Towards a Modular and Flexible SDN Control Language

Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla

Towards a Modular and Flexible SDN Control Language

Messaoud Aouadj, Emmanuel Lavinal, Thierry Desprats, Michelle Sibilla
University of Toulouse, IRIT
118 Route de Narbonne, F-31062 Toulouse, France
Email: {aouadj, lavinal, desprats, sibilla}@irit.fr

***Abstract*—Software Defined Networking (SDN) is a recent paradigm that aims to reshape the way we configure and manage today's networks. To fulfill this goal, SDN relies on control languages to programmatically express the desired network behavior, making it possible to quickly change and innovate within the network. As a consequence, having an expressive and powerful control language will unlock the full potential of this approach and enable new opportunities for developing network control applications. As such, numerous works addressed this issue, where the most recent ones have used network abstractions in order to spare administrators from dealing with the complex and dynamic nature of the physical infrastructure. However, we think that these languages rely on abstractions that are not the most appropriate ones for expressing modular and reusable control policies. In this paper, we present work in progress towards a new high-level, modular and flexible SDN control language. One novelty of this language is to integrate a network abstraction model that allows a *clear separation* between simple transport functions and richer network services. We believe that this approach will allow administrators to design and deploy control applications that can be easily maintained and reused.**

I. Introduction

Networks have become increasingly dynamic, complex and hard to control due to major evolutions in computing environments, such as the exponential growth of mobile devices, desktop and server virtualization, the wide adoption of cloud computing or the advent of "Big Data". Administrators are therefore looking for more flexible networks that can quickly adapt to the evolving needs of today's enterprises, carriers, and end users. Software Defined Networking (SDN) is the latest attempt in order to respond to this lack of flexibility of current network architectures. In order to do so, SDN decouples the control plane (which decides how to handle packet flows) from the data plane (which forwards packet flows according to decisions taken by the control plane), and centralizes it in a logical and programmable entity called *controller* [1, 2, 3, 4]. By using this logical and central point of control, network administrators are able to quickly define and change network behavior by simply (re)programming the controller using the provided programming interfaces. Communications between the control plane and the data plane are then enforced via an open and a well-defined protocol such as Openflow, which is currently the most accepted standard [5].

Unfortunately, current SDN controllers provide low-level programming interfaces that have several limits, the most restrictive ones being: *i)* the inability to write separate modules that compose and *ii)* the obligation to directly deal with the complex and dynamic nature of the physical infrastructure.

Regarding the first restriction, it is well known that administrators often need to install multiple functions on their networks such as routing, monitoring, access control or load balancing. Using programming interfaces of current controllers to implement these functions as independent and separate modules, that can later be composed to achieve high-level goals, is a complicated and error-prone process. Indeed, to compose existing control modules, it is necessary to *manually* combine their logic in a totally new program in order to avoid overlapping problems between rules of different modules that apply on the same packet flows.

The second restriction is that network administrators are obliged to specify their control policies directly upon the global view of the physical infrastructure that is provided by the controller. The drawbacks with this approach is that administrators need, on the one hand, to deal with a large amount of informations that are irrelevant to their high-level goals (*e.g.* even in the case of specifying an access control policy, administrators must also consider issues related to packet forwarding between intermediate nodes) and, on the other hand, they have to constantly adapt their policies to changes that may occur in the physical infrastructure (*e.g.* discovery of a new path, link or device failure).

All these limits make the controller's programing interfaces less productive and difficult to use in practice, since they do not allow to build modular control programs that can be easily maintained and reused.

In this paper, we present work in progress towards the definition of a new high-level control language for SDN platforms. The goal is to design a control interface that overcomes the previously presented deficiencies. In order to do so, we designed our language so that it satisfies the following key principals:

- *Expressiveness*: the language's primitives must enable administrators to specify behaviors that describe their high-level goals, rather than specifying instructions that describe how these goals will be implemented on the underlying network.
- *Modularity*: network administrators must be able to implement their network functions as separate modules that can be, on the one hand, easily *composed* to build control programs and, on the other hand, *reused* over different physical infrastructures.
- *Flexibility*: the language must allow to specify control policies that respond to the variety of current function-

alities (*e.g.*, routing, access control, monitoring), and in various contexts of use (*e.g.*, campus networks, data centers, operator networks). Moreover, the language must not impose too strong restrictions in order to be able to meet, as far as possible, future requirements as they arise.

To satisfy these requirements, we put network virtualization at the very heart of our language. While there are many motivations to virtualize networks (*e.g.*, isolation, customized network services), easing their management is probably the most important one [6]. Indeed, virtualization exposes logical abstractions (*i.e*, virtual networks) that are decoupled from the physical infrastructure. These abstractions provide just enough information to specify high-level goals, thus making control policies both easier to write, since only the desired behavior is expressed, and modular (subsequently reusable), since they are no more attached to a particular infrastructure. However, virtualization presents two major design challenges: the choice of the network abstraction model that will be used to abstract the physical infrastructure (*i.e.*, the forwarding plane), and the technology needed (*i.e.*, the network hypervisor) to map the logical state onto the underlying physical infrastructure [7]. In this paper, we mainly address issues related to the first challenge.

The novelty introduced by our proposal language is that unlike existing works, we rely on a new network abstraction model that we think is more appropriate for our language design requirements. Indeed, since we consider virtualization as a cornerstone component, we must be mindful of the fact that the choice of the abstraction model will significantly impact the language's *fundamental* properties, namely: it's expressiveness, modularity and flexibility.

The remainder of this paper is organized as follows: in section II, existing works are briefly presented. In section III, we discuss network abstraction models that are currently used by existing control languages, then we describe our new approach. Section IV gives an overview of our language's key elements. An illustration program is exposed through a toy example in section V. Finally, we conclude and shortly present ongoing work.

II. Related Work

Proposing advanced programming interfaces for SDN controllers has already been the subject of numerous research projects. In this section we briefly present the most important ones and their main contributions.

Early works have addressed issues related to the low-level nature of programming interfaces and their inability to build control modules that compose. The *FML* language [8] is one of the very first, it allows to specify policies about flows, where a policy is a set of statement, each representing a simple *if-then* relationship. *FML* also includes two conflict resolution mechanisms which provide administrators with a convenient way to specify how different rules should be composed. *Frenetic* [9] is a high-level language that pushes programming abstractions one-step further. *Frenetic* is implemented as a python library and comprises two integrated sub-languages: *i)* a declarative query language that allows administrators to read the state of the network and *ii)* a general-purpose, functional and reactive library for specifying packet forwarding rules. Like *FML*, *Frenetic* provides constructors and operators that make queries and functions composition a straightforward exercise.

Additional recent proposals introduced modern features that allow to build more realistic and sophisticated control programs. Indeed, languages such as *Procera* [10] and *NetCore* [11] offer the possibility to query traffic history, as well as the controller's state, thereby enabling network administrators to construct dynamic policies that can automatically react to conditions like authentication or bandwidth use.

Traffic isolation issues were also addressed in works like *FlowVisor* [12] and *Splendid Isolation* [13]. *FlowVisor* is a software slicing layer placed between the control plane and the data plane. This slicing layer allows to divide the data plane into several slices completely isolated, where each slice can have its own and distinct control program. Following the same idea, Guts *et al.* proposed *splendid isolation* which is a language that allows, on one side, to define network slices in a simple and elegant way and, on the other, to formally verify isolation between these slices. *Splendid isolation* was proposed as an alternative to *FlowVisor*. Indeed, the authors argued that isolation should be formally verified at the language level instead of relaying on potentially buggy low-level mechanisms such as an intermediate software layer.

Recently, Monsanto *et al.* proposed the *Pyretic* language [14], which we believe is by far the most advanced work on building modern programming interfaces for SDN controllers. Indeed, *Pyretic* introduced two main programming abstractions that have greatly simplified the creation of modular control programs. First, they provide, in addition to the existing parallel composition operator, a new sequential composition operator that allows to apply a succession of functions on the same packet flow (*e.g.*, access control then routing). Second, they enable network administrators to apply their control policies over abstract topologies, thus constraining what a module can see (information hiding) and do (protection).

Pyretic abstract topologies may contain a mix of physical switches, and virtual ones that are overlayed over the physical infrastructure. We believe there is a better alternative that will best suit our language design requirements. In the next section, we discuss in detail abstraction models that are proposed in the literature, then we present our new approach.

III. Choosing the right abstraction model

One of the major challenges of virtualizing software-defined networks is the choice of the network abstraction model that will be used to abstract the physical infrastructure. There are currently two main approaches: *i)* the overlay network model and *ii)* the single router abstraction model.

To the best of our knowledge, most network control languages use the *overlay network* model (*Fig.* 1a) which consists in overlaying a virtual network of multiple switches on top of a shared physical infrastructure [7]. Virtual switches are very similar to standard switches in the physical infrastructure: they include lookup tables, ports and expose a set of basic forwarding actions. Virtual switches can also map to one or more physical switches, and are connected to each other within a logical topology via virtual links.

As an alternative to the *overlay network* model, Keller and Rexford proposed the *Platform as a Service* model (*Fig.* 1b) [15]. This model abstracts the network view in a single logical router (which may also be viewed as one big switch) in order to enable network administrators to focus solely on their in-network functions (*i.e.*, any functionality that benefits from being inside the network) rather then worrying about managing the virtual network. The single router includes three main processing components: 1) a routing component that provides the ability to customize path selection 2) a data plane component that exposes some basic functionalities like forwarding and 3) a general-purpose processing component that exposes in-network functions like firewall, load balancing or access control. Also, the model has been extended by McCauley *et al.* [16] in order to better respond to large-scale networks characteristics.

The question, then, is which model to choose for our network control language, taking into account that the abstract model must ensure the expressiveness, the modularity and the flexibility of the language.

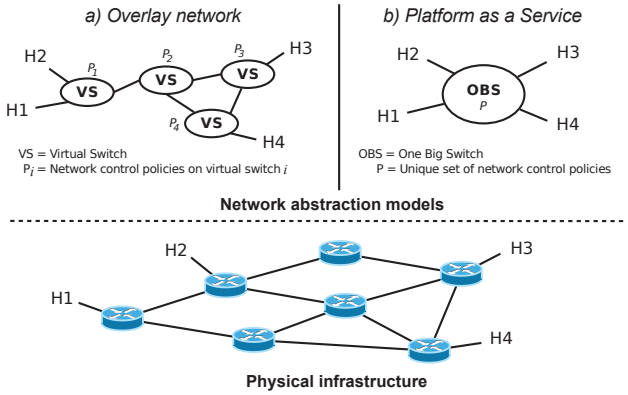

Fig. 1. Existing network abstraction models

A. Models discussion

As mentioned earlier, the biggest advantage of using the *Platform as a Service* model is that it allows network administrators to focus only on the expression of in-network functions that they plan to install on their network. However, we think that, from a network programming language point of view, this model suffers from a big disadvantage: it forces network administrators to put different in-network functions within the same router, thereby the resulting application will be a monolithic program in which the logic of different in-network functions are inexorably intertwined, making them difficult to test, debug, maintain and reuse. Moreover, forcing network administrators to always use a single router as an abstract topology can significantly affect one of the language's fundamental characteristics, namely its flexibility. For instance, using this model clearly makes it difficult to define middlebox functions (*e.g.*, firewall, deep packet inspection) or to represent a network that contains multiple administrative boundaries.

On the other hand, the *overlay network* is a more modular approach, since the model allows network administrators to define multiple logical switches, on which they can install in-network functions. These switches can be afterwards reused to, easily and quickly, construct sophisticated network control applications. However, we think that this model suffers from one major shortcoming, that is, unlike the *platform as a service* model, there is no distinction between in-network functions and packet transport functions, despite the fact that these two auxiliary policies solve two different problems. Indeed, this shortcoming makes the definition of in-network functions more difficult, since their specification must consider issues related to packet transport across the virtual network (*e.g.*, selecting the appropriate virtual path among several available).

B. Edge and Fabric: lifting up the modularity at the language level

In order to overcome the limitations of both models, we relied on a well-known idea within the network designer community, which is making an explicit distinction between the network edge and network core devices, as it is the case with MPLS networks.

Explicitly distinguishing between edge and core functions was also used by Casado *et al.* in a proposal for extending current SDN infrastructures [17]. We propose to integrate this concept in our network abstraction model (*Fig.* 2), thereby *lifting it up* at the language level. Network administrators will thus build their virtual networks using two types of virtual devices:

- *Edges* which are general-purpose processing devices used to support the execution of in-network functions.
- *Fabrics* which are more restricted processing devices used to deal with packet transport issues.

Considering the above discussion, using edges and fabrics will allow us to overcome the limitations of both previous models. Indeed, using fabrics enables network administrators to abstract packet transport issues, thereby allowing them to focus solely on the definition of complex in-network functions. By contrast, the possibility to use multiple edges allows, on one side, to maintain the language's flexibility and, on the other side, to decouple and distinguish in-network functions, thus facilitating their test, debug and, more especially, their reuse.

Finally, we believe that decomposing control policies into transport and in-network functions will enable network administrators to write control programs which are much easier to understand, reason about and maintain.

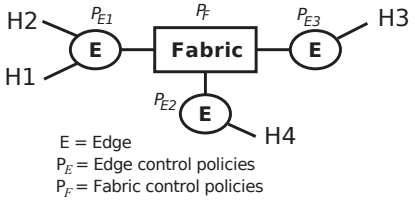

Fig. 2. "Edge and Fabric" network abstraction model

IV. Language overview

Using the Pyretic language, that we have previously presented in section II, to specify control policies upon an abstract topology is a challenging task, mainly because administrators need to use a complex transformation process which involves writing three auxiliary policies that make use of switches and links of the physical infrastructure.

In order to avoid such difficulties, we have chosen to make a complete separation between control policies and the physical infrastructure. Indeed, our programming approach is that network administrators provide two separate modules: the first one contains the principal control program, and the second one is a simple initialization module that gives information about the mapping between virtual units and switches present at the physical level. Regarding the control program, it will be composed of two main parts: the first part deals with the design of the virtual network, and the second part contains control policies that will be applied over the virtual network, without any reference to the physical infrastructure. In the following, we describe in more detail each of these parts. Figure 3 summarizes the key elements of the language.

A. Virtual network design

In order to allow network administrators to easily and clearly design their virtual networks, we have chosen a fully declarative approach. Thus, building a virtual network would only imply describing virtual devices and the connections (*i.e.*, virtual links) that exist between them.

Virtual Network Design:
addHost (name)
addNetwork (name)
addEdge (name , ports)
addFabric (name , ports)
addLink((name , port) , (name , port))
Edge Primitives:
Filters : match(h=v) | all_packets | no_packets
Actions : forward(destination) | modify(h=v) | tag(label) | drop
Queries : packet(limit) | byte_count(every) | packet_count(every)
Fabric Primitives:
catch(flow)
carry(destination, requirements=None)
Composition Operators:
parallel composition: +
sequential composition: ≫

Fig. 3. Summary of the language's key elements

We distinguish three types of components, depending on their role in the virtual network.

The first type of components are *hosts* and *networks* which are used to represent sources and destinations of data flows. A *host* can represent a single end system (*e.g.*, end host, application-level gateway or proprietary hardware appliance), while a *network* can represent a range of end systems. Note that the use of these two components is not mandatory, but strongly encouraged as a way to make control policies easier to read and write. Indeed, they allow administrators to manipulate identifiers that are *meaningful* to their high-level goals, instead of dealing with classical port numbers, network addresses and all sorts of other low-level parameters.

The second type of components are *edges* which are general processing devices placed at the border of the virtual network in order to support in-network functions installation. Edges can either play the role of host-network interfaces or the role of middleboxes. Indeed, following the approach we propose, ingress edges will receive incoming data flows, inspect packet's headers to identify which in-network function is to be considered, and redirect flows either to an egress edge for delivery to the destination or to an intermediate edge for potential further treatment. In addition, it is important to stress that edges are purely logical entities that can map to one or more switches in the physical infrastructure.

The third and last type of components are *fabrics* which represent the network's raw forwarding capacities. The fabric's primary purpose is packet transport. It exposes only a minimal set of forwarding primitives and uses a specific addressing mechanism that is much simpler than the one used by edges (*i.e.*, using a unique label instead of several header fields). In normal cases, all edges in the virtual network will be connected to a unique fabric. However, in some specific cases, virtual networks can include more than one fabric according to the network administrator's high level goals. Indeed, it is important to note that two fabrics within the same model will map to two *separate* collections of physical switches. This design choice allows us to capture specific network policies such as expressing an explicit physical backup path for critical data flows.

Once network administrators have finished with the description of virtual devices, they will then just need to set-up the different virtual links in order to connect hosts or networks to edges, and edges to fabrics.

B. High-level policy functions

Using two types of virtual devices, namely edge and fabric, implies having two distinct instruction sets. Indeed, this will allow the two components to *evolve separately*, focusing on their specific problems.

Fabrics expose two main primitives that are *catch* and *carry*. The first primitive captures an incoming flow on one of the fabric's ports. Data flows are identified based on a label that has been inserted beforehand by an edge. The second instruction *carry* transports a flow from an input port to an output port, it also allows to specify some forwarding requirements such as maximum delay to guarantee or minimum bandwidth to offer.

Edges are more complex devices than fabrics, and hence expose a richer set of instructions. Edge primitives are divided into three main groups : *Filters*, *Actions* and *Queries*.

Filters are primitives that do not change the packet's contents. The language's main filter is the *match(h=v)* primitive, which, when installed on a edge, returns a set of packets that have a field *h* in their header matching the value *v*.

Contrary to filters, *actions* are primitives that can change packets value or location. They are applied on sets of packets that are returned by installed filters. The simplest action is *drop*

which discards a packet received on one of the edges input port. The *forward* action allows to move, within the same edge, a packet from an input port to an output port. The *modify* action is used to update one or more of the packet's header fields. Lastly, the *tag* action allows to attach a label onto incoming packets, considering that labels are the unique information that a fabric will use to identify a packet.

The third and last group of edge primitives are *queries*. Like actions, queries are applied on filters. We distinguish two main kinds of queries depending on the type of information they return. The first kind is composed of *packet_count* and *byte_count* which, as their name suggests, allow to periodically poll packet and byte counters that are associated to filters. The second kind of query is *packet* which allows to poll entire raw packets. In addition to providing the ability to conduct network monitoring, queries enable network administrators to construct dynamic policies by allowing them to associate queries to callback functions that are executed each time a raw data is collected or a timer has elapsed.

Finally, we drew inspiration from Pyretic work in order to provide our language with composition operators that enable network administrators to easily combine, in a parallel ($+$) or a sequential ($\gg$) way, edge and fabric policies.

V. Toy example

This section presents a simple use case in which we illustrate a preliminary version of our high-level network control language. The overall management goal of this use case is to configure an enterprise network in order to prevent external access to sensitive resources. The policy is that any user who is part of the enterprise's internal network can have access to all available resources (*i.e.*, web server and computer cluster). On the contrary, external users can only have access to web resources and are not allowed to access the enterprise's cluster.

As described previously, the first step consists in describing a virtual network that matches our high-level goals, thus abstracting all irrelevant information that are related to the physical infrastructure. The following extract is used to describe the virtual network showed in figure 4 (notice that not all links are represented in this extract):

```
# Virtual network topology
topo.addEdge(name="ingress", ports=(1, 2, 3))
topo.addEdge(name="egress", ports=(1, 2, 3))
topo.addEdge(name="gateway", ports=(1))
topo.addFabric(name="fabric", ports(1, 2, 3))
topo.addNetwork(name="internal_users")
topo.addNetwork(name="Internet")
topo.addHost(name="web_server")
topo.addHost(name="computer_cluster")
topo.addLink(("ingress",3), ("fabric", 1))
topo.addLink(("gateway",1), ("fabric", 2))
topo.addLink(("egress",3), ("fabric", 3))
# ...
```

Having described the virtual network, the next step is the specification of the control policy. The subsequent piece of code represents the in-network function that will be installed on the ingress edge. This function configures an edge so that it classifies incoming internal flows as "trusted" and the external ones as "unreliable". Once the classification has been done, the edge will simply forward flows to the fabric in order to be transported to their right destination.

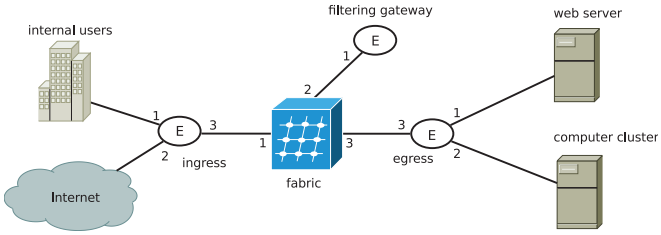

Fig. 4. Virtual network topology use case

```
# ingress function
def classify(VIdentifier):
  match(edge=VIdentifier,source="internal_users") >>
    tag("trusted_flow") >>
      forward("fabric")
  match(edge=VIdentifier,source="Internet") >>
    tag("unreliable_flow") >>
      forward("fabric")
```

In the context of this example, the gateway is only designed to analyze unreliable flows. We therefore use the following transport function to configure the fabric so that unreliable flows are transported to the filtering gateway, while trusted ones are directly transported to the egress edge.

```
# fabric function
def transport(VIdentifier):
 catch(fabric=VIdentifier,flow="trusted_flow") >>
   carry("egress")
 catch(fabric=VIdentifier,flow="unreliable_flow") >>
   carry("gateway")
```

For the gateway's configuration, we define the below in-network function that performs two actions. The first one is to discard all flows that want to reach the enterprise's cluster, since only unreliable flows are redirected to the gateway. The second one is to reclassify all web flows as "trusted" flows, since they are allowed to access the enterprise's web server.

```
# gateway function (for unreliable flows)
def filter(VIdentifier):
  match(edge=VIdentifier,destination="web_server") >>
    tag("trusted_flow") >>
      forward("fabric")
  match(edge=VIdentifier,destination="computer_cluster") >>
    drop()
```

The last in-network function simply configures the egress edge in a manner that it forwards web requests to the web server and forwards computation requests to the computer cluster.

```
# egress function
def deliver(VIdentifier):
  match(edge=VIdentifier,destination="web_server") >>
    forward("web_server")
  match(edge=VIdentifier,destination="computer_cluster") >>
    forward("computer_cluster")
```

Due to space constraints, we did not detail the control policy responsible for handling server and cluster responses.

It is important to stress that none of the previous in-network functions consider packet transport issues. Indeed, all focus only on their high-level goal (*i.e.*, classifying in ingress, filtering in gateway and delivering in egress), and at the end, functions just send data flows to the fabric which ensures the transportation to the right destination.

```
# main function
def main():
  return classify("ingress") + filter("gateway") +
    deliver("egress") + transport("fabric")
```

Finally, we call the *main* function that allows, on one side, to pass arguments to transport and in-network functions and, on the other, to orchestrate their execution in order to obtain the overall desired network behavior. Here we pass the identifier of the corresponding virtual device on which each function will apply. After execution of this *main* program, the policy resulting from the combination of the four functions is processed and enforced onto the physical infrastructure by a runtime system, which we are currently prototyping using the Python language and the POX controller [2].

VI. CONCLUSION AND CURRENT WORK

This paper described the design of a new high-level language for "programming" software-defined networks. We used network virtualization as a main feature in order to spare administrators the trouble of dealing with the myriad of irrelevant information that are related to the physical infrastructure, thus complying with the SDN promise to make network programming easier. The novelty of this language lies in the use of a new abstract model that explicitly identifies two kinds of virtual units: *i) Fabrics* to abstract packet transport functions and *ii) Edges* to support, on top of host-network interfaces, richer in-network functions (firewall, load balancing, caching, etc.). We think that this model offers the proper level of abstraction, by providing just enough information, to clearly specify the network's desired behavior according to the traffic's type. Moreover, this network abstraction model covers our language design requirements, namely its expressiveness, modularity and flexibility.

Currently, we are working on the design and the technical development of a network hypervisor that will support the control language we presented. In addition to the main control module, which contains virtual network and control policies declaration, the network hypervisor will rely on a mapping module consisting of initialization information and mapping instructions linking virtual network components to real physical elements. The definition of this module will largely depend on one side, on the physical infrastructure (network topology, host deployment, etc.) and on the other, on the administrator's virtual network design choices.

Technically speaking, these mapping instructions will mainly consist in associative arrays binding each virtual unit (*i.e.*, edge, fabric, host or network) as well as their parameters to their respective physical counterparts of the underlying infrastructure. Associating network addresses to hosts and virtual networks, or mapping an edge's ports to physical ones (knowing that these physical ports may belong to different physical switches) are examples of such mapping instructions. These mapping rules will be reused afterwards by the network hypervisor's runtime in order to generate a policy for the physical infrastructure that is semantically equivalent to the one applied over the virtual network. It will then be the hypervisor's responsibility to enforce the policy on the underlying network.

We are presently implementing our high-level network control language as a domain-specific language embedded in Python. To map the logical state of the virtual network onto the physical infrastructure, the prototype relies on the POX controller, an open source development platform for Python-based SDN control applications. In the current state of work, the initialization module will be manually specified by network administrators, but our long term goal is to be able to automatically generate part of this module, by relaying in particular on topology information returned by the controller.

REFERENCES

[1] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 3, 2008.

[2] "The POX controller," online: http://www.noxrepo.org/pox/about-pox/, accessed: 2014-05-28.

[3] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[4] D. Erickson, "The beacon openflow controller," in *Proc. of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'13)*. ACM, 2013.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, 2008.

[6] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, 2013.

[7] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the Network Forwarding Plane," in *Proc. of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO'10)*. ACM, 2010.

[8] 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*. ACM, 2009.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *SIGPLAN Notices*, vol. 46, no. 9, 2011.

[10] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *Proc. of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. ACM, 2012.

[11] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *SIGPLAN Notices*, vol. 47, no. 1, 2012.

[12] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[13] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *Proc. of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. ACM, 2012.

[14] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software Defined Networks," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, 2013.

[15] E. Keller and J. Rexford, "The "Platform As a Service" Model for Networking," in *Proc. of the 2010 Internet Network Management Workshop (INM/WREN'10)*. USENIX Association, 2010.

[16] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker, "Extending SDN to large-scale networks," Open Network Summit 2013 (ONS), Research Track, 2013.

[17] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: A Retrospective on Evolving SDN," in *Proc. of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. ACM, 2012.