Composing Software Defined Networks

Joshua Reich[*]*, Christopher Monsanto*[*]*, Nate Foster*[†]*, Jennifer Rexford*[*]*, David Walker*[*]
[*]Princeton [†]Cornell

Abstract

In Software Defined Networking (SDN), an application comprising many disparate tasks must be converted to a single set of packet-processing rules on the switches. Unfortunately, today's SDN platforms do not support expressing these tasks as separate modules, and composing them to create an application. This leads to monolithic programs that are neither portable, nor reusable. In this paper, we present the FV system that presents each module with an abstract *view* of the network topology customized to the application logic, where one module may implement the "switching fabric" for another. For example, a firewall module may run on "one big switch" that is implemented by a routing module. The programmer can specify network views, as well as the relationship between (virtual) switches in different views. For example, conceptually the firewall functionality runs before the routing functionality. Using *sequential composition*, the FV compiler can synthesize a single set of rules and queries for each physical switch. FV includes a *core* language for defining policies as mathematical functions in an imperative style familiar to Python programmers, as well as a *module* language that supports abstraction (*i.e.,* network views) and protection (*i.e.,* specifying what traffic a module can measure and control). FV enables the creation of sophisticated SDN applications, as illustrated by example programs running on our FV prototype.

1 Introduction

Network management is a difficult task. Operators must configure a array of services executing simultaneously on multiple devices, from routing and traffic monitoring, to access control and server load balancing. Software-defined networking (SDN) provides programmers with the basic *controls* for addressing these problems, but it does not provide the *abstractions* that are needed to manage the accompanying complexity. Programmers must reason manually, in unstructured, ad hoc ways about low-level dependencies between code in different modules, as well as between code and the physical network. The result is applications that are neither portable nor reusable.

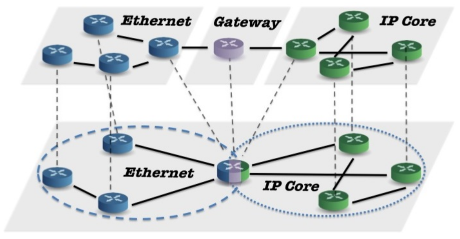
Modularity is essential for managing complexity in any software systems, and SDNs are no exception. The unique challenge in SDNs is that the modules that make up an application ultimately execute on the same devices—*i.e.,* they process the same traffic using the same flow table entries on the same switches. Although some SDN platforms support abstractions for "slicing" the network (to allow different applications to handle different portions of the traffic) [21, 7], "virtualizing" the network (to decouple programs from the underlying topology) [15, 19], or "composing" monitoring queries with a forwarding policy [4], these abstractions do not effectively address the question of how to build a *single application* out of multiple, separately defined and reuseable modules that affect the handling of the *same traffic*.

This paper presents FV, a language that provides a collection of flexible programming constructs for supporting "programming in the large." These constructs make it possible to build sophisticated controller applications by composing simpler modules together. Each module has the illusion of running over a network topology tailored to that module—a *network view*. A load balancer or firewall module may see the network as "one big switch" [2, 15, 9], while a learning switch module may see the links between switches on which it operates.

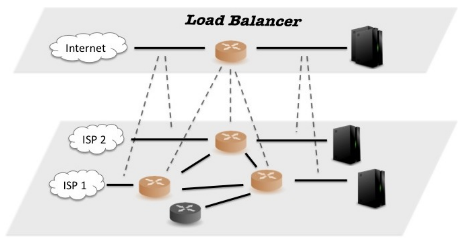
1.1 Constructing Modular SDN Applications

To highlight the challenges of building modular SDN applications, consider the following enterprise scenarios.

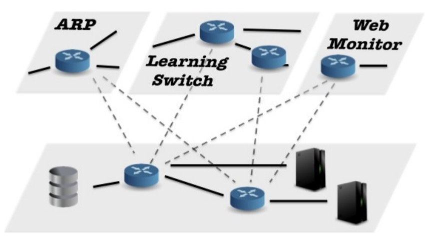
Combining Ethernet, IP, and gateway routing: Enterprise networks often consist of several Ethernet islands interconnected by gateway routers to an IP core, as shown in Figure 1(a). To implement this behavior, an SDN programmer would have to write a single, monolithic program that handles network events differently depending on the role the switch is playing in the network. This program would implement MAC learning



(a) Combining Ethernet, a gateway, and an IP core



(b) Combining server load balancing and routing



(c) Combining Ethernet routing, ARP, and Web monitoring

Figure 1: Example modular SDN applications.

and flooding to unknown destinations for switches within Ethernet islands, shortest-path routing on IP prefixes for switches in the IP core, and gateway logic for devices connecting an island to the core. Note that the gateway logic would be complicated, as the switch would need to act simultaneously as a switch, router, and bridge.

A better alternative would be to implement the Ethernet islands, IP core, and gateway routers using separate modules operating on a subset of the topology, as shown in Figure 1(a). This design would allow the gateway router to be decomposed into three virtual devices: one in the Ethernet island, another in the IP core, and a third interconnecting the other two. Likewise, its logic could be decomposed into three orthogonal pieces: a repeater that responds to ARP queries for its gateway address, a standard Ethernet switch, and a standard IP router. The programmer would write these modules separately and the controller would compose them into a single program that installs rules on the underlying physical switches.

Combining routing and middleboxes: Enterprises often run middleboxes such as firewalls or server load balancers. Ideally operators would be able to distribute middlebox functionality across a collection of commodity switches instead of using expensive appliances, as shown in Figure 1(b). To achieve this today, a programmer would have to write an application that simultaneously implements the load balancer and routing logic, carefully constructing forwarding rules that pick a server replica and also forward packets to that destination.

As above, a better alternative would be to decompose this functionality into separate modules. One module would implement the load balancer against a network view containing of one switch with two ports—one connected to the Internet and another to the internal hosts, as shown in Figure 1(b). A second module would route traffic across the underlying physical topology, constructing the "switching fabric" for the virtual switch configured by the load balancer. This design would allow the programmer to replace the load balancer application—*e.g.,* with a firewall—without having to change the underlying forwarding logic, and vice versa. Again, the controller would combine these modules into a program that computes the rules for the underlying switches.

Combining routing, broadcast, and monitoring: Enterprise networks often handle discovery and ordinary traffic, as shown in Figure 1(c). Today, a programmer would need to write a single program that handles both classes of traffic. This program would "bake in" a single way of handling discovery, and would install special rules for handling broadcast packets (to avoid performing MAC learning). Ideally the programmer would be able to write two modules—one for discovery and another for ordinary traffic. The discovery module would flood queries and replies while the other module would perform MAC learning. Later, the discovery module could be replaced with one implementing a directory mapping between MAC and IP addresses as in VL2 [5], without having to change the other module.

Continuing the example, suppose the programmer needed to extend the application to monitor the total amount of Web traffic. If the application were written as a single monolithic program, they would need to modify the program to combine the monitoring and forwarding logic to produce a single set of rules that simultaneously *count* the port-80 traffic and *forward* traffic based on the destination, even though each packet can match at most one rule. Doing this would entail manually constructing the "cross-product" of the two policies, and query and sum the relevant traffic counters to monitor the Web traffic [4]. A better alternative would be to write a single new module that monitors Web traffic, and compose it with the forwarding modules.

1.2 Language Support for Policy and Composition

To effectively build sophisticated SDN applications, programmers need abstractions for developing individual modules and for specifying how those modules should be combined together into a single program. One approach, which is used in most existing SDN controller platforms [6, 1, 22, 3], is to offer programmers the same abstractions for constructing modules as are used for interacting with physical switches—*i.e.,* the OpenFlow

API. But this approach means that programs that abstract away from the underlying topology must handle an interface that is both “wide” (with many different network events), and “flat” (with few higher-level constructs). These programs must invoke network event handlers, transform virtual forwarding tables to physical ones, and maintain virtual network state, such as counters. Doing this is not impossible, but it requires a huge effort. Moreover, in practice, virtual switches are typically implemented using hypervisors, which makes certain forms of composition such as deeply nested virtualization impractical [2]. Finally, if one is virtualizing the OpenFlow API, one might want or need to build any higher-level abstractions multiple times—once *in* the hypervisor and once *on top of* the hypervisor.

This paper describes a different approach. We present a language for programming SDNs called FV, that includes a collection of powerful new constructs that make it easy to express many different patterns of program composition. Using FV, programmers can write the code for a single module using high-level abstractions instead of manipulating low-level forwarding table configurations. They can also define network views that abstract away from the underlying physical topology and write programs against these views. To construct a view, the programmer defines a topology of virtual switches, and a mapping from virtual ports to ports in an underlying network (which may itself consist of a mix of virtual and physical elements). The programmer also identifies the traffic that can enter and leave the virtual network at each edge port (*i.e.,* , a subset of the “flowspace”). The implementation of each virtual switch—that is, its “switching fabric”—is provided by one or more other modules. Finally, they can combine modules written against these views using simple operators that support either sharing or isolating the underlying network.

FV is based on our earlier work on the Frenetic language [4, 12], but extends it with a generalized packet model, stack-based operations for manipulating packet contents, and an explicit sequential composition operator. Thse constructs make it easy to express complicated packet-processing pipelines naturaly, as the composition of several phases without having to manually stitch those phases together. In addition, abstractions traditionally provided by hypervisors such as slicing and topology virtualization become completely trivial to implement. Moreover, because they are expressed at the language-level, these constructs can be freely combined and are not limited to a single level of nesting.

The paper makes the following research contributions:

- We demonstrate how to create modular SDN applications using network views;
- We extend the Frenetic language with a generalized packet model, sequential composition operators, and familiar imperative control operators (Section 2);
- We present constructs for creating network views and combining modules, and describe compilation techniques for implementing these features on physical switches (Section 3); and
- We describe a prototype implementation and evaluation of FV on a case study (Section 4);

The paper ends with a discussion of related work in Section 5, and a conclusion in Section 6.

2 Core FV

FV is an extension and redesign of the Frenetic language [4]. Like Frenetic, FV contains a high-level language for specifying static policies, which may be thought of as “snapshots” of a network’s global forwarding behavior. Indeed, many of the elements of FV, especially parallel policy composition, are inspired directly from Frenetic, but other elements, such as the sequential (functional) policy composition are new. The latter comes from a new perspective on the basic interpretation of policies as *abstract mathematical functions*. As the examples in this section and the next show, these operators facilitate decomposition of complex network management tasks into simpler parts, and are also critical infrastructure for defining network views.

Like Frenetic, FV programs handle dynamic behavior by generating a series of static policies, which are processed by the underlying run-time system and installed in the network. In Frenetic, such a series was generated using functional reactive programming, while in FV, we adopt an imperative style in which programmers may install an initial policy and imperatively update it over time. We believe this imperative style will be more familiar to typical systems programmers. On the other hand, this imperative style does not lend itself directly to good composition properties as two modules separately updating the global network policy are likely to clobber each other. Hence, without further support (in the form of network views) this new programming framework, while enjoying a familiar “look and feel”, is structurally inferior. In Section 3 we show how to define and create network views, how to confine a series of imperative policy updates to a single view, and how to manage the interactions between views so that views defined separately can productively interact. The rest of this section, explains the core FV programming infrastructure.

2.1 Static Policies

In conventional SDN platforms, forwarding policies are usually specified using lists of pattern-action rules, with one list for each switch. When a packet arrives at a switch, the switch performs the action associated with the first (*i.e.,* highest priority) matching pattern in the list. Representing policies using lists of rules makes implementations easy, as there is a one-to-one correspondence between the rules defined by the programmer and the forwarding table entries on each switch. But on the other hand, they make it difficult to construct sophisticated policies out of simple building blocks. If list L_1 defines a set of actions A_1 for a given packet and list L_2 defines a different set of actions A_2, then the list obtained by concatenating L_1 with L_2 will not necessarily define the *union* of actions A_1 and A_2, which is often what the programmer wants. In particular, concatenation can have a variety of unexpected effects depending on how the rule priorities and patterns interact. Hence, instead of depending upon prioritized lists of simple pattern-action rules, FV defines a higher-level language of static policies aimed at making it easy for programmers to describe the forwarding behavior of the network in a compositional way.

FV programmers are encouraged to think of every FV policy as a function from *located packets* (packets annotated with the current switch and ingress port) to *sets of located packets* and to ignore how those functions are implemented by the underlying run-time system. By interpreting policies as mathematical functions, we lift the level of abstraction at which programmers operate, enable several additional forms of composition, which fall out as an almost inevitable consequence of our design.

Primitive actions. The simplest FV policy is one that does nothing but apply a basic *action* to a located packet, yielding a new set of located packets as a result. For example, the `fwd(port)` action, when interpreted as a function, receives any located packet as an argument and produces the singleton set containing the same packet relocated to port `port` as a result. The `drop` action, on the other hand, receives any located packet as an argument and produces the empty set of packets. Intuitively, a policy that generates the empty set has dropped its input. The `passthrough` action accepts any packet and returns the singleton set containing that packet. The `all` action receives any packet located at port p on switch s and produces an output set containing the same packet at each output port on s except p. Finally, the action `modify(h=v)` receives a packet as an input and produces the singleton set containing the same packet—at the same location (!)—except that header `h` is set to `v`. In summary, some actions produce one result (forwarding, modification), some actions produce no results (dropping) and some actions produce many results (flooding).

Functional composition. On its own, the ability to modify a packet while leaving it in place is not useful—one only cares about the modified packet if it is ultimately forwarded to some destination. To couple packet modification with transport, we use *functional composition*. Typically, when given two functions, f and g, their composition $f \circ g$ is defined as the function h such that $h(x) = g(f(x))$. Since our policies are interpreted as functions, we use a similar notion. The only difference is that since our policies are functions from packets to sets of packets, when given two policies `C1` and `C2`, we define their composition `C1 >> C2` as the function `C3` such that:

```
C3(packet) = C2(p1) U ... U C2(pn)
  when {p1,...,pn} = C1(packet)
```

In other words, we apply `C1` to the input, generating a set of packets (`p1`,..., `pn`) and then apply `C2` to each of those results, taking their union as the final result. As an example, consider the following policy, which modifies the VLAN tag of any incoming packet to 1 and then forwards the modified packet out port 3.

```
modify(vlan=1) >> fwd(3)
```

As a more elaborate example, consider a complex policy `P2`, designed for forwarding traffic with a variety of different VLAN tags (tags 1, 2, 3, *etc.*) across a network. Now, suppose we would like to apply the policy for tag 1 to a particular subset of the traffic arriving on network. To do so, we may write a policy `P1` to select and tag the relevant traffic. To use `P1` and `P2` in combination, we exploit functional composition: `P1 >> P2`. Such a program is quite modular: if a programmer wanted to change the forwarding component, she would change `P2`, while if she wanted to change the set of packets admitted to the VLAN-1 network, she would change `P1`. In the next section, we will see how to use this paradigm to structure the construction of general-purpose virtual networks.

Policy restriction and parallel composition. Actions and compositions of actions are universal policies: they apply to all packets. To restrict the basic packet-forwarding functions defined by actions, FV supports policy restriction, `&` C, where P is a predicate over packets and C is another policy. Interpreted as function, P `&` C applies the function C to the input packet if it satisfies P, and otherwise returns the empty set.

Predicates include `all_packets` and `no_packets`, which match all or no packets respectively, conjunction (`&`), disjunction (`|`), and negation (`~`). The predicate `match(h=v)` tests whether header `h` matches the basic value `v`. As an example, consider the policy `P3` below, which forwards packets on port 1 with destination IP matching `1.1.1.*` (multiple arguments to match implicitly conjoin the matches).

```
match(inport=1, dstip="1.1.1.*") & fwd(2)
```

The policy `P4`, below, forwards packets on port 1 with destination IP matching `1.1.2.*` to port 5.

```
match(inport=1, dstip="1.1.2.*") & fwd(5)
```

To compose policies P3 and P4, FV includes a *parallel composition* operator: P3 | P4 behaves as if P3 and P4 were executed on every packet simultaneously. In other words, given an input packet p, P3 | P4(p) returns the set of packets S1 U S2 when P3(p) returns S1 and P4(p) returns S2.

Continuing our example, if a programmer wanted to apply the policy P3 | P4 to particular switch s1 and a different policy P6 to switch s2, she could construct the following composite policy P7.

```
P5 = P3 | P4
P6 = ...
P7 = match(switch=s1) & P5
   | match(switch=s2) & P6
```

After recognizing a security threat from source IP address, say address 1.2.3.4, the programmer might go one step further creating policy P8.

```
P8 = ~match(srcip="1.2.3.4") & P7
```

An equivalent, but often convenient alternative to such the negations is a policy constructed using subtraction to quotient out unwanted forwarding behavior:

```
P8' = P5 - match(srcip="1.2.3.4")
```

We will sometimes use the policy if_-, which works as a conditional: if the current packet has srcip="1.1.1.1", then if_(match(srcip="1.1.*.*"), drop, passthrough) drops the packet. Conditional policies can be encoded using parallel composition, restriction, and negation.

A generalized packet model. So far, we have presented the packets processed by our static policies as records that maps standard header fields (*e.g.,* , source IP, destination, IP, etc.) to values. However, in order to use these static policies in a context involving multi-level abstract networks, we generalize this packet model in two ways.

First, we allow programs to define and use *virtual fields*. For example, in the next section, we will explain how our compiler uses virtual fields vinport, voutport, and vswitch. Policies can be written against such fields as though they are ordinary fields such as source IP or destination MAC. The compiler is responsible for compiling virtual, extended packets into real packets that represent the same information. In our case, the compiler could use bits of the VLAN field to encode the additional headers. Other mechanisms such as MPLS labels could also be used. In any event, the programmer need not concern themselves with how the encoding is achieved—he or she is free to define policies over these high-level, abstract packets.

Second, each field of a packet may carry a stack of values. To support this model, FV includes two additional primitive actions: push and pop. For example, consider a packet A defined as follows.

```
{switch: p2, inport: 3, ... }
```

The physical switch is p2 and physical ingress port is 3. Applying push(switch=v1) would yield the following packet B:

```
{switch: [v1, p2], inport: 3, ... }
```

The switch field is now a stack with virtual switch v1 on top and physical switch p2 underneath. When interpreting predicates such as match(switch=p2), only the topmost value on the stack is used. Hence, match(switch=p2) is true when interpreted over packet A and false when interpreted over packet B. As we will see in the next section, our compiler pushes new virtual identifiers on these stacks to "lift" a packet up on to a virtual switch and to interpret the static policy for that virtual switch over the packet. As a special case, matching a header against the value None tests if the stack for the header in question is empty. Note that, like match, we allow multiple arguments to push and pop which simply mean to push (or pop) more than one value at a time from the current packet.

As an aside, the primitive modify action may be thought of as a pop followed by a push of the same value. Another other action is the move action, which pops a value off of one field and pushes it on to another field. Our compiler uses move to move a value from the virtual switch field vswitch to the real switch field switch to create the illusion that a policy defined over a high-level virtual switch is running on the bare hardware. Using this mechanism, high-level policies cannot distinguish between execution on a virtual network and execution on a physical network.

Summary. So far, we have defined a static policy language over a generalized packet model whose primary components are: (1) primitive actions such as passthrough, fwd, drop, all, modify, push, pop, and move that each define packet transformaton functions, (2) a restriction operation (&) that limits the set of packets to which a policy applies, (3) a parallel composition operation (|) that forms the union of two policies, (4) functional composition (>>), and (5) a conditional (if_) that selects a policy depending of whether a predicate holds or not. These operators are completely general and may be applied in any order or nested to arbitrary depth; the underlying run-time system is responsible for analyzing and deploying the policy as a prioritized list of rules implementable on conventional SDN hardware. Figure 2 summarizes the syntax of the static policy definition language.

2.2 Using static policies

Once the programmer has defined a static policy, they may use it by installing it on a network view. Figure 3 presents a complete FV program. The function hub takes a network view as an argument and defines a policy

Predicates:

P ::= `all_packets` | `no_packets` | `match(h=v)` | P & P | (P | P) | ~P

Actions:

A ::= `drop` | `passthrough` | `fwd(loc)` | `modify(h=v)` | `all` | `push(h=v)` | `pop(h=v)` | `move(h1=h2)`

Policies:

C ::= A | `if_`(P,C,C) | (C | C) | P & C | C - P | C >> C

Figure 2: Summary of static policy syntax

```
def hub(net):
  pol = all                # define pol
  net.install_policy(pol) # install it
```

Figure 3: A complete program: `hub.py`.

that floods incoming packets and then installs that policy on the network. The next section describes how to define and manage custom network views.

2.3 Crafting dynamic policies

The hub in Figure 3 implements an unchanging, static policy, but many SDN policies are dynamic—they react to network events such as switches coming up, new flows arriving, or changes in network load. A dynamic FV policy is really just a Python program that calls `net.install_policy` multiple times, thereby installing a series of static policies on switches in sequence. To further help users build dynamic policies, we have developed some additional libraries such as the bucket library, which is described next.

Buckets. Many SDN applications require a means to monitor and react to network traffic. The *bucket* is a simple FV abstraction for doing so. Abstractly, a bucket is simply an ordered list of packets. Underneath the covers, the run-time system takes care of the details involved in implementing this abstraction: it installs rules to divert packets to the controller, polls statistics on switches when needed, and represents the information content of buckets efficiently.

Buckets may be created and then referenced inside policies. If `b` is a bucket, the policy `fwd(b)` sends the relevant packets to the bucket `b`, as opposed to some specified port. Buckets support the Python iterator protocol, so a programmer may use standard constructs such as `for` loops to process the packets that appear in a bucket. Figure 4 presents a deep packet inspection function based on buckets. The policy does nothing but draw packets out of the network. If desired, the policy could have both forwarded packets with source IP `1.2.3.4`

```
def dpi(net):
  b = bucket()
  pol = match(srcip="1.2.3.4") & fwd(b)
  net.install_policy(pol)
  for pkt in b:
    print "I see packet: ", pkt
```

Figure 4: Deep packet inspection.

```
def learn(net):
  pol = all
  net.install_policy(pol)
  q = query_unique(net, all_packets,
        fields=["srcmac", "inport"])
  for pkt in q:
    pred = match(dstmac=pkt.srcmac)
    pol -= pred                    # (A)
    pol |= pred & fwd(pkt.inport) # (B)
    net.install_policy(pol)
```

Figure 5: Ethernet learning switch.

to the bucket `b`, and flooded all packets (including those with source IP `1.2.3.4`) using parallel composition:

```
pol = match(srcip="1.2.3.4") & fwd(b) | all
```

Because the pattern of allocating a bucket, defining a policy to forward packets matching a predicate (`pred`) to the bucket, and installing that policy on a particular network (`net`) is a common idiom, FV packages these operations together into queries. The expression `query(net, pred)` generates the stream of packets that results. In addition, instead of injecting all packets that match the query into the stream, one may choose to inject just those packets with unique values in a particular set of headers. For example,

```
query_unique(net, all_packets,
  fields=["srcmac", "inport"])
```

generates a stream of packets, with one packet per source MAC-ingress port pair.

Putting it together: Ethernet learning switch Figure 5 presents a simple learning switch that illustrate the pieces of FV discussed so far. Initially, the `learn` function installs a default policy that floods. Next, it defines a query `q` that generates a stream of packets with unique source MAC and inport.[1] For each packet `pkt` in the stream, the code creates a predicate that tests destination MAC addresses against `pkt`'s source MAC. The code the constructs a new policy that (a) stops flooding when it sees that destination MAC address, and (b) forwards packets out `pkt`'s inport when it does see that destination MAC address. This new policy is installed, updating the old one. Readers familiar with NOX will appreciate how much simpler, more abstract, and more direct this code is than the corresponding NOX program [6].

[1]For the purposes of this example, we assume that hosts do not move so each source MAC will be associated with at most one inport.

3 Composing Networks

The central goal of the FV language is to provide programmers with a means to craft modular network management applications. The core language described in the previous is an important stepping stone towards this goal as it provides programmers with several flexible and uniform ways to construct complex policies from simple, independent pieces. However, the language is missing two key features: abstraction (*i.e.,* information hiding) and protection (*i.e.,* control over which parts of a policy influence which traffic). In this section, we show how to wrap the FV core in a module system based around the idea of defining virtual network views. These network views support abstraction of physical networks and they also allow programmers ensure that separately-defined views interact in a controlled fashion. In the following subsections, we (a) describe the interaction mechanisms FV provides for working with network views, (b) show how network views are defined and sketch how they are compiled back in to the FV core language so they can be processed by our run-time system and loaded on to switches, and (c) demonstrate how the process of view creation can be automated.

3.1 Network interactions

When decomposing a complex network management problem in to a set of tasks, the programmer must be mindful of how those tasks interact. As a initial example, consider two separately-defined network management functions `A()` and `B()`, which both install network policies.

```
def A(net):
  net.install_policy(polA)

def B(net):
  net.install_policy(polB)

def both(net):
  A(net); B(net)
```

Recall that the semantics of `net.install_policy` is to completely replace the current network policy with a new one. Such a semantics, when coupled with an implementation that provides per-packet consistent update, makes it easy to reason about network forwarding behavior and easy to support powerful verification tools [18]. Yet the downside is that `A` and `B` above do not collaborate: `A` installs its policy globally across `net`, and then `B` installs its policy, completely clobbering what `A` had just done. Consequently, we must define new mechanisms that allow `A` and `B` to be defined separately and yet also interact productively to solve a complex network management problem. To do so, FV admits three non-trivial interaction modes: *sharing*, *isolation* and *abstraction*.

Sharing. When two collaborators *share* an underlying network, they can both see and control all traffic that runs over that network. A canonical use-case for shared networks occurs when one of or more of the collaborators "reads", monitoring network traffic, while the other "writes", controlling packet forwarding—with only one writer, there are no write-conflicts to resolve. For example, the hub, defined in Figure 3, might share a network with the `dpi` function in Figure 4. Such sharing can be orchestrated as follows.

```
def dpi_and_hub(net):
  net1 = share(net)
  net2 = share(net)
  run(hub, net1)
  run(dpi, net2)
```

Above, we first generate the shared sub-networks `net1` and `net2` and then we run the `hub` and `dpi` in separate threads on the distinct but shared networks. Without sharing, either the `hub`'s policy (`all`) would be installed and no DPI monitoring would occur, or the `dpi` policy (`fwd(b)`) and packets would be forwarded to the bucket but not to their proper recipients. Using the sharing interaction mode, *both* policies are installed simultaneously. This is achieved inside our run-time system by listening for policy changes on either shared network and then merging those changes. In effect, the global policy installed on the network is as follows.

```
all | (match(srcip="1.2.3.4") & fwd(b))
```

Isolation. To contrast the sharing interaction mode, consider the case for isolation. In the following code fragment, we define two simple forwarding policies: one for registered traffic (traffic that satisfies the predicate `reg_pred`) and one for unregistered traffic (traffic that satisfies the disjoint predicate `unreg_pred`).

```
reg_pol =
    match(switch=1) & reg_pred & fwd(2)
  | match(switch=2) & fwd(3)

unreg_pol =
    match(switch=1) & unreg_pred & fwd(2)
  | match(switch=2) & web_pred & fwd(3)
```

In the first policy, all registered traffic is forwarded out port 2 (connected to switch 2) and then out of port 3; in the second policy all unregistered traffic is forwarded to switch 2, but only the web traffic is allowed to proceed. If these two policies are registered simultaneously on a shared network then unregistered traffic will be forwarded to switch 2 via `unreg_pol`, and once there will be forwarded via `reg_pol` out of port 3. In other words, `reg_pol` unintentionally acts to defeat the security measures put in place by `unreg_pol`. To ensure traffic does not ping-pong back and forth between polices, programmers may use isolation.

```
def reg_traffic(net):
  net.install_policy(reg_pol)

def unreg_traffic(net):
  net.install_policy(unreg_pol)
```

```
def reg_and_unreg_traffic(net):
  run(unreg_traffic, isolate(net1))
  run(reg_traffic, isolate(net2))
```

All traffic about to enter an isolated network is logically copied on to that network at ingress points, processed by the rules of the network and then copied off the network at egress points. (We show how to define ingresses and egresses in the next section.) Hence, logically, all traffic at the entry point on switch 1 is copied on to both networks. The unregistered policy does not specify an action for the registered traffic so that traffic is dropped off the isolated unregistered network. Conversely, the unregistered traffic is dropped off the registered network. Hence, combined, the two policies collaborate to forward both kinds of traffic properly. However, the policies for both are defined independently (perhaps by different programmers); each policy may be edited or altered independently of the other.

Abstraction. One flaw in the above program design is that the programmer responsible for crafting `reg_pol` must be careful not to accidentally process traffic at the ingress of their network that should be handled by the unregistered network. A second flaw of the above design is that this portion of network policy is designed to do nothing more than establish a differing firewalls for registered and unregistered traffic. Doing so does not require knowledge of the detailed topology. Instead, the policies may be implemented over simpler "one-big-switch" abstractions of the underlying network.

FV can help programmers rectify both design flaws by allowing them to define *abstractions* or *transformations* of the underlying network. These transformations can refine the traffic admitted to the network, refine the traffic emitted from the network, hide the existence of physical devices and change the apparent topology of the network. To generate a new abstract network for customization on top of `net`, the programmer simply calls `abstract(net)`; the next section will explain how to configure such abstract networks. In terms of protections, `abstract()` is identical to `share()`.

Putting it all together The third scenario discussed in Section 1.1 presents a chance to demonstrate how these concepts fit together. Both the address resolution and web monitoring programs are only concerned with links attached to end-hosts, while the Ethernet code must consider how internal links are used as well. Consequently, both of the former programs will operate on abstracted views, while the latter will run on an unabstracted view in this example. Figure 6 graphically outlines the sequence of *isolate*, *share*, and *abstract* calls required.

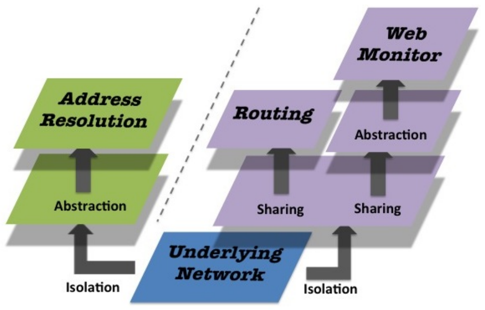


Figure 6: Application of *isolation*, *sharing*, *abstraction*.

3.2 Defining Network Views

To define network views, programmers can either use a high-level specification mechanism or a low-level specification mechanism; the former is more concise but less flexible; the latter is more verbose, but more powerful.

High-level specifications. Figure 7 presents the definition of the simplest possible abstract network, which makes two physical switches appear as though they are just one. It then deploys the `hub` policy, defined in Figure 3, on top of the virtual switch. The high-level takeaway from this example is simply that the virtual switch definition and the forwarding policy are defined by completely separate pieces of code. The given virtual switch can be replaced by another switch that includes 3, 4, *etc.*physical switches connected via an arbitrary topology. Likewise, to choose to run a learning switch instead of a hub on the virtual network, the programmer need only change one line of code, replacing the call to `hub` with the call to `learn`, defined in Figure 5.

Examining the details of Figure 7, the central elements involved in defining a virtual network are the virtual map (`vmap`) and the virtual topology (`vtopo`). The virtual map associates each virtual switch and port in the network with a physical switch and port. For instance, the second line of the `vmap` definition in Figure 7, marked `(B)`, read left to right, dictates that virtual switch `V`'s second port is implemented as physical switch 2, port 1.

Additionally, one may specify a forwarding policy that routes packets along the physical network to their (virtual) destination. If missing, as in our example, a shortest path routing policy is automatically generated for us using the virtual topology.

Low-level specifications. The low-level specification language can be thought of as a compiler intermediate language: It serves as the target for compilation of the high-level virtual network specifications, and is itself further compiled into ordinary, core FV policies. While we view it as an intermediate language, programmers are still free to craft topology transformations in it directly.

Figure 8 re-presents the virtual hub specified in Fig-

```
V = Switch(1)                    # virtual
P1 = Switch(1); P2 = Switch(2) # physical
vmap = {
  (V, 1): [(P1, 1)],            # (A)
  (V, 2): [(P2, 1)]             # (B)
}
def virt_hub(net):
  vnet = abstract(net)
  vnet.from_vmap(vmap)
  vnet.topology = Graph({V, ports={1, 2}})
  hub(vnet)
```

Figure 7: High-level virtual hub.

ure 7. In fact, the code of Figure 8 is essentially the output of the first phase of our specification compiler, modified slightly for presentation. The components to focus on in this new figure are the specification of the network *ingress_policy*, *egress_policy*, and *physical_policy*. These three components are specified using core FV policies. The primary task of these policies is to manage and use the virtual fields `vinport` (the virtual inport where the packet entered the switch), `voutport` (the virtual outport the packet is destined to) and `vswitch` (the virtual switch the packet is currently travelling across).

To understand the compilation process, begin by considering the `ingress_policy` in Figure 8. This definition was generated by extracting the ingress information contained in lines `A` and `B` of Figure 7. It states that if a packet arrives on physical switch `P1`, inport 1 then the `vinport` should be set to 1 and if a packet arrives on physical switch `P2`, inport 1 then the `vinport` should be set to 2. In either case, the `vswitch` field should be set to record the fact that the packet has entered virtual switch `V`. Notice that `ingress_policy` does not move (that is, modify the `outport` field) the packet anywhere—it does not need to. That will be taken care of by the combination of the physical policy and the client policy. The beauty of the basic FV design is that it is possible to separately specify a distinct logical component and then to compose it with other components to create a meaningful policy.

The `egress_policy` defines when packets leave a virtual switch. Again, the specification in Figure 8 is generated from information contained in lines `A` and `B` of Figure 7. This policy states that when a packet arrives at on physical switch `P1` (on any port), and is destined for virtual outport 1, or arrives at on physical switch `P2` (on any port), and is destined for virtual outport 2 then it is about to exit the virtual switch. When a packet leaves a virtual switch, we remove its virtual headers using the `pop` command.

Finally, the `physical_policy` describes the available routes across the virtual switch. For instance, the first disjunct of the `physical_policy` states that if

```
ingress_policy =
  match(switch=P1, inport=1) &
    push(vswitch=V, vinport=1)
| match(switch=P2, inport=1) &
    push(vswitch=V, vinport=2)
egress_policy = match(vswitch=V) &
  if_(match(switch=P1, voutport=1)
    | match(switch=P2, voutport=2),
    pop("vswitch", "vinport", "voutport"),
    passthrough)
physical_policy = match(vswitch=V) & (
   match(switch=P1, voutport=1) & fwd(1)
 | match(switch=P1, voutport=2) & fwd(2)
 | match(switch=P2, voutport=1) & fwd(1)
 | match(switch=P2, voutport=2) & fwd(2))
def virt_hub(net):
  vnet = abstract(net)
  vnet.ingress_policy = ingress_policy
  vnet.physical_policy = physical_policy
  vnet.egress_policy = egress_policy
  vnet.topology = vtopo
  hub(vnet)
```

Figure 8: Low-level virtual hubs.

the virtual switch is `V` and the physical switch is `P1` and the virtual outport is set to 1 then the packet should be forwarded out of port 1. Otherwise, if the virtual outport is 2 then the packet should be forwarded out of port 2. Notice that the physical policy does not set the forwarding policy; it provides alternate paths through the physical switching fabric. The compiler arranges for the client policy to set the virtual outport (or not if the client chooses to drop a packet) and thereby determines the eventual route for packets through the physical network.

There are several reasons why programs written at this level of abstraction have more power than those written in the terse high-level language. First, these low-level policies give programmers the flexibility to set the ingress and egress policies with more precision. For instance, if the programmer wish to admit only a subset of the traffic arriving on physical switch `P1`, inport 1, to the their network, she could add an additional constraint to the policy. For example, rewriting the first disjunct of the ingress policy as follows blocks entry by packets with source IP `1.2.3.4`.

```
match(switch=P1, inport=1)
   & ~match(srcip=1.2.3.4)
   & push(vswitch=V, vinport=1)
```

More generally, the implementer of a network can provide transparent services to client programs by refining network definitions at this level of abstraction. As another simple example, consider the possibility of monitoring the traffic flowing from physical switch `P1` to physical switch `P2`. To do so, we might set up a bucket `b` and siphon off the appropriate traffic by modifying the physical policy:

```
match(vswitch=V, switch=P1) & (
def virtualize_policy(ingress_policy,
                      egress_policy,
                      physical_policy,
                      user_policy):
  return if_(~match(vswitch=None,
                    vinport=None,
                    voutport=None),
     (ingress_policy                  # A
      >> move(switch="vswitch",       # B
              inport="vinport")       # C
      >> user_policy                  # D
      >> move(vswitch="switch",       # E
              vinport="inport",       # F
              voutport="outport")),   # G
      passthrough)
  >> physical_policy                  # H
  >> egress_policy                    # I
```

Figure 9: Syntactic virtualization transformation.

```
    match(voutport=1) & fwd(1) |
    match(voutport=2) & fwd(2) & fwd(b))
```

Other possibilities include erecting a firewall between switches or balancing the load across a set of physical switches. Importantly, while the code fragments presented here illustrate the use of static policies for the sake of simplicity, we can just as easily set up dynamic policies that react to network events such as changes in load or new switches coming online or going offline.

Compiling to core FV. Compilation is a simple syntactic transformation on the combination of the low-level specification components (the ingress policy, the egress policy, and the physical policy) and the high-level (virtual) user policy. Figure 9 defines a function `virtualize_policy` that implements the core elements of the transformation. In this figure, we have elided just a few, fairly minor elements of the translation. Nonetheless, the fact that we can implement the core elements of an arbitrary topology transformation in just a few lines highlights the remarkable expressive power of the core FV policy language.

To understand the transformation, let us look at a sample packet `P = {switch:P2, inport:1}` and trace through the effect of the transformation on the virtualized hub in Figure 8. The first step is to check if the incoming packet is already virtualized. We do so by testing for the presence of virtual headers. If the packet hasn't been virtualized, we assume that it is at the ingress of a virtual switch and we must calculate the routing parameters (virtual switch, virtual inport, virtual outport) for the physical policy to begin routing the packet. Otherwise, we assume that the packet is inside a virtual switch that spans more than one physical switch. In such a situation, we already have the necessary routing information, and do not need to consult the ingress or user policies.

In this case, `P` has not been virtualized, and we take the first branch of the `if_` policy, marked `A` in Figure 9. Next we invoke the `ingress_policy` which, by the second disjunct, pushes the headers `vswitch=V` and `vinport=2`, giving us the packet `P'`:

```
{switch:P2, inport:1, vswitch:V, vinport:2}
```

Our next step will be to calculate the outport of the virtual switch that `P'` should leave the virtual switch (if any). The `user_policy` is responsible for determining the outport. Before running it, however, we must lift the packet's apparent environment to the same level of abstraction as the `user_policy`. To do this, we move the virtual switch header to the physical switch header and do the same for the virtual inport and physical inport headers (lines `B` and `C`). After the move, we obtain the packet `P''`:

```
{switch:[V, P2], inport:[2, 1], ...}
```

and, since policies can only inspect the top of the stack of any header, the virtualization is transparent to the user policy (applied to the packet at line `D`). The tenant floods, which, respect to our virtual topology and inport, generates the packet `P'''`:

```
{switch:[V, P2], inport:[2, 1], outport:1, ...}.
```

Finally, before sending `P'''` to the physical policy for routing along the virtual switch fabric, we must restore the virtual headers, as it is in general important to know both the current physical parameters and the (destination) virtual parameters. After one final move (lines `E`, `F`, `G`), we obtain the final packet `Proute`, which will traverse the network:

```
{switch:P2, inport:1,
 vswitch:V, vinport:2, voutport:1}.
```

We now pipe `Proute` into the physical policy for the first hop (line `H`). The physical policy, by the third disjunct, sends `Proute` out of port 1, and the egress policy (line `I`) does not apply as we have yet to leave V. On the next hop, we process `Proute'`:

```
{switch:P1, inport:2,
 vswitch:V, vinport:2, voutport:1}.
```

This packet has already been virtualized, so we do not calculate virtual headers for it and instead skip straight to the physical policy, which sends the packet out of port 1. This time, however, the forwarded packet matches the egress policy, which pops the virtual headers before sending the packet. This insures two properties: first, if the next hop is a host, the packet has no virtual headers and therefore does not need a VLAN header, which prevents many hosts from properly processing the packet. Second, if there was another virtual switch next, the packet at the next hop would arrive without virtual headers, triggering the first branch of the `if_` policy to calculate the necessary virtual headers for routing the packet along the virtual switch's fabric.

```
def topology_to_vmap(topo):
  vmap = {}
  for switch,port in boundary(topo):
    vmap[(v_switch, Port(len(vmap)))] =
      [(switch, port)]
  return vmap

vn = abstract(net)
for t in net.topology_changes:
  vmap = topology_to_vmap_dict(t)
  vn.from_vmap(vmap)
```

Figure 10: Automated "big switch" view construction.

3.3 Automating Dynamic Views

FV is not limited to the static view definitions shown earlier. Indeed, FV allows programmers to write dynamic views that adapt to the current network topology.

More precisely, a dynamically changing topology can be considered a stream of static topologies. FV provides access to this stream via a network's `topology_changes` attribute. Figure 10 shows a function can automatically create an appropriate transformation mapping between a single "big switch" and an arbitrary underlying network topology. Each time the network topology changes, the `topology_changes` attribute generates a new static topology from which a new mapping is automatically calculated and a new view built. Generating other transformed views (e.g., spanning tree, multicast trees) becomes as simple as modifying the `topology_to_vmap` function.

4 Case Study: Gateway Forwarder

FV's key contribution lies in making it easy to write modular network applications. This is not a contribution that can be easily assessed through quantitative plots. Rather, in this section we demonstrate how FV can succinctly address the motivating scenarios presented in Section 1.1. Space constraints restrict us to delving into only two of those scenarios in the following discussion. However, the complete code solutions integrated with the FV compiler/runtime system and Mininet are available at `http://www.frenetic-lang.org/`[2].

Fundamentally, the logic needed to implement the solution outlined in Figure 1, consists of a client program paired with a view for each of Ethernet, Gateway, and IP core. Of these, the logic of the gateway forwarding is the most interesting.

```
# C is the center node switch
# V1, V2, V3 are virt. switches

ingress_policy =
  match(at=None) & (
    match(switch=C, inport=1)
      & push(vswitch=V1, vinport=1)
  | match(switch=C, inport=2)
      & push(vswitch=V1, vinport=2)
  | match(switch=C, inport=3)
      & push(vswitch=V3, vinport=2)
  | match(switch=C, inport=4)
      & push(vswitch=V3, vinport=3))
| ((match(at=(1, 3))
      & push(vswitch=V1, vinport=3)
  | match(at=(2, 1))
      & push(vswitch=V2, vinport=1)
  | match(at=(2, 2))
      & push(vswitch=V2, vinport=2)
  | match(at=(3, 1))
      & push(vswitch=V3, vinport=1)) >>
                            pop("at"))

egress_policy = pop("vswitch",
                    "vinport",
                    "voutport")

# (C)
physical_policy = lambda vnet:
    match(vswitch=V1, voutport=1) & fwd(1)
  | match(vswitch=V1, voutport=2) & fwd(2)
  | (match(vswitch=V1, voutport=3) &
      (push(at=(2, 1)) >> vnet.vpolicy)
  | match(vswitch=V2, voutport=1) &
      (push(at=(1, 2)) >> vnet.vpolicy)
  | match(vswitch=V2, voutport=2) &
      (push(at=(3, 1)) >> vnet.vpolicy)
  | match(vswitch=V3, voutport=1) &
      (push(at=(2, 2)) >> vnet.vpolicy)
  | match(vswitch=V3, voutport=2) & fwd(3)
  | match(vswitch=V3, voutport=3) & fwd(4)

vtopo = Graph()
vtopo.add_node(V1,
    ports={Port(1), Port(2), Port(3)})
vtopo.add_node(V2,
    ports={Port(1), Port(2)})
vtopo.add_node(V3,
    ports={Port(1), Port(2), Port(3)})
vtopo.add_edge(V1, V2)
vtopo.add_edge(V2, V3)


# gw_ip is IP for V2, gw_mac is MAC for V2

def gateway(net):
  pol = ~match(dstip=gw_ip)
        & match(switch=V2, inport=1)
        & fwd(2)
  net.install_policy(pol)

  respond_to_arp(gw_ip, gw_mac)

  q = query_unique(net,
        match(switch=V2, inport=2),
        fields=["dstip"])

  for pkt in q:
    mac = arp.resolve(pkt.dstip)
    pol |= match(dstip=pkt.dstip)
           & modify(dstmac=mac) & fwd(1)
    net.install_policy(pol)

def virt_gateway(net):
  vnet = abstract(net)
```

[2]Code will be released no later than the NSDI camera-ready deadline.

```
# (A)
vnet.partial_virt = (C, 1): (V1, 1),
                    (C, 2): (V1, 2),
                    (C, 3): (V2, 2),
                    (C, 4): (V2, 3)
vnet.ingress_policy = ingress_policy
vnet.egress_policy = egress_policy
# (B)
vnet.physical_policy = physical_policy(vnet)
vnet.topology = vtopo

run(learning_switch, vnet)
run(ip_core, vnet); run(gateway, vnet)
```

Since the view seen by the gateway has only two ports, `port 1` coming from the Ethernet and `port 2` coming from the IP core, all the gateway needs to do for incoming packets on `port 1` is either respond to them with the appropriate MAC address (if the packet is an address resolution request) or, otherwise, forward across to the IP core. Traffic in the reverse direction is hardly more complicated. Incoming packets must be destined for an address on the local LAN. Thus for each new destination IP, that address must be resolved to the corresponding layer two MAC and then fowarded out `port 1`. Once packets leave the gateway forwarder on `port 1` or `port 2`, they will be handled appropriately by either the learning switch or IP client programs, respectively, without any need for special purpose logic in the gateway forwarder. Furthermore, the same gateway forwarding code will work under multiple different implementations of either adjacent client program.

The implementation of the virtualization is similar to the low-level virtual hub in Figure 8, except for two more advanced implementation techniques. The first feature, on the line below the marker `(A)`, signals to FV that only *partial* virtualization is desired. This has the effect of `vnet` inheriting `net`'s topology, and rewiring links to `C` to `V1` and `V3` (the topology on `V1`, `V2`, and `V3` is captured by `vtopo`). It also has the effect of making the ingress, egress, and physical policies passthrough packets not on `vtopo` without virtualization.

The second implementation technique (lines `(B)` and `(C)`) exploit lazy packet evaluation of policies by constructing a recursive policy (`vnet.vpolicy` refers to the completely virtualized policy). This technique is necessary to simulate packet forwarding behavior on virtual switches that are not mapped to any underlying physical switch, such as the gateway `V2` (we use the internal `at` header as local state to disambiguate what virtual switch we are actually at).

5 Related Work

FV's module language supports rich interactions between modules, encompassing *abstraction*, *isolation*, and *sharing*, as discussed in Section 3.1, One or both of these first two have been supported to some degree in

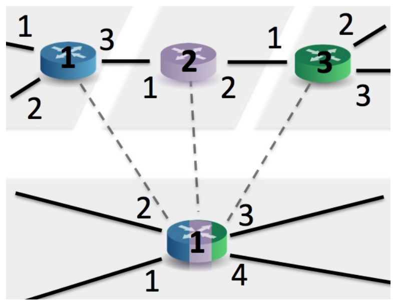


Figure 11: Case study topology.

second-generation SDN controllers (*e.g.,* [20, 7, 14, 10, 17]). Most controllers are either commercial platforms, or ongoing open-source efforts, making it difficult to pinpoint the features of each system precisely. Based on publicly-available information, we compare FV to previous work based on form(s) of interaction supported:

Isolation: Some platforms use virtualization to support *isolation* by allowing each module to handle traffic for a different tenant or traffic class [20, 7, 14]. FV not only supports these isolation features, but allows them to be used in combination with both sharing and abstraction.

Abstraction: Some controller platforms allow applications to run on a completely abstracted view of the topology in which the network appears as one layer-2 switch [14] or a subgraph of the physical network [20, 7]. Others provide underlying facilities such as global network information bases which could be used to support abstraction [10, 17]. And yet others such OpenStack's Quantum [16] project incorporate such capabilities into cloud orchestration platforms. Yet, none provide the fundamental capability to flexibly describe how to run one module on an arbitrary virtual topology of virtual switches implemented by another module, as in FV.

Sharing: While we are not aware of any controller platform supporting sharing, the Frenetic [4] language does provide limited sharing to single-writer, multiple-reader programs. However, the output of one program cannot be used as the input to another, nor does it provide the abstraction and isolation features needed to productively compose multiple writers.

Beyond recent work on SDN controllers, network virtualization has long been an active research area. The term "network virtualization" has many different interpretations, such as early VLAN and VPN technologies, overlay networks running on end hosts, and techniques for embedding virtual topologies on a shared network infrastructure. In contrast, our work on FV does not target a specific use case (*e.g.,* sharing a network testbed, hosting multiple tenants in a data center, dividing physical resources, or hiding the internal network topology). Instead, these are specific applications that could run on *top* of FV. Instead, FV is a general platform for building modular, portable, and reusable control applications.

FV is part of a line of research on applying programming language techniques to SDN, including research on FML [8], Nettle [22], Resonance [13], and Frenetic [4, 11]. Our main contribution is a strong emphasis on building modular programs through composition. This work goes far beyond our original Frenetic system to support "programming in the large" through abstract network views and sequential composition, as well as a familiar imperative style of programming.

6 Conclusion and Future Work

We believe the right level of abstraction for programmers is not a low-level interface to the data-plane hardware, but instead a higher-level language for writing and composing modules. FV is a new language that allows SDN programmers to build large, sophisticated controller applications out of small, self-contained modules. It provides the programmatic tools that to allow network managers to master the complexities of their domain.

References

[1] Beacon: A Java-based OpenFlow control platform., Nov 2010. See `http://www.beaconcontroller.net`.

[2] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *PRESTO*. ACM, 2010.

[3] Floodlight OpenFlow Controller. `http://floodlight.openflowhub.org/`.

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, Sep 2011.

[5] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[6] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[7] S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, Aug 2012.

[8] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *WREN*, pages 1–10, 2009.

[9] E. Keller and J. Rexford. The 'platform as a service' model for networking. In *Internet Network Management Workshop and Workshop on Research in Enterprise Networking*, Apr 2010.

[10] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, Oct 2010.

[11] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. *SIGPLAN Not.*, 47(1):217–230, Jan 2012.

[12] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programs. In *POPL*, Jan 2012.

[13] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control in enterprise networks. In *WREN*, Aug 2009.

[14] Nicira. It's time to virtualize the network, 2012. `http://nicira.com/en/network-virtualization-platform`.

[15] Nicira. Networking in the era of virtualization. 2012.

[16] OpenStack. OpenStack Quantum, 2012. `http://wiki.openstack.org/Quantum`.

[17] `http://www.noxrepo.org/pox/about-pox/`.

[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, Aug 2012.

[19] S. Shenker. The future of networking and the past of protocols, Oct 2011. Invited talk at Open Networking Summit.

[20] R. Sherwood, M. Chan, G. Gibb, N. Handigol, T.-Y. Huang, P. Kazemian, M. Kobayashi, D. Underhill, K.-K. Yap, G. Appenzeller, and N. McKeown. Carving research slices out of your production networks with OpenFlow. *SIGCOMM CCR*, 40(1):129–130, 2010.

[21] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI*, Oct 2010.

[22] A. Voellmy and P. Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, Jan 2011.