let use = lookupUse dev useTable,
                 use > cap]
       ⊙ remove (adminResets world)
```

In *overSetSF* we use the .|. operation, which merges two events in order to let *devEv* stand for any event changing the usage or cap setting of a device. We then use an event comprehension to check whether the usage for any device for which an event occurred now exceeds its usage cap. We omit the policy that simply checks that the sender of a packet is not a member of set provided by *overSetSF*.

6. RELATED WORK

We discuss related work in terms of the distinctions made in our system architecture in Figure 1. There are many OpenFlow controller frameworks available, such as NOX [11], Maestro [27] and Floodlight [2], and POX [3]. These frameworks provide low-level control over switch flow tables and are typically imperative and object-oriented. Nettle [24,25] is also a network controller, but differs from the above systems by allowing the low-level control programs to be written in a domain specific language based on FRP and embedded in a functional programming language.

Flow-based Management Language (FML) [13] is a policy language for the NOX controller that allows operators to declaratively specify network policies and therefore serves a similar purpose as Procera. We compare Procera with FML in detail in Section 3.

Frenetic [10] is a network programming language built on NOX; it has two sub-languages: (1) a declarative network query language, and (2) a functional and reactive network policy management library based on FRP. Frenetic occupies a role somewhere between a policy layer and a network controller: it deals with network details such as routing and switch rules, but also introduces an abstraction that allows multiple network management programs to compose without interfering. Frenetic and Procera have some similar language constructs, such as time windowing; While Frenetic applies these operations only to packets, Procera allows these to be applied to any event stream.

Declarative routing [18] is recursive query language for network routing. It is designed to simplify distributed router programming, rather than logically centralized controllers.

7. SUMMARY

We have introduced Procera, a language that allows network operators to specify high-level, reactive network control policies that cannot easily be expressed in today's languages for software-defined network control. Procera applies the principles of functional reactive programming to provide a declarative, expressive, and compositional framework that allows operators to express network policies based on both reactive and temporal behaviors, which are typically necessary to express common, simple network policies. In our ongoing work, we are implementing a scalable Procera controller and exploring how the composability that FRP facilitates might make it easier to compose orthogonal network policies (e.g., traffic load balance and access control) in a common policy control framework. We are also deploying Procera on BISmark home routers; we plan to use it to enable home network users to express policies such those we have described in this paper, either directly, or indirectly via a graphical user interface.

Acknowledgments

This work was supported in part by a gift from Microsoft Research, NSF Awards CNS-1018021, CNS-1040715, and a grant from the GENI Project Office, "GENI OpenFlow Campus Buildout".

REFERENCES

[1] CMU network bandwidth usage guildline. `http://www.cmu.edu/computing/network/connect/guidelines/bandwidth.html`.

[2] Floodlight, Java-based OpenFlow Controller. `http://floodlight.openflowhub.org/`.

[3] POX, Python-based OpenFlow Controller. `http://www.noxrepo.org/pox/about-pox/`.

[4] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[5] H. Ballani and P. Francis. Conman: a step towards network manageability. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 205–216, New York, NY, USA, 2007. ACM.

[6] T. Benson, A. Akella, and A. Shaikh. Demystifying configuration challenges and trade-offs in network-based isp services. *SIGCOMM Comput. Commun. Rev.*, 41(4):302–313, Aug. 2011.

[7] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 6:1–6:12, New York, NY, USA, 2010. ACM.

[8] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM.

[9] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proc. 2nd Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005.

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

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

[12] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-serve: Load-balancing web traffic using openflow. Sigcomm Demonstration, 2009.

[13] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, pages 1–10, New York, NY, USA, 2009. ACM.

[14] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: a tale of two campuses. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 499–514, New York, NY, USA, 2011. ACM.

[15] H. Kim, S. Sundaresan, M. Chetty, N. Feamster, and W. K. Edwards. Communicating with caps: managing usage caps in home networks. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, SIGCOMM '11, pages 470–471, New York, NY, USA, 2011. ACM.

[16] H. Kim, A. Voellmy, S. Burnett, N. Feamster, and R. Clark. Lithium: Event-driven network control. Technical Report GT-CS-12-03, Georgia Institute of Technology, 2012.

[17] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing, Vol. 4, No. 1*, pages 67–95, Feburary 1986.

[18] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 289–300, New York, NY, USA, 2005. ACM.

[19] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, pages 3–17, Pittsburgh, PA, Aug. 2002.

[20] J. McCarthy. Situations, actions, and causal laws. *Stanford University Artificial Intelligence Project, Stanford University; reprinted in M. Minsky (ed.), Semantic Information Processing, MIT Press*, pages 410–417, 1968.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communications Review*, Apr. 2008.

[22] R. Paterson. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 229–240, New York, NY, USA, 2001. ACM.

[23] V. Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[24] R. Rocha and J. Launchbury, editors. *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539 of *Lecture Notes in Computer Science*. Springer, 2011.

[25] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In Rocha and Launchbury [24], pages 235–249.

[26] Z. Wan and P. Hudak. Functional Reactive Programming from first principles. In *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[27] T. S. E. N. Zheng Cai, Alan L. Cox. Maestro: A System for Scalable OpenFlow Control, December 2010. `http://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf`.