A Survey on SDN Programming Languages: Towards a Taxonomy

Article *in* IEEE Communications Surveys & Tutorials · April 2016

DOI: 10.1109/COMST.2016.2553778

4 authors:

Celio Trois
Universidade Federal do Paraná
13 PUBLICATIONS **67** CITATIONS
SEE PROFILE

Marcos Didonet Del Fabro
Universidade Federal do Paraná
66 PUBLICATIONS **946** CITATIONS
SEE PROFILE

Luis Carlos Erpen Bona
Universidade Federal do Paraná
101 PUBLICATIONS **496** CITATIONS
SEE PROFILE

Magnos Martinello
Universidade Federal do Espírito Santo
64 PUBLICATIONS **305** CITATIONS
SEE PROFILE

A Survey on SDN Programming Languages: Towards a Taxonomy

Celio Trois, Marcos D. Del Fabro, Luis C. E. de Bona
Departamento de Informatica (C3SL)
Universidade Federal do Parana (UFPR)
Curitiba - Brazil
{ctrois, didonet, bona}@inf.ufpr.br

Magnos Martinello
Departamento de Informatica
Universidade Federal do Espirito Santo (UFES)
Espirito Santo - Brazil
magnos@inf.ufes.br

***Abstract*—Network devices have always been considered as configurable black boxes until the emergence of Software-Defined Networking (SDN). SDN enables the networks to be programmed according to the user requirements; furthermore, it allows the network to be easily modified to suit transient demands. However, how do we program the network? SDN-compliant switches offer a low-level interface that makes programming error-prone. The controllers provide Application Programming Interfaces (APIs), but still low-level, limited and inflexible. High-level languages have the potential to be a better alternative to program the network. There exist several SDN programming languages implementing different sets of functionalities, and focusing on solving various issues. In face of all this diversity, no published paper outlines a pragmatic view allowing to compare the SDN languages. This paper presents a systematic survey of up-to-date OpenFlow-based SDN programming languages. Our approach relies on a taxonomy comprising all prominent features found in those languages. A detailed review of the existing works is carried out investigating the foundational parts of the languages with their contributions. Examples are discussed to illustrate the fundamental abstractions. Lastly, all gathered information is summarized, discussing the main ongoing research efforts and challenges. Future abstractions and features to be incorporated into the next generations of SDN programming languages are also considered.**

***Index Terms*—Software-Defined Networking, Programming languages, Northbound Interface, Language Classification.**

I. Introduction

Software-Defined Networking (SDN) is an emerging architecture that decouples the network control and forwarding functions, enabling the network to become directly programmable, and the underlying infrastructure to be abstracted for applications and services [1]. Many foundational parts for building a widely used SDN architecture need to be designed, and their standardization is still an open issue.

The Open Networking Foundation (ONF)[1] was created as a nonprofit organization dedicated to the development and standardization of Software-Defined Networking. ONF has presented a high-level view of SDN architecture, dividing it into three main layers: Data Plane, Control Plane, and Application Plane [2]. The network devices (Data Plane) are managed by a remote and decoupled Control Plane. The Control Plane observes and controls the whole network from a logically centralized point, offering functions such as access control, orchestration, and network virtualization. The network functions (e.g. routing, load balancer, firewall), are performed by the network applications (Application Plane).

The SDN architecture is detailed in *Section II*; this section also exhibits and compares real use case applications, implemented in three distinct SDN programming levels: In (1) Low-level Programming, the network devices are programmed directly through Control-Data-Plane Interface (CDPI). With (2) API-based Programming, the network is programmed by using the controllers' Application Programming Interfaces (APIs). Finally, the (3) Domain-Specific Programming Languages use the controllers' APIs to provide higher-level abstractions. The first two programming levels offer a limited set of constructs and, to implement a more complex application, takes many lines of code. Furthermore, the entire network behavior must be expressed in a monolithic application. In levels 1 and 2, the developer must manually handle the overlapping tasks, typically by programming the switches individually; hindering the programming and maintenance of SDN programs. On the other hand, the programming languages (level 3) propose higher-level constructs, including modular composition, monitoring, topology abstraction, and so forth, to automatically program the switches. The SDN programming languages are the focus of this study, and they are going to be presented in details in this survey.

SDN programming languages are evolving, however, to the best of our knowledge, there was not yet in the literature, a comprehensive survey focusing them; neither providing a detailed review of their features nor presenting their evolutions and innovations. There exist publications surveying overall aspects of SDN [3], [4], [5], [6], [7], [8], [9], [10]. There are also papers focusing on specific areas such as virtualization

[1]http://www.opennetworking.org/

[11], [12], security [13], [14], [15], [16], [17], optimizations for the transmission medium [18], [19], [20], [21], [22], traffic engineering [23], multicasting [24], controllers [25], and network (function) virtualization [26].

Given the importance of languages for programming the network, and the inexistence of surveys focusing on SDN programming languages, we have systematically studied them, investigating their features and evolutions. We have delimited, among the existing Northbound Interface (NBI) publications, those proposing a language to program the network. Using the Feature-Oriented Domain Analysis (FODA) [27] method, we have gathered all the prominent language features, mapping and grouping the similar ones. This step has been a difficult work, since there is no NBI standardization, and many different nomenclatures were found to the same given feature. These studies led to the creation of a taxonomy for classifying the languages (*Section III*). The languages were classified according to their programming paradigm, how policies are defined, how the flows are installed on switches, and the abstractions offered by them. Some of these languages provide a general and wide set of operations, including abstractions to query network information, to specify flow matching and actions, and to compose programs by grouping modules. On the other hand, some languages yield to specific areas, such as fault-tolerance or policy delegation.

An evolutionary review of the raised languages has been reported to highlight their main contributions (*Section IV*). A genealogy has also been depicted to expose the relationships between these languages. Some language examples were presented to demonstrate their syntax and to enhance the innovations introduced by them. We have summarized the gathered information, exposing some still open issues and presenting directions for future research (*Section V*) and finally, we have concluded the survey in *Section VI*.

II. Software-Defined Networking

This section discusses the SDN architecture and explains the different ways to program these networks. SDN aims to overcome the conventional network infrastructure limitations; in the existing pre-SDN network infrastructure, each device must be individually configured, eventually causing misconfigurations and errors, such as forwarding loops, unintended paths, and contract violations. Furthermore, each device has a specific purpose on the network, for example, switches are used to connect computers to a Local Area Network (LAN), routers interconnect several LANs and/or provide an Internet connection, and firewalls filter out unwanted packets [28].

SDN, in contrast, allows the functionality of the network to be defined/modified after being deployed. It presents an open architecture that makes the network configuration and management flexible and programmable. New features can be added without changing the hardware, allowing the network to evolve at the same speed as the software [29]. The SDN architecture is composed of three main layers, Data Plane, Control Plane and Application Plane. *Figure 1* displays a conceptual representation of the architectural components and their interactions. These layers and their interactions are going to be outlined below. The figure also indicates where the different programming levels reside in the SDN architecture. These levels are discussed in *Section II-A*.

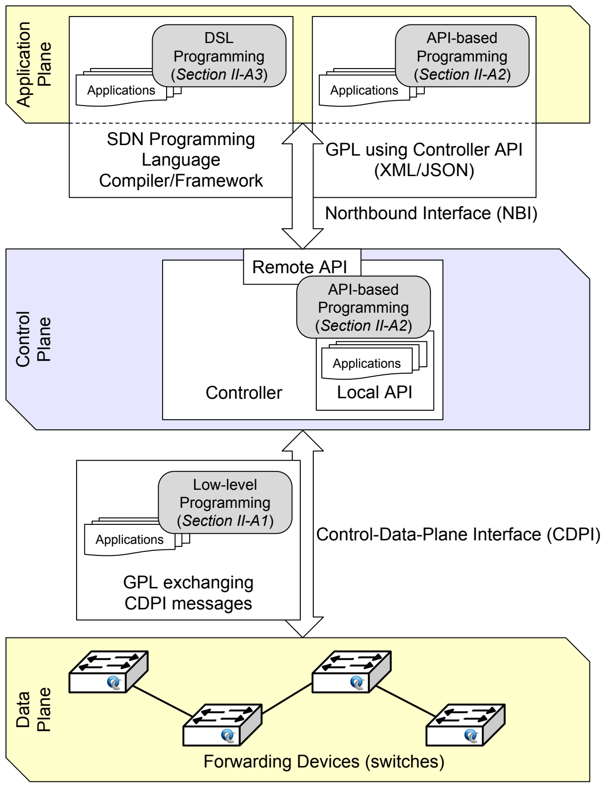
Application Plane
DSL Programming (Section II-A3)
Applications
API-based Programming (Section II-A2)
Applications
SDN Programming Language Compiler/Framework
GPL using Controller API (XML/JSON)
Northbound Interface (NBI)
Remote API
Control Plane
API-based Programming (Section II-A2)
Applications
Controller
Local API
Low-level Programming (Section II-A1)
Applications
GPL exchanging CDPI messages
Control-Data-Plane Interface (CDPI)
Data Plane
Forwarding Devices (switches)

Fig. 1. SDN Architecture.

The lowest layer in SDN architecture is the Data Plane layer; it is also known as Forwarding Plane. This layer is comprised of Forwarding Devices (network devices, or switches); they are responsible for forwarding the packets through ports and links. These devices usually use a matching table to look up the destination address of incoming packets and forwards them to the proper outgoing port(s). They implement a CDPI agent to communicate with Control Plane layer. One Forwarding Device may also be defined across multiple physical network elements [30]. Many network manufacturers are launching SDN-compatible devices[2] as well, there are various implementations of open-source software switches and switching platforms, such as Open vSwitch, Pantou, ofsoftswitch, and XORPlus [26].

The CDPI is often called as Southbound Interface. This interface is defined between Data Plane and Control Plane layers and has to provide at least programmatic control of all forwarding operations, capabilities advertisement, statistics reporting, and event notification. To date, OpenFlow [31] is the most used CDPI but it is not the only one, there exist some other proposals such as NETCONF [32], OVSDB [33], LISP [34], POF [35], and OpFlex [36]. The scope of this survey is limited to OpenFlow-based proposals because it has been adopted by ONF and it has become the de facto standard;

[2]https://www.opennetworking.org/openflow-conformance-certified-products

furthermore, the vast majority of SDN published papers use it as CDPI.

The main component of Control Plane layer is the controller. The controller exposes and abstracts network functions and operations via NBI programmatic interfaces. These functions can include orchestration [37], [38], [39], mediation [40], and virtualization [41], [42], [43], [44]. The controller enables the SDN applications to exchange information with Data Plane. There are many controller proposals, such as NOX [45], POX [46], Beacon [47], Ryu [48], Floodlight [49], and OpenDayLight [50].

NBIs are interfaces between Control Plane and Application Plane. They typically provide abstract network views, enabling to express the network behavior and requirements. Usually, this interface is provided by the controller through its remote API, through a framework providing a Northbound API (*Section IV-P*), or through a high-level SDN programming language. The programming languages are the scope of this survey, and they will be detailed throughout this paper.

Application Plane is the highest level in the SDN architecture. This plane is composed of one or more SDN Applications. These Applications consist of one Application Logic and one or more NBI Drivers. The Applications explicitly, directly, and programmatically communicate their network requirements and desired network behavior to the SDN controller via NBIs. There are many works proposing SDN Applications to deal with different network areas, such as routing, firewalling, load balancing, monitoring, and QoS.

A. SDN Programming

To understand the SDN programming, it is important to recall its basic operation [31]: When a network device receives a new packet, it searches in its flow tables to find an entry matching the packet header fields (e.g. destination MAC or IP address). If a matching entry is found, the entry's actions are performed on packets belonging to that flow (e.g. the action might be to forward out to a specified port). If no match is found, the device forwards the packet to the controller. The controller manages the switches flow tables by adding, modifying, or removing their entries.

The most common categories of SDN applications include forwarding and firewalling. We have implemented two applications: a static switch and a packet filter firewall. These examples were deployed in a simple topology consisting of two hosts and three switches (*Figure 2*). Although simple, this topology allows illustrating some advantages in programming SDN with higher-level languages supporting fundamental concepts (constructs) such as modular composition, automatic priority handling, dynamic policy instantiation, and topology abstraction.

In a static switch, all flow tables are filled manually, using the destination MAC address to forward packets out to the physical ports. In a real environment, this example is impractical because it is not scalable, *i.e.* in adding a new host or switch, all flow tables must be manually reprogrammed. The example also does not allow mobility, that is, the hosts can never change the switch's port they are connected to. The final

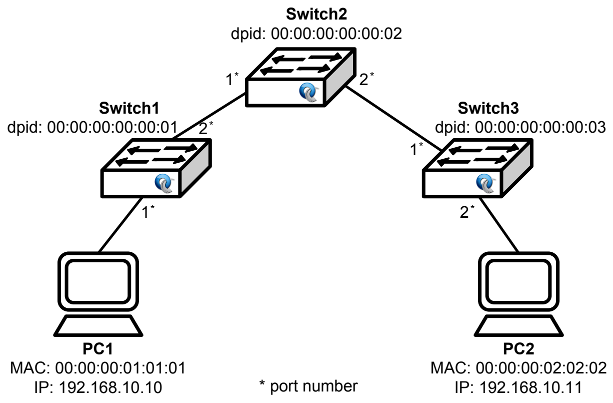

Fig. 2. Topology used to implement the example applications.

issue lies in all decisions being deployed manually, in a large environment, it becomes highly error-prone.

The packet filter firewall looks at the packet header (network addresses and ports) and determines if that packet should be allowed or blocked [51]. The second implemented application specifies packet filter rules to deny the File Transfer Protocol (FTP). This protocol uses two TCP ports to separate control and data connections. The TCP port number 20 is used to exchange the data, and TCP port number 21 is used to transfer the protocol control messages, so both ports have to be blocked. This example demonstrates that low-level premises, such as priorities, can be abstracted through languages constructs.

These applications were programmed in the three different levels shown in *Figure 1*: Low-level Programming, API-based Programming, and Domain-Specific Language (DSL) [52] Programming. Next sections explain these levels, showing the applications source code, and highlighting the differences among these programming levels.

1) Low-level Programming: We have labeled Low-level Programming when the networks are programmed directly through CDPI. OpenFlow stands out as the major initiative in CDPI, but it does not make the programming task easier. Actually, from the perspective of programming languages, it may be considered as the assembly language of network programmability. OpenFlow forces the programmers to manipulate bit patterns in packets and carefully manage the shared rule-table space [53]. Moreover, each network device must be programmed individually, errors and inconsistencies must be manually handled by the network programmer. To develop applications in this programming level, it is necessary to use a General-purpose Language (GPL) (such as C, Java, Python, or shell script) to exchange CDPI messages directly with the switches.

To implement the first example (static switch), using Low-level Programming, it is necessary to specify, in all switches, all the forwarding decisions. It is similar to configure conventional routers with static routing, but using MAC addresses rather than IP addresses. To program the switches, it has to be sent OpenFlow *FlowMod* messages, specifying the *dl_dst* in the *match* header and the destination *port* in *action* header. The structure of these messages is presented in *Figure 3*.

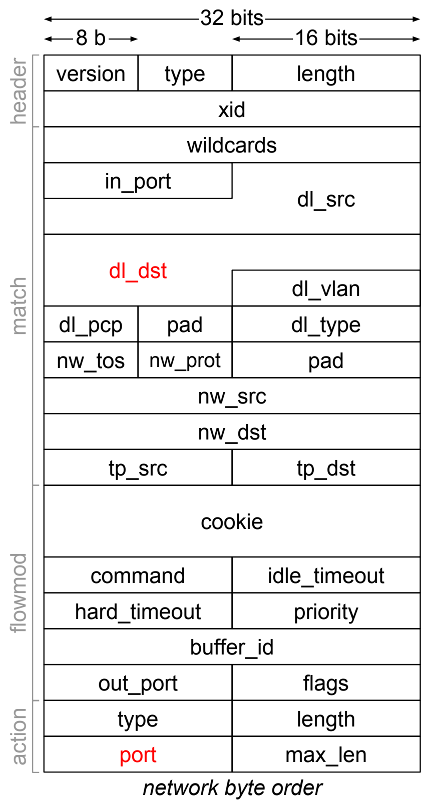

Fig. 3. OpenFlow FlowMod Message.

The topology shown in *Figure 2* was implemented using Open vSwitch [54]; this software switch implementation provides a set of tools to manage the switches. The command *ovs-ofctl* has been used to send the OpenFlow messages to configure the devices to act as a static switch. *Figure 4* shows the commands, implemented in a Linux shell script, to program all switches to act as static switches.

```
Switch1
 ovs-ofctl add-flow Switch1 priority=100,dl_dst=00:00:00:01:01:01,actions=output:1
 ovs-ofctl add-flow Switch1 priority=100,dl_dst=00:00:00:02:02:02,actions=output:2
Switch2
 ovs-ofctl add-flow Switch2 priority=100,dl_dst=00:00:00:01:01:01,actions=output:1
 ovs-ofctl add-flow Switch2 priority=100,dl_dst=00:00:00:02:02:02,actions=output:2
Switch3
 ovs-ofctl add-flow Switch3 priority=100,dl_dst=00:00:00:01:01:01,actions=output:1
 ovs-ofctl add-flow Switch3 priority=100,dl_dst=00:00:00:02:02:02,actions=output:2
```

Fig. 4. Static switch programmed directly with OpenFlow.

Similarly, the second application, the packet filter firewall, was implemented in the same topology by using the command *ovs-ofctl* (*Figure 5*). In this case, it was necessary to insert rules into the switches to drop packets with TCP destination port equal to 20 and 21. The priority was defined to 200 to be greater than the forwarding rules presented in *Figure 4*.

In the first example, flows had to be inserted into all switches to allow the communication between PC1 and PC2. When a new host is connected to this topology, all switches have to be reprogrammed, specifying rules to correctly forward traffic to it. In the second example, it was necessary to be aware of the priority of firewalling rules to be greater than the forwarding ones; otherwise, TCP communication would not be blocked.

```
Switch1
 ovs-ofctl add-flow Switch1 priority=200,tcp,tp_dst=20,actions=drop
 ovs-ofctl add-flow Switch1 priority=200,tcp,tp_dst=21,actions=drop
Switch3
 ovs-ofctl add-flow Switch3 priority=200,tcp,tp_dst=20,actions=drop
 ovs-ofctl add-flow Switch3 priority=200,tcp,tp_dst=21,actions=drop
```

Fig. 5. Packet filter firewall to deny FTP ports programmed directly with OpenFlow.

2) *API-based Programming:* Applications implemented in this programming level use the APIs exposed by the controllers. These applications are translated by the controller into CDPI messages. However, these APIs present the same problems as Low-level Programming; they oblige programmers to reason manually, in unstructured and ad hoc ways, with low-level dependencies among different parts of their code. For instance, an application that performs multiple tasks (e.g., routing, monitoring, access control, and server load balancing) must ensure that packet-processing rules installed to perform one task do not override the functionality of another. This fact leads to monolithic applications where the logic for different tasks is inexorably intertwined, making the software difficult to write, test, debug, and reuse [55].

The controllers may expose two types of APIs: local and remote. When using the local APIs, the SDN applications are implemented internally in the controller, using its classes, methods, and interfaces. These applications must be written in the controller's language, and must run on the same host. For example, to implement the first *ovs-ofctl* command, shown in *Figure 4*, using the Floodlight local API[3], it would be necessary the piece of code shown in *Figure 6*. The other commands used to program the static switch application have not been shown, because they are repetitive and to save space. However, the whole application can be made by repeating the *Figure 6* code, modifying the port (line 4) and destination MAC address (line 7).

```
// Declare the flow
 1. OFFlowMod fmTo=new OFFlowMod();
 2. fmTo.setType(OFType.FLOW_MOD);
 // Declare the action
 3. List actionsTo=new ArrayList();
 4. OFAction outputTo=new OFActionOutput((short)1);
 5. actionsTo.add(outputTo);
 // Declare the match
 6. OFMatch mTo=new OFMatch();
 7. mTo.setDataLayerDestination(
        Ethernet.toMACAddress("00:00:00:01:01:01"));
 8. fmTo.setActions(actionsTo);
 9. fmTo.setMatch(mTo);
 // Push the flow
10. staticFlowEntryPusher.addFlow("FlowTo",fmTo,dp);
```

Fig. 6. Modifying flow tables with local Floodlight API.

Remote APIs can act as an abstraction/mediation level between the two different environments; they allow the different technologies to exchange information. Usually, the information is exchanged through Extensible Markup Language (XML) or JavaScript Object Notation (JSON). Applications using this

[3]Floodlight API documentation is available at:
https://floodlight.atlassian.net/wiki/display/floodlightcontroller

type of API can be written in any GPL and can run on a different host than the controller. We have opted to implement the examples using only remote APIs because the source code is cleaner than local APIs.

POX [46] is a popular controller that provides a remote API[4] for modifying the switches flow tables. *Figure 7* shows the required JSONs to specify the static forwarding. The program *curl* may be used for sending these JSON to the controller. The API *set_table* method was used to insert the flows into switches tables.

```
Switch1
 '{"method":"set_table","params":{"dpid": "00-00-00-00-00-01","flows":
   [{"actions":[{"type":"OFPAT_OUTPUT","port":1}],
     "match":{"dl_dst":"00:00:00:01:01:01"}},
   {"actions":[{"type":"OFPAT_OUTPUT","port":2}],
     "match":{"dl_dst":"00:00:00:02:02:02"}}]}}'
Switch2
 '{"method":"set_table","params":{"dpid": "00-00-00-00-00-02","flows":
   [{"actions":[{"type":"OFPAT_OUTPUT","port":1}],
     "match":{"dl_dst":"00:00:00:01:01:01"}},
   {"actions":[{"type":"OFPAT_OUTPUT","port":2}],
     "match":{"dl_dst":"00:00:00:02:02:02"}}]}}'
Switch3
 '{"method":"set_table","params":{"dpid": "00-00-00-00-00-03","flows":
   [{"actions":[{"type":"OFPAT_OUTPUT","port":1}],
     "match":{"dl_dst":"00:00:00:01:01:01"}},
   {"actions":[{"type":"OFPAT_OUTPUT","port":2}],
     "match":{"dl_dst":"00:00:00:02:02:02"}}]}}'
```

Fig. 7. JSONs to program static switch using the POX controller API.

Some controllers offer specific APIs to ease the development of network applications; the Ryu controller, for example, provides APIs to deal with firewalling, routing, and setting QoS. So, to implement the second application, we have used the Ryu Firewall API[5].

Figure 8 shows the necessary JSONs to be sent to switches to deny the FTP ports. Due to the OpenFlow matching fields prerequisites, matching on the TCP port is allowed only if the Ethernet type is explicitly set to IPv4 and the IP protocol is set to TCP. In the same application, implemented with Low-level Programming (*Figure 5*), the *ovs-ofctl* command includes a shorthand notation to specify the Ethernet type and IP protocol[6].

```
Switch1
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "20",
   "actions": "DENY", "priority": "200"}'
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "21",
   "actions": "DENY", "priority": "200"}'
Switch3
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "20",
   "actions": "DENY", "priority": "200"}'
 '{"dl_type": "IPv4", "nw_proto": "TCP", "tp_dst": "21",
   "actions": "DENY", "priority": "200"}'
```

Fig. 8. JSONs to program packet filter firewall to deny FTP ports using the Ryu Firewall API.

Controllers provide APIs to program the network but, as it was possible to see through these examples, the functionalities offered are low-level, just like the OpenFlow protocol. We could also see that, even using a specific firewall API, all rules must be inserted manually in the right switches, and the rules priority must be carefully managed to avoid forwarding rules with higher priority.

In these two first programming levels, to create an application with multiple functionalities, for example, the static switch combined with the firewall, it is necessary to create a unique program implementing the both functions. Again, the rules priorities must be carefully inserted to avoid higher priority functions from being overlapped.

3) Domain-Specific Language Programming: A Domain-Specific Language (DSL) is a programming language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [56]. The SDN programming languages have a compiler (or framework) responsible for translating their high-level constructs to messages understandable by a controller API. All the studied SDN programming languages were classified as Domain-Specific Language (DSL), providing higher-level constructs to optimize the network resources utilization, and adding new abstractions for different programming aspects, such as monitoring, security, and virtualization.

Some languages were designed to address specific problems, while others offer more general abstractions, and can be used to create any network application. To emphasize the differences in abstractions offered by the different SDN languages, we have implemented the first application, the static switch, in two programming languages: Pyretic [53], [55] and Merlin [57].

Figure 9 shows the static switch implemented in Pyretic[7]. The Pyretic's compiler translates network applications into Ryu (or NOX) remote API commands. The Pyretic language is embedded in Python, and every program must have a *main* method and import at minimum the Pyretic core library.

To perform the static switch behavior, we have defined a method called *static_switch*. This method uses the construct *match* to filter packets with destination MAC address 00:00:00:01:01:01 (PC1). The forwarding to port 1 is performed through construct fwd. Similarly, the packets to PC2 are forwarded to port 2. This was only possible because all switches use port 1 to forward to PC1 and port 2 to forward to PC2; usually, that is not true, so it is necessary to include a new match to specify, in each switch, which port should be forwarded (e.g. $((match(switch = Switch1)\ \&\ match(dstmac = EthAddr(\text{'00:00:00:01:01:01'}))) >> fwd(1)))$.

```
from pyretic.lib.corelib import *

def static_switch():
  return ((match(dstmac=EthAddr('00:00:00:01:01:01')) >> fwd(1)) +
          (match(dstmac=EthAddr('00:00:00:02:02:02')) >> fwd(2)))

def main():
  return static_switch()
```

Fig. 9. Static switch programmed with Pyretic.

[4]POX API documentation is available at: https://openflow.stanford.edu/display/ONL/POX+Wiki

[5]Ryu Firewall API documentation is available at: https://osrg.github.io/ryu-book/en/html/rest_firewall.html

[6]The *ovs-ofctl* command translates the *tcp* keyword to "dl_type=0x0800, nw_proto=6"

[7]Pyretic documentation is available at: https://github.com/frenetic-lang/pyretic/wiki/Language-Basics

We have also implemented the static switch in the Merlin framework[8]. This framework that allows administrators to express network policies in a high-level, declarative language (*Section III-C2*) based on regular expressions. *Figure 10* shows the source code of a static switch implemented in Merlin.

```
ipSrc = 192.168.10.10 and ipDst = 192.168.10.11 -> .* ;
ipSrc = 192.168.10.11 and ipDst = 192.168.10.10 -> .* ;
```

Fig. 10. Static switch implemented in Merlin.

The Merlin language provides a higher-level abstraction ($.*$) to specify path selection; the dot symbol indicates matches any single network device and the asterisk means repetition. This abstraction can be used to define the forwarding without worrying about the switches, links, or ports; it automatically installs the necessary rules in the right switches. The Merlin compiler computes the shortest path between the hosts and program the flow entries in switches between PC1 and PC2. Further details on path selection abstraction are presented in *Section III-D6*.

The second example, the packet filter firewall to deny FTP ports, was also implemented in Pyretic and its source code is presented in *Figure 11*. To implement this new function, we have created another method, called *firewall*, to drop all packets matching TCP ports 20 and 21. The construct negation ($\sim$) in conjunction with *match*, filter only packets not matching the specified destination TCP ports (*dstport*), so FTP packets will be dropped, and all other packets can be forwarded.

```
from pyretic.lib.corelib import *

def firewall():
  return (~match(dstport=20) & ~match(dstport=21))

def static_switch():
  return ((match(dstmac=EthAddr('00:00:00:01:01:01')) >> fwd(1)) +
          (match(dstmac=EthAddr('00:00:00:02:02:02')) >> fwd(2)))

def main():
  return firewall() >> static_switch()
```

Fig. 11. Packet filter firewall to deny FTP ports programmed with Pyretic.

Both functions static switch and packet filter firewall are finally composed in the *main* method through the sequential modular compositor (>>). This compositor receives the packets outputted from *firewall* method (all packets except those with TCP destination ports 20 and 21), and forwards these packets to the *static_switch* method. The forwarding decisions implemented in the *static_switch* method are then performed.

Languages also offer advantages when implementing applications using dynamic policies (*Section III-B*). These policies permit the network behavior to change in response to some event. These events can be originated from the network itself (e.g. congested links, link up/down, or bandwidth usage) or come from an external source, for example, a user verification from an Authentication Server or an attack alert from an Intrusion Detection System (IDS).

The following example will be used to explain the advantage using a programming language to implement dynamic applications: the network access will be limited to devices based on maximum bandwidth usage. The topology shown in *Figure 2* was used to allow PC1 and PC2 to exchange up to 5GB of HTTP data per minute. To implement this example by directly inserting OpenFlow rules in the switches or using the controllers APIs, it would be necessary to implement the following algorithm in any GPL:

1) Create flow table rules matching exclusively HTTP traffic between PC1 and PC2
2) Send periodic statistics requests to the switches to collect the HTTP amount transmitted
3) Interpret the statistic replies and store them in a manner allowing to know, every minute, how much has been transmitted
4) Test if the total transmitted is lower than 5GB
    a) If so, continue forwarding traffic between PC1 and PC2
    b) Otherwise, drop the traffic until the total transmitted comes lower than 5GB.

A higher-level SDN language may specify constructs to simplify this application. For example, windowed history monitoring (*Section III-D7*) can be used to receive periodic information about traffic statistics. The source code of an application, written in Merlin, to limit PC1 and PC2 to exchange up to 5GB of HTTP data per minute is presented in *Figure 12*.

```
(bytes
  { location { PC1, PC2 } and tcp.src ~ 80 ,
     sliding(60) }
) < 5GB
```

Fig. 12. Application implemented in Merlin to limit PC1 and PC2 to exchange up to 5GB of HTTP data per minute.

This application is made up of several nested constructs. The inner-most construct describes a set of packets: all packets between PC1 and PC2 with TCP source port 80 sent or received in the last 60 second sliding window. Next, the aggregate function *bytes* reduces this set to a scalar value by taking the sum of the sizes of all packets in the set. Last, the logical constraint stipulates that this scalar must be less than 5GB.

Despite the simplicity of the examples implemented in a tiny topology, we can observe a substantial difference in favor of the languages, regarding topology abstraction, automatic priority control, or dynamic policy instantiation. Topology abstraction is relevant especially when the topology complexity increases. Automatic priority control, dynamic policy instantiation, and modular composition (*Section III-D3*) are useful to overcome the intricacy of multiple concurrent applications running on the network. Moreover, as stated earlier, in the first two programming levels, all rules must be inserted individually in every switch, where the different tasks priorities have to be handled manually.

The abstractions presented in these examples are not exhaustive concerning those offered by SDN languages, on the contrary, languages are evolving and increasingly providing higher-level constructs in many different aspects. Up to now, due to the lack of a study on SDN programming languages, it

[8]Merlin documentation is available at: https://github.com/merlin-lang/merlin

is hard to choose the appropriate language for each particular case. This extensive survey explores the existing SDN languages, showing the abstractions offered by them. A detailed review of languages is also presented, exposing the evolutions and interactions among them.

III. SDN Language Classification

We have proposed a taxonomy that allows identifing which features are offered by the SDN languages; it may assist developers and researchers in finding which language best fit their requirements. This taxonomy has been created through the Feature-Oriented Domain Analysis (FODA) method. The primary focus of the FODA method is the identification of prominent or distinctive features on specific domains, in our case, the SDN programming languages. Features are the attributes of a system that directly affect end-users, who have to make decisions regarding the availability of features in the system, and they need to understand the meaning of the features in order to use the system. A feature diagram is a hierarchically arranged set of features. Relationships between a parent feature and its child features are categorized as: *and* – all sub-features must be selected, *alternative* – only one subfeature can be selected, *inclusive or* – one or more can be selected, *mandatory* – features that required, and *optional* – features that are optional [58]. *Figure 13* presents the symbology to represent the feature diagrams.

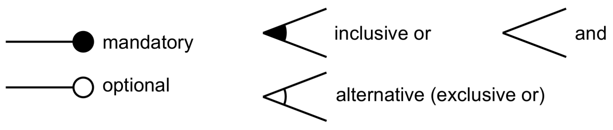

Fig. 13. Feature diagram legend.

We have firstly researched papers proposing NBI abstractions. Once those papers were identified, we have used the FODA Context Analysis to filter only the proposals presenting language-related aspects as key characteristics. These papers also described other information such as language framework, compiling techniques, and optimizations – we have focused only on language aspects. In the next FODA phase (Domain Modeling), all relevant features and their relationships were raised and mapped into a feature diagram. Due to the lack of standardization, some features presented with various names by different authors have been grouped.

Figure 14 shows the top-level feature diagram, where the first subnode level represents the major axes: Flow Installation, Policy Definition, Programming Paradigm, and the Abstractions provided by the languages. For better visualization, the *Abstractions* were plotted in a separate feature diagram (*Figure 15*). We have not included in our taxonomy elementary OpenFlow operations such as forwarding, packet header modification, and others. The features have been identified based on the existing SDN programming languages. They might not be exhaustive concerning all SDN requirements, and additional features may be included with the evolution of the languages. Potential future directions for new features and abstractions are presented in *Section V-A*.

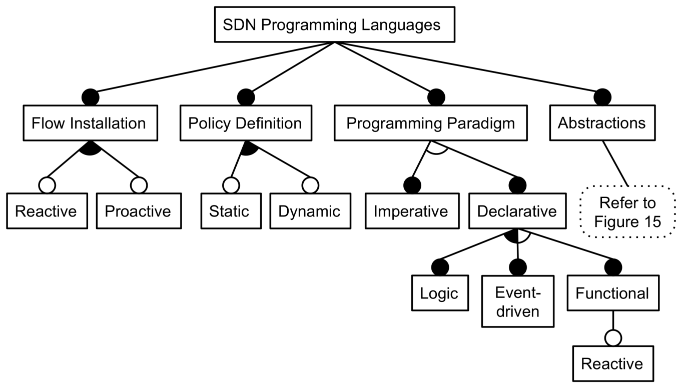

Fig. 14. Top-level classification feature diagram.

Table I classifies the languages according to the features presented in *Figure 14*. The first line refers to the way the languages install rules on switches; they can be classified as reactive and/or proactive. Languages derived from another language can inherit this feature. Languages can also allow the creation of static and/or dynamic policies; this information is summarized in the second line. The last line shows the programming paradigm. We have found languages implemented in Imperative, Logic, Event-driven, Functional, and Functional Reactive paradigms. The following sections explain all these features in further details.

A. Flow Installation

Flow installation refers to the way the languages install the rules on switches. An OpenFlow switch forwards packets according to rules stored in its flow tables. The controller handles these tables by adding or removing those rules. The rules can be installed in switches either reactively (when a new flow arrives at the switch), proactively (controller installs rules beforehand), or in a hybrid way. Hybrid flow installation is a combination of both, *reactive* and *proactive* approaches.

In the *reactive* approach, when a new packet arrives at the switch, a lookup operation is performed in its flow tables. Then if there is no match between the arriving packet and switch table entries, the switch creates an OpenFlow *packet-in* packet and sends it to the controller. The controller receives the *packet-in* and, based on its program logic, it creates and installs a rule in the switch flow tables. Languages that use reactive flow installation forward all new flows to controllers. Among the surveyed languages, we have observed several languages using reactive flow installation [59], [60], [61], [62], [63], [64], [65], [66]. The main problem with this reactive approach is that the latency is increased because the controller must be consulted on every new flow.

Proactive installation eliminates the latency induced by consulting a controller on every new flow. The flow tables are populated ahead of time for all traffic matches that could come into the switch. By pre-defining all flows and actions in the switches flow tables, the *packet-in* event never occurs. As a result, all packets are forwarded at switch processing capacity (e.g. line rate), by doing a flow lookup in the switches flow tables [67]. Languages that provide proactive installation

TABLE I
LANGUAGES FEATURES SUMMARIZATION

Feature \ Language	FML [59]	Nettle [60]	Frenetic [61]	Procera [62]	Flog [63]	NetCore [69]	Frenetic-OCaml [70]	Pyretic [53]	FlowLog v1 [72], v2 [85]	FatTire [73]	NetKAT [71]	Merlin [57]	PonderFlow [65]	NoF [66]	Kinetic [74]	Legend * = inherited
Flow Installation	R	R	R	R	R	R/P	R/P	R/P	R/P*	R/P*	R/P	R	R	R	R/P*	R=Reactive, P=Proactive
Policy Definition	S	D	D	D	D	D	D	D	D	D	D	D	D	D	D	S=Static D=Dynamic and Static
Programming Paradigm	L	FRP	FRP	FRP	L/ED	FRP	FRP	I	v1:L v2:ED	F	F	F	ED	ED	ED	I=Imperative, L=Logic, ED=Event-driven F=Functional, FRP=Funct. Reactive

pre-compute forwarding tables for the entire network; the controller just modifies the switches flow tables when it has to react in case of link failures or any other external event. The proactive approach is harder to be implemented because it requires the controller to know the traffic flows in advance[9], usually by keeping the traffic history, to configure all paths before they are used [68]. We have found languages providing proactive flow installation natively [69], [70], [55], [71] and languages that were implemented on top of these, inheriting the proactivity [72], [73], [74]. All these languages also implement the reactive approach.

B. Policy Definition

A policy is formally defined as "an aggregation of policy rules" [75]. Each policy rule is comprised of a set of conditions and a corresponding set of actions. The conditions define when the policy rule is applicable. Once a policy rule is activated, one or more actions contained by that policy rule may be executed. SDN programming languages provide high-level statements to define network policies. These policies can be triggered in two ways, statically or dynamically. We have classified the network languages according to this aspect. It is important to realize that a language can specify static policies, dynamic policies or both.

Static policies apply a fixed set of actions in a pre-determined way according to a set of pre-defined parameters [75]. Static policies are the most traditional method of firewalling. An example of a static rule-based policy would be to allow/deny an IP address to access a server. Other examples of static policies: social network traffic in not allowed during regular working hours; video-sharing websites are only permitted after 5:00 PM.

FML [59], the first SDN programming language, is the only one that allows only static policies. The code below is one FML example using static policy to allow flows from TCP port 80 and IP source 10.0.0.1.

$$allow(Flow) \Leftarrow Prot = 80 \land H_s = 10.0.0.1$$

Unlike the static, *dynamic* policies are enforced based on changing conditions of the network, such as congestion, packet loss, bandwidth usage, the loss of a network device, or any external event. To support the dynamic, and sometimes the unexpected nature of the network, actions can be triggered when an event causes a policy condition to be met [75].

Many network systems have dynamic behavior, for example, intrusion detection and prevention systems detect certain sequences of events and trigger action to deny the source of these events. An example of an application using dynamic policies has been presented in *Section III-B*. With exception to FML [59], all the other languages allow to create dynamic policies.

C. Programming Paradigm

The programming paradigm differs the way of building the structure and elements of the computer (or network) programs. Capabilities and styles of programming languages are defined by their supported programming paradigms. Each paradigm contains a set of concepts that make it more appropriate to solve certain kind of problems. For example, object-oriented programming is more suited for applications with a large number of related data abstractions, often organized in a hierarchy. Logic programming is well suited for transforming or navigating complex symbolic structures according to logical rules [76]. Next sections explain the paradigms found in SDN languages.

1) Imperative: The imperative paradigm can be viewed as the traditional computer programming model. This paradigm lets the programmer specify all the steps to solve a particular task. Computation is seen as a task to be performed by a processing unit that manipulates and modifies memory. Imperative languages provide controls to modify execution flow: loops, conditional branching statements and procedures [77]. The well-known imperative languages are FORTRAN, COBOL, Basic, Pascal, C, C++, PHP, Python, Java. Among the surveyed languages, only Pyretic [53], [55] uses this paradigm.

The following example is a Pyretic function to add firewall rules. The program verifies if the pair of MAC addresses $mac1$ and $mac2$ are set to $True$ in the $\$self.firewall$ list. If so, it just $prints$ a message on the terminal; otherwise, it sets to $True$ in the list. As can be seen in this example, the language provides the traditional if and $return$ conditional statements and also function and variables definitions.

[9]wildcard rules can be used for flow aggregation

```
def AddRule (self, mac1, mac2):
    if (mac2, mac1) in self.firewall :
        print "Firewall rule already exists"
        return
    self.firewall[(mac1, mac2)] = True
    print "Adding rule in %s : %s" % (mac1, mac2)
    self.update_policy()
```

2) Declarative: In declarative programming, one can specify what the program must do, without specifying how to do it [78], [79]. Probably the most known example of declarative languages is the Structured Query Language (SQL) used to query computer databases, where a query is stated, and the database engine is responsible for executing it. As sub-paradigm of these declarative languages, we have found proposals fitting in *logic*, *event-driven*, *functional*, and also *functional reactive* paradigms, or a mix.

Logic programming is a declarative paradigm based on formal logic. The language compiler applies an algorithm that scans every possible combination into a set of defined inference rules applied to the axioms to resolve queries [80]. The most known examples are Prolog [81], Answer Set Programming (ASP) [82], and Datalog [83]. Logical languages may implement recursion and negation. Recursion indicates that the algorithms can be implemented by means of recursive predicates. Negation allows users to specify negative predicates. We have perceived three languages defined as logic, based on Datalog. FML [59], FlowLog v_1 [72], and Flog [63] that uses both logic and event-driven paradigms.

Event-driven programming enables the programs to respond to events. Upon receiving an event, an action is automatically triggered. The action may carry out some computation as well as give rise to some new event [84]. Generally in event-driven paradigm, there exists a hidden main loop listening to the events and dispatching the appropriate actions. The following example is a program written in the second version of FlowLog [85] to blacklist all IP address trying to use telnet protocol (TCP port 23). When a *packet_in* event is received, this code is executed. The program filters all packets with TCP port = 23, and then add the sender's IP address to a blacklist.

```
ON packet_in(p) WHERE p.nwProtocol = 23 :
    INSERT (p.nwSrc) INTO blackList ;
```

Many event-driven applications are stateless: when the application finishes processing an event, the application has not been changed by it. The opposite of a stateless application is a stateful application. Stateful are applications modified by their processed events. Specifically, stateful applications remember or maintain information between events, and what they remember can be changed by events [86]. Flog [63] is a mix between logic and event-driven. The second version of FlowLog [85], PonderFlow [65], NoF [66], and Kinetic [74] were considered event-driven languages.

Functional programming treats computation as the evaluation of mathematical functions, avoiding state changes and mutable data. Functional programming languages become stateless and provide referential transparency, meaning that the result of a function will be the same for a given set of parameters no matter where, or when, it is evaluated. Referential transparency greatly enhances modularity, as a function does not depend on the context where it is being executed [77]. Thanks to the brevity, expressiveness, and availability of sophisticated data structures and algorithms, modern functional programming languages are being used in a wide range of scientific applications, from numerical analysis to visualization [77]. As examples of functional computer programming languages, one can cite Erlang [87], OCaml [88], Clojure [89], and Haskell [90]. Among the surveyed languages, we have encountered three functional languages [73], [71], [57].

The *reactive* programming paradigm provides abstractions to express programs as reactions to external events. It automatically manages time and data flows and also computation dependencies. Programmers do not need to worry about the order of events. This property of automatic management of data dependencies can be observed in spreadsheet systems. A spreadsheet typically consists of cells, which contain values or formulas. If a value of a cell changes then, the formulas are automatically recalculated [91].

Functional Reactive Programming (FRP) [92] is a method of modeling reactive behavior in purely functional languages. FRP permits the modeling of systems that must respond to input over time in a simple and declarative manner. A program in an FRP language corresponds quite closely to a mathematical model of the system being implemented. The primary concepts of FRP are signals (time-varying values) and events (collections of instantaneous values, or time-value pairs). FRP achieves reactivity by providing constructs for specifying how signals change in response to events [93]. The following code is a simple Nettle [60] example to flood all network packets. The *packetIns* filters only the OpenFlow *PacketIn*[10] messages from the incoming message stream. For each such event, it applies *sendRcvdPacket flood*, instructing the switch to send the port using the action *flood*, which results in the switch forwarding the packet on every port except the incoming port.

```
floodPackets1 = proc evt → do
    packetInEvt ← packetIns ≺ evt
        returnA ≺ sendRcvdPacket flood packetInEvt
```

Elm [94], Flapjax [95], and Yampa [96] are examples of FRP languages for computer programming. FRP has been used in five studied languages [60], [61], [62], [69], [70].

D. Abstractions

An abstraction is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the user and suppresses details that are, at least for the moment, immaterial or diversionary [97]. Abstractions provide a good way to manage complexity [98]. The abstractions used in programming languages tend to emphasize the functional properties of the software – what is computed rather than how the computation is carried out.

[10]OpenFlow PacketIn message is sent from switch to the controller, when there is no match on switch flow tables.

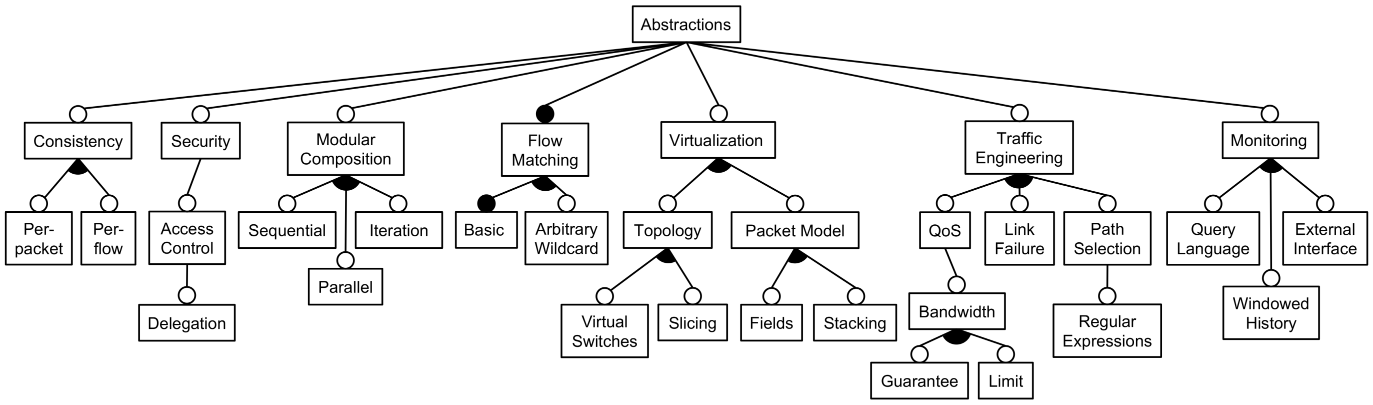

Fig. 15. Language abstractions feature diagram.

In a recent work, Casado et. al. have written about overall SDN abstractions [99] splitting them into the following areas: (1) network-wide structures provide abstractions to discover the topology, link fault detection, information about host locations, link capacities, and others. (2) Distributed updates propose abstractions with mechanisms that provide consistency guarantees during updates in network devices. (3) Modular composition allows programs to be decomposed into several modules. It also provides compositional functions (e.g., parallel or serial) to group these modules. (4) Virtualization decouples the application logic from the physical topology. This abstraction simplifies programs, ensures isolation, enhances scalability, and provides portability and fault-tolerance. (5) Formal verification provides tools for automatically checking formal properties and diagnosing user's network application problems. The application of formal methods in networking was exhaustively studied by Qadir and Hasan [78]. In their survey, they present several studies focusing on formal verification, including programming languages, Data Plane and Control Plane verification, and also debugging tools.

Some abstractions raised by Casado et. al. were implemented in the studied languages. *Figure 15* shows the abstractions reported in the studied languages. The next sections explain these abstractions, citing the languages implementing them.

Table II summarizes the abstractions provided by each language. The only language dealing with consistency updates is Frenetic-OCaml [70]. Some languages offer access control abstractions, allowing to define or delegate user permissions on network resources. Languages may also specify modular composition, allowing multiple modules to run sequentially, in parallel, or repeatedly (iteration). Abstractions to ease the flow matching were also proposed by some languages. Network virtualization can also be instantiated directly through languages constructs (virtual switches and slicing). Pyretic is the only language to propose virtual header fields and virtual stacking. High-level abstractions to implement QoS, deal with link failure, and simplify path selection were also found in some languages. Languages also provide abstractions to facilitate monitoring tasks, by proposing a query language, windowed history, and exposing an interface to deal with external events.

1) Consistency: This abstraction enables a controller to update the forwarding state of the entire network, ensuring a packet never traverse a path during a transition between two states. Every packet is either processed with the old configuration, or the new network configuration, but not a mixture of the two [99]. With consistency abstraction, programmers can bypass the problem of verifying program invariants across network updates. They just need to verify the initial and final states [100].

There are works focusing on consistency abstractions [101], [102], [103], [104] but, the consistency is performed automatically through mechanisms implemented in these solutions. Reitblatt et. al. [100], [105] have extended Frenetic [61], [106], including abstractions to support *per-packet* and *per-flow* consistency. A per packet consistent update guarantees that every packet flowing through the network is processed with exactly one forwarding policy. Analogously, per flow consistency guarantees that all packets in the same flow are handled by the same version of the policy. Frenetic-OCaml has maintained support to this abstraction.

2) Security: All surveyed languages present some basic security aspects for denying or allowing specific traffic; these basic aspects were not included in the taxonomy. The existing research can mostly be classified into two main categories: network access control and attack detection and defense [7]. Attack detection and defense is usually handled through specific SDN applications [107], [108], [109] or frameworks [110], [111], [112]. Network *access control* can be implemented through specific solutions [113], [114], but also can be done via programming language constructs.

Access control is the function of specifying access rights to resources, which is related to information security and computer security. More formally, "to authorize" is to define an access policy [115]. The access control abstractions allow defining what services or resources a user can access. For example, in a multi-tenancy data center, it is possible to define if a user has permission to add or remove a flow in a specific switch. Authorization is useful to allow tenants to modify global network policies to suit their demands. Among the researched languages, we have reported Merlin [116] and PonderFlow [65] providing authorization. Merlin also allows

TABLE II
LANGUAGES ABSTRACTIONS SUMMARIZATION

Feature		Language	FML [59]	Nettle [60]	Frenetic [61]	Procera [62]	Flog [63]	NetCore [69]	Frenetic-OCaml [70]	Pyretic [53]	FlowLog v1 [72], v2 [85]	FatTire [73]	NetKAT [71]	Merlin [57]	PonderFlow [65]	NoF [66]	Kinetic [74]	**Legend** X = Implements Natively * = Inherited
Consistency									P/F									P=Per-packet F=Per-flow
Security	Access Control													X	X			
		Delegation												X				
Modular Composition				S/P	S	S/P		S	S/P	S/P	S/P*	S/P*	S/P/I				S/P*	S=Sequential, P=Parallel, I=Iteration
Flow Matching	Basic		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X*	
	Arbitrary Wildcard				X			X	X	X			X				X*	
Virtualization	Topology	Virtual Switches								X			X					
		Slicing								X			X	X		X		
	Packet Model	Fields								X								
		Stacking								X								
Traffic Engineering	QoS		L											L/G			L	L=Limit G=Guarantee
	Link Failure											X						
	Path Selection		L									RE	RE	RE				L=Language Construct, RE=Regular Expression
Monitoring	Query Language				X		X		X		X							
	Windowed History				X	X		X	X	X			X	X			X*	
	External Interface		X			X		X			X					X	X	

delegating the authorizations hierarchically.

```
inst auth+  switchPolicyOps {
    subject  <User>  /NetworkAdmin;
    target <OFSwitch>  /switches;
    action addFlow(), removeFlow(), enable(), disable();
}
```

The previous example shows a PonderFlow [65] program for creating a positive authorization policy allowing all network administrators to add/remove flows and to enable/disable all switches in the network.

3) Modular Composition: Modularity is a useful technique for controlling the complexity of programs. Programs are decomposed into distinct modules with precisely specified and tightly controlled interactions. Separate modules are interesting for maintainability and code reuse. By using modularity, it is necessary to specify pathways for interaction among the modules to form a complete system [117]. Building modular SDN applications requires support for composition of multiple independent modules where each module partially specifies how traffic should be handled [55]. Modules can be understood as packet-processing functions that consume a packet history and produce a set of packet histories [118]. For grouping these modules, we have identified three different modular compositors: *sequential*, *parallel*, and *iteration*.

Sequential composition gives the illusion of one module operating on the packets produced by another. Given two policy functions f and g operating on a located packet p, sequential composition applies g to each of the located packets produced by $f(p)$, to produce a new set of packets [55]. For example, suppose the programmer has one firewall module, and another to route packets inside his network. These modules can be grouped with a sequential compositor so that the received packets will be initially processed by the firewall policies, and then be internally routed in the network. Seven languages have presented native support for sequential composition [60], [61], [62], [69], [70], [55], [71], and three have inherited it [72], [73], [74].

Parallel composition enables multiple policies operating concurrently on separate copies of the same packets. Given two policy functions f and g operating on a located packet p, parallel composition computes the multiset union of $f(p)$ and

$g(p)$ – that is, every located packet produced by either policy [55]. For example, one module monitors traffic by source IP address, while another one routes traffic by destination IP address. By using the parallel compositor, both route and monitor functions are performed simultaneously. Among the surveyed languages, this compositor was found natively in [60], [62], [70], [55], [71], and it was inherited by [72], [73], [74].

The *iteration* composition was found only in NetKAT [118]. This operator allows a function f to behave as the union of f composed with itself zero or more times. Once declarative languages do not implement loop statements, an iteration operator may be useful in some cases the function must be executed recursively, for example, a routing function.

4) Flow Matching: OpenFlow switches have flow tables, and these tables contain a set of flow entries. Each flow entry consists of match fields, counters, and a set of actions. Packet match fields are used for table lookups, and typically they include various packet header fields, such as TCP port, Ethernet source, or IP destination address. On receipt of a packet, an OpenFlow switch starts by performing a lookup in its flow tables. A packet matches a flow table entry if the packet header fields match those defined in the flow table entry [119]. SDN programming languages must provide matching abstractions; they were classified into two categories: *basic* matching and *arbitrary wildcard*.

A language is classified as *basic* matching when it supports the standard matching operations specified by OpenFlow. OpenFlow v1.0 provides specification to support exact match and prefix wildcard. In an exact match, the values of packet headers must be identical to those values defined in the match table. A match field may be wildcarded by its prefix (e.g., 192.168. $*$.$*$). In newer versions of OpenFlow (v1.1+), the switches may optionally support arbitrary wildcards by using bitmask (e.g., 192. $*$ .10.$*$). The bitmasks are defined so that a 0 in a given bit position indicates a “don’t care” match for the same bit in the corresponding field, whereas a 1 means to match the bit exactly [119]. Some proposed languages were classified as basic matching [59], [60], [62], [63], [72], [73], [57], [65], [66] because they implement only the matching abstractions offered by OpenFlow.

OpenFlow v1.0 compliant switches do not support arbitrary wildcards in matching operations. Some languages present abstraction to *arbitrary wildcard* [61], [69], [70], [55], [71], [74], even if the hardware does not support it. If the switch has basic support for wildcards, then the compiler needs to generate a policy that may be larger than would be practical. For example, this wildcard 192.$*$.10.$*$ should be converted in 256 flow entries from 192.0.10.$*$ until 192.255.10.$*$. Thus, the compiler generates overapproximations that match a superset (192.$*$.$*$.$*$) of the traffic specified by the original predicate and uses the reactive strategy (*Section III-A*) to send these packets to the controller [69].

5) Virtualization: This abstraction refers to the act of decoupling the service (logical) from its realization (physical) [42]. In computer science, virtualization has been used since the mainframes era. The concept of multiple coexisting logical networks has already appeared in networking, presented as Virtual LANs (VLANs), Virtual Private Networks (VPNs), Active Networks [120], and overlay networks [121].

According to Jain and Paul [11], virtualization is useful to provide: (1) Sharing – when a resource is overestimated for a single user, then it might be better to divide it into multiple virtual ones. (2) Aggregation – if the available resources are small, it is possible to create a larger virtual resource by grouping the small ones. (3) Isolation – in a shared network, enables to provide virtual “isolated slices” for the different users, preventing their information from being accessed. (4) Dynamics – enables the resources to be easier relocated, and (5) easier to manage – virtual devices are easier to manage because they are software-based and expose a uniform interface through standard abstractions. With the advent of SDN, this concept grew widely, and it is being used to virtualize many network elements such as ports, cards, switches, links, and functions (NFV) [122]. It is also used to virtualize topologies by mixing virtual and physical devices in an abstract topology. In our research, we have organized these abstractions offered by languages into two groups: *topology* and *packet model*.

Network virtualization allows multiple virtual networks to run over a shared infrastructure, and each virtual network can have a much simpler (more abstract) topology than the underlying physical network [123]. The abstract *topology* may consist of a mix of physical and virtual switches, and may have multiple levels of nesting on top of the real network [55]. Existing languages propose two ways to virtualize the topologies. The first topology abstraction, also named one-to-many or multiplexing [124], allows the programmers to create multiple *virtual switches* in the same physical device. It usually is accomplished through software switches [54], [125], [126], but this abstraction is also available in two languages [55], [118].

The second topology abstraction has been named *slicing*. It has different designations in the studied papers: big switch, aggregation, overlay networks, many-to-one, and slicing. In our taxonomy, we have grouped these concepts, understanding that there is no significant difference among them to justify such separation. This abstraction allows the network to be “sliced”. A slice is composed of a subset of switches, ports, and links. The packet processing on each slice is dictated exclusively by the program for that slice, not being affected by the programs for any other slices (isolation). By slicing a network, it is possible to define which packets (or flows) will be redirected to which slice, enabling, for example, different applications to be routed through different slices. There exist many different approaches providing slicing in SDN [41], [43], [127], [30], [128]; among the studied languages, we identified four languages [55], [71], [64], [66] with this abstraction.

The *packet model* abstraction was found only in Pyretic. In this language, each packet flowing through the network is a dictionary [129] that maps field names to values. The dictionaries include entries for the packet location (either physical or virtual), standard OpenFlow headers (e.g., source IP, destination IP, source port), and the virtual packet fields. Programmers can also add new virtual *fields* on packets header, and these fields are not limited to simple bit strings, they may also be arbitrary data structures. According to the authors, this extension provides a general way to associate high-level information with packets and to enable coordination among modules. In addition to extending the packet by including virtual fields, abstract packet model also extend packets by allowing every field (including non-virtual ones) to hold a stack of values instead of a single bit string. $Stacking$ simulate a packet traveling through multiple levels of abstract networks. For example, to "lift" a packet onto a virtual switch, the run-time system *pushes* the location of the virtual switch onto the packet. Having done so, that virtual switch name sits on top of the concrete switch name. When the packet leaves the virtual switch, the run-time system *pops* a field off the appropriate stack.

6) Traffic Engineering: One can say that traffic engineering is concerned with performance optimization of operational networks. In general, it encompasses the application of technology and scientific principles to the measurement, modeling, characterization, and control of network traffic. Traffic engineering also includes the application of such knowledge and techniques to achieve specific performance objectives [130]. Routing optimization plays a key role in traffic engineering, *i.e.*, finding efficient routes to achieve the desired network performance [131]. Quality of service (QoS) and resilience schemes were also considered as major components of traffic engineering [23].

Kreutz et. al. [9] have identified many studies focusing on traffic engineering, most of them were classified as SDN applications. These works aim on different aspects of traffic engineering, such as QoS, load balancing, link failure, and energy saving. Akyildiz et. al. classified the traffic engineering abstractions in a broader scope: flow management, fault tolerance, topology update, and traffic analysis/characterization. We have encountered languages providing abstractions to deal with QoS, $link\ failure$, and $path\ selection$.

The Quality of Service (QoS) can be defined as a set of service requirements to be met by the network while transporting a connection or flow [132]. These requirements can be expressed in terms of integrity constraints (packet loss), temporal constraints (delay), and timing restrictions for the delivery of consecutive packets belonging to the same traffic stream (delay variation) [133]. In our survey, we have perceived languages providing constructs to limit the bandwidth in terms of minimum and maximum usage. Minimum bandwidth allocates, to a specific traffic, a *guarantee* that the bandwidth will be more than the minimum specified value. Two languages have implemented constructs to guarantee bandwidth: FML [59] and Merlin [57]. Maximum bandwidth is also named "bandwidth cap", and it allows the programmers to specify the maximum *limit* for certain flows. It was found only in Merlin.

One objective of traffic engineering is to simplify reliable network operations [130]. Reliable network operations can be facilitated by providing mechanisms that enhance network integrity, and by embracing policies emphasizing network survivability [133]. Network survivability refers to the capability of a network to maintain service continuity in the presence of faults. Some studies propose fault-tolerance on SDN-based networks [134], [135], [136]. FatTire [73] is the only studied language that provides abstractions to quickly change the forwarding behavior in cases of $link\ failure$. FatTire allows specifying sets of legal paths through the network, including the number of backup paths. However, it can handle only link-level failures, not supporting device failures. FatTire compiler uses fast-failover groups to calculate every possible failure and precomputes appropriate backup paths. Fast failover groups were introduced in OpenFlow v1.1+ to enable the switch to change forwarding without requiring a round trip time to the controller [119].

The last traffic engineering abstraction was called $path\ selection$. It enables programmers to define constraints on network paths, allowing to specify which paths the flows must or must not pass through the network. Path selection is being used in SDN networks to optimize load-balancing routing [137] and QoS [138]. The abstraction was primarily implemented in FML [59] by means of two language constructs: $waypoint$ requires a flow to pass through a particular node in the network and $avoid$ that forbids a flow from passing through a particular node.

A regular expression is a specific text string used for describing a searching pattern. It is often used for string matching, but it is also a natural and well-studied mathematical formalism for describing path selection through a graph [139]. Some network languages use $regular\ expression$ to declare path constraints on sequences of network locations [73] or packet transformations [71], [64].

$$\begin{array}{c}(tpDst\ =\ 22\ \Rightarrow\ [*.IDS.*])\\ \uplus\ (any\ \Rightarrow\ [GW.*.A])\end{array}$$

This example is a FatTire program composed of two components. A security policy, given by the first line, which states that all SSH traffic must traverse the IDS; and a routing policy, given by the second line, which states that traffic from the gateway (GW) must be forwarded to the access switch (A), along any path.

7) Monitoring: One advantage of programming the network is the possibility to change its behavior due to some event, for example, changing the routing because a link becomes overloaded. To implement this, it is necessary to read information from the network. Some languages present a specific $query\ language$ to gather the network data. Other languages permit to query the network considering a specific period of time (e.g., last 5 minutes). We have used the term $windowed\ history$ for these window-based abstractions. Besides the languages, there are publications related to monitoring the network information [140], [141], [142], [143].

Some languages present a specialized subset of constructs to query network information. This $query\ language$ allows the programmers to express the statistics needed in their programs. Query language raises the level of abstraction and frees the programmers from worrying about the way the query rules are installed on the switches. It also allows the program to be split in querying information and policy specification. Query languages include constructs for filtering a set of packets in the network, grouping results by header fields, aggregating by number or size of packets, and limiting the number of values returned. This grouping is interesting, for instance, to instruct the switches to redirect certain flows to specific network functions such as Deep Packet Inspection (DPI). Some surveyed languages provide a sublanguage specifically to query the network [61], [63], [70], [85].

Another abstraction allows *windowed history* queries on the network. These windows can be defined in terms of time, number of packets, or packet size. Many programs need to receive periodic information about traffic statistics, for example, the programmer may use a windowed query to collect information minutely. If the language does not support this abstraction, it would be necessary to check manually switch counters, and registering triggers to inquire the counters repeatedly every minute.

```
Select(bytes)
GroupBy([srcip])
Every(60)
```

This program, written in Frenetic [61], [106], looks at all traffic, groups by source IP address, and aggregates the number of bytes every 60 seconds. This abstraction was found in eight programming languages [61], [62], [69], [70], [53], [71], [116], [74].

We have named as *External Interface* the ability of the SDN languages to access information beyond the scope of the language. This information can include external references to SQL queries over databases, hash tables, and arbitrary code written in another language. It can also include external events originated, for example, by intrusion detection systems or authentication servers. We have identified six languages that explicitly claim to provide support to access external information and events [59], [62], [69], [85], [66], [74].

IV. SDN Programming Languages

A comprehensive review of to date SDN programming languages is presented in this section. The languages have been described in chronological order, and their evolutions and relationships are reported. The main contributions and innovations have also been highlighted and referenced to the proposed taxonomy. Some programs have been exhibited in order to illustrate the contributions, as well as to expose the language syntax.

Figure 16 shows the genealogy of these languages. Each node represents a programming language. The relationship among them was based on the information contained in their papers, websites, and source code repositories. The dates were also estimated considering these three information sources. A language has continuity line if it has been renamed or if it provides a source code repository; in this case we have also plotted its last commit date. If a language appeared only once (publication date), it was represented by just the node, without the continuity line.

One can perceive that there are two peculiar situations: (1) FlowLog [72] was initially implemented on top of NetCore [69], but at a given time, Frenetic-OCaml [70] replaced NetCore. The approximate time of this modification is shown in genealogy with an arrow. The same happened with the (2) Frenetic-OCaml, which replaced the NetCore by NetKAT. We believe that these changes occurred due to the discontinuity of NetCore.

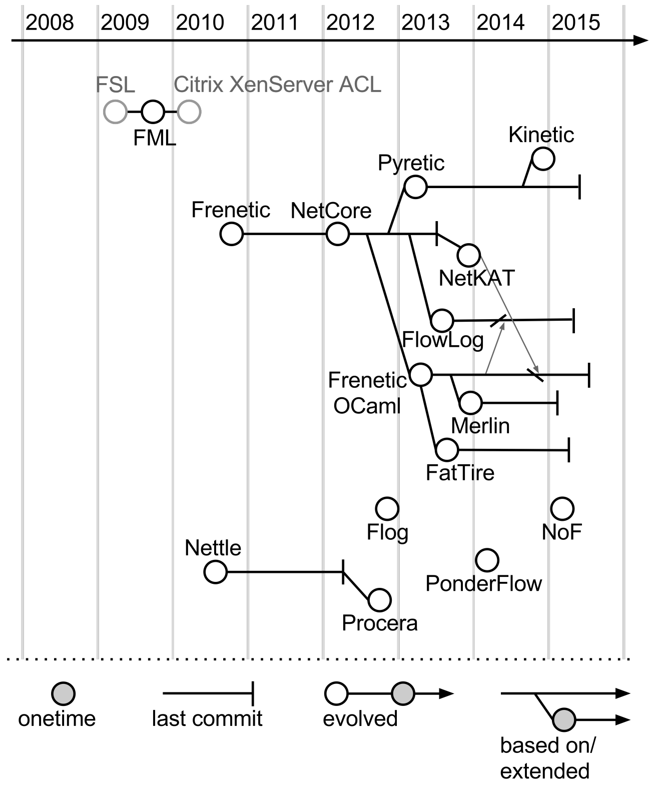

Fig. 16. Genealogy of SDN languages.

Table III provides an overview of the languages development state. It includes the license of open-source languages, in which language they were implemented, their repository, the last commit date, the number of commits, branches, releases and contributors. This table along with *Figure 16* provide relevant information to assist researchers in choosing a language to best fit their needs.

A. Flow-based Management Language

Flow-based Management Language (FML) is the first SDN programming language. It provides a high-level declarative policy language based on logic programming (*Section III-C2*). It relies on non-recursive Datalog with negation [83], where each statement represents a simple *if-then* relationship. In a technical report, the language was initially defined as Flow-based Security Language (FSL) [144], however, in the following publication, it was renamed to FML [59]. Information available on the author's webpage[11] shows that it was sold to Citrix Systems to be used as the Access Control Language (ACL) on Citrix XenServer. In the genealogy (*Figure 16*), we have represented FML in black, and the other designations were plotted in gray.

This language provides abstractions to statically allow and/or deny specific flows. An example of FML program expressing this abstraction was shown in *Section III-B*. FML also includes abstractions to redirect or avoid the flows to pass through (or avoid) a specific host, and to impose bandwidth

[11]http://www.cs.uic.edu/~hinrichs/projects.htm

TABLE III
SDN Programming Languages Development Information

Language	License	Implemented in	Repository	Commits	Branches	Releases	Contributors	Last commit
FML [59]	—	C++, Python	—	—	—	—	—	—
Nettle [60]	SD-3-Clause	Haskell	https://github.com/AndreasVoellmy/nettle-openflow	30	1	0	1	Jul 27, 2015
Frenetic [61]	—	Python	—	—	—	—	—	—
Procera [62]	—	Haskell	—	—	—	—	—	—
Flog [63]	—	—	—	—	—	—	—	—
NetCore [69]	BSD-3-Clause	Haskell	https://github.com/frenetic-lang/netcore-1.0	598	4	1	9	Nov 3, 2014
Frenetic-Ocaml [70]	LGPL-3.0	OCaml	https://github.com/frenetic-lang/frenetic	4781	36	12	31	Dec 14, 2015
Pyretic [53]	BSD-3-Clause	Python	https://github.com/frenetic-lang/pyretic	1708	18	1	14	Aug 29, 2015
FlowLog v2 [85]	LGPL-3.0	OCaml	https://github.com/tnelson/FlowLog	1025	1	1	3	Apr 10, 2015
FatTire [73]	LGPL-3.0	OCaml	https://github.com/frenetic-lang/fattire	754	1	0	5	Mar 21, 2015
NetKAT [71]	LGPL-3.0	OCaml	https://github.com/frenetic-lang/netkat	61	3	0	3	Dec 3, 2013
Merlin [57]	LGPL-3.0	OCaml	https://github.com/merlin-lang/merlin	5	1	0	1	Jan 21, 2015
PonderFlow [65]	—	Java	—	—	—	—	—	—
NoF [66]	—	Python	—	—	—	—	—	—
Kinetic [74]	BSD-3-Clause	Python	https://github.com/frenetic-lang/pyretic/tree/kinetic	759	6	1	14	Sep 26, 2014

limits on those flows (*Section III-D6*). It also allows policies to be written by using external information sources (*Section III-D7*). Parameters to specify maximum values for latency, jitter and bandwidth have been described in the language. However, they were not implemented, so they were not considered in the taxonomy.

B. Nettle

Nettle [60] was the second SDN programming language. Its programming paradigm is based on the principles of Functional Reactive Programming (FRP). The *Section III-C2* has posed an example of a Nettle program reacting to $packet\_in$ OpenFlow events. According to the authors, FRP provides an elegant way to express computation in event-driven domains. The key ideas and constructs of Nettle are strongly influenced by Yampa [96], an FRP-based language used in robotics and animation.

Nettle supports discrete (event-based) and continuous (time-varying) operations. Event-based allows it to capture communication patterns to and from OpenFlow switches. The continuous behavior enables the creation of dynamic policies (*Section III-B*), for example traffic engineering – dynamic load balancing. Nettle has introduced the parallel and sequential composition operators (*Section III-D3*).

C. Frenetic

The name Frenetic was originally used as the name of a programming language. Over time, it has been used as the name of a project to represent a set of SDN programming languages. Lately the name Frenetic was used again to specify the name of a new language. To avoid misunderstanding, we have called this second version of Frenetic as Frenetic-OCaml. As shown in *Figure 16*, most of the existing SDN programming languages are derived from Frenetic.

Frenetic [61], [106] provides a declarative SQL-like query language (*Section III-D7*) for classifying and aggregating network traffic, as well as a functional reactive (*Section III-C2*) combinator library for describing high-level packet-forwarding policies. The combinator library is inspired on Yampa [96] and its implementation is based on FlapJax [95].

The language provides compositional constructs which facilitate modular reasoning and enable code reuse. Frenetic has innovated by providing arbitrary wildcard (*Section III-D4*) and windowed history for grouping information by time, number of packets, or packet size. *Section III-B* has presented an example of host usage being collected every minute. This example is implemented using the windowed history construct $Every$.

D. Procera

As well as Nettle and Frenetic, Procera [62] is based on FRP, using ideas from Yampa [96]. Despite no information was confirmed that Procera might be an extension of Nettle, given that they were proposed by the same author with similar structural characteristics, then we have considered it in the genealogy as being based on Nettle. Procera was used in various prototype deployments in campus and home networks [145].

The language incorporates external events originated from sources other than OpenFlow switches, allowing it to express policy that reacts to conditions such as user authentications, time of day, bandwidth use, or server load. Procera also provides a collection of signal functions and abstract data types that allow programmers to describe policies by using windowed event histories (*Section III-D7*). Both Frenetic and Procera have windowed history abstraction, but Frenetic applies this operation only to network packets or flows, while Procera allows windowed history to be applied to any event stream (OpenFlow or external) [62].

The following code is a Procera program using event history abstraction and external events. An external system collects the host usage logs every few seconds. The $usageEvents$ is an event created to filter just the usage events from all existing events (*world*). Once filtered, a sliding window of five days is used to create a device usage mapping. Lastly, a grouping operation is performed, and the module returns a sum of all usage.

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

E. Flog

Flog [63] combines ideas from FML and Frenetic. It uses the same idea of FML by proposing a logic programming language to control SDN. From Frenetic, it adopts the idea of dividing the language in three components: a mechanism for querying network state, a mechanism for processing data gleaned from queries (or other sources), and a component for generating rules to be installed on the network switches. The authors believe that "logic programming is good for this domain because of the success of FML, and because so much of SDN programming is table-driven collection and processing of network statistics".

Flog is the first language to be designed as stateful using event-driven (*Section III-C2*) and logic paradigms. Each time a network event occurs, the Flog program is executed. The execution of the Flog program has two effects: it generates some state(s) that may be used to drive the program in future network events, and it generates the policies to be deployed on switches. Flog implements two event databases: a current database whose values are used in the current execution and a future database that will be used the next time the Flog program is executed (on the subsequent network event). Flog's next state is empty unless the values are explicitly carried over.

```
# Network Events
flow(scrip = IP, inport = P) --> seen(IP, P)
# Information Processing
seen(IP, P) +-> learn(IP, P).
learn(IP, P) +-> learn(IP, P).
# Policy Generation
|> flood, level(0).
learn(IP, P) --> dstip(IP) |> fwd(P), level(1).
```

The previous code is a Flog program for a simple learning switch. The program monitors all the packets arriving at the switch, and groups them by source IP and *inport*, storing this information in the *seen* relation. In the processing phase, the information is transferred to the persistent *learn* database. The policy generation phase first generates a low priority rule that unconditionally floods all packets that arrive at the switch. Then, based on information it has learned, it generates higher priority rules that perform more precise forwarding. Specifically, for every *learn*(*IP*, *P*) fact in the persistent database, the program generates a forwarding rule that directs packets with destination IP out port *P*.

F. NetCore

NetCore [69] was defined as the replacement of Frenetic [61], [106] for expressing forwarding policies. This language has been used as the core language in the new version of Frenetic, written in OCaml, and also in Pyretic. According to the authors, NetCore is at the forefront by presenting formal semantics and proofs of correctness.

The following NetCore program acts as a query over the network traffic history, filtering all web traffic (*DstPort* = 1010000) and returning *true* if the total number of web packets sent is less than 10000, or if the packet *p* comes from a particular sender (*SrcAddr* = 10.0.0.1).

```
filterWeb = DstPort : 1010000
cond (∑, s, p) = cardinality ∑ 10000 ||
                 p.SrcAddr == 10.0.0.1
```

One contribution of NetCore is to generate rules proactively (*Section III-A*), instead of strictly reactively. Its compiler generates as many OpenFlow-level rules proactively as possible, otherwise it uses the reactive method. There are three main situations where the NetCore compiler cannot proactively generate all the rules it needs to implement a policy: (1) the policy involves a query that groups by IP address (or other header field) – such a query would require one rule for each of the 2^{32} IP addresses if generated ahead of time, but only one rule for each IP address that actually appears in the network if unfolded dynamically; (2) the policy involves a function that cannot be implemented natively on the switch hardware; (3) the policy involves a primitive pattern that cannot be implemented efficiently on the switch hardware [70].

G. Frenetic-OCaml

Frenetic-OCaml [70] is a new version of the Frenetic language [61], [106], implemented in OCaml. It was initially used NetCore as its core language for forwarding decisions, however NetCore was replaced by NetKAT [71]. The language maintains the network query language from the previous version of Frenetic. The query language allows reading information about network state, including traffic statistics and topology changes. The run-time system handles the details of polling switch counters, aggregating statistics and responding to events. As well as its core language, it implements proactive rule installation.

Frenetic-OCaml is the only language to provide constructs to ensure, during an update, that all packets (or flows) will be processed with an old policy or the new policy, and never a mixture of them. It implements per packet and per flow consistency (*Section III-D1*). At the time of its publication, the language only provided the sequential composition operator. However the information on its repository shows that the parallel composition has been also implemented (*Section III-D3*).

```
def repeater() :
    rules = [Rule(inport : 1, [fwd(2)]),
             Rule(inport : 2, [fwd(1)])]
    register(rules)
def web_monitor() :
    q = (Select(bytes) *
         Where(inport = 2 & srcport = 80) *
         Every(30))
    q >> Print()
def main() :
    repeater()
    monitor()
```

The code above shows the composition between two modules: *repeater* and *web_monitor*. The *repeater* implements the rules to forward the traffic, while the *web_monitor* query the amount of HTTP traffic arriving on ingress port 2. The *main* module simply assembles these modules into a single program.

H. Pyretic

Pyretic [53], [55] is a language of the Frenetic Project. It is the only SDN language implemented with the imperative programming paradigm. A Pyretic program is shown in *Section III-C1*. It provides abstrations that are embedded in Python. Pyretic uses an extension and generalization of the NetCore for specifying static and dynamic forwarding policies.

The language allows to combine multiple policies together using parallel and sequential composition operators. Pyretic integrates monitoring function through a query language just like Frenetic. Pyretic also offers a library for topology abstraction that can represent multiple underlying switches as a single derived virtual switch or, alternatively, one underlying switch as multiple derived virtual switches. The virtualization policies can be composed together with other policies, or used as the basis for yet another layer of virtualization. Pyretic also introduces an Abstract Packet Model that allows the creation of virtual packet header fields. It also holds a stack of packet field headers, instead of a single bit string (*Section III-D5*).

I. FlowLog

FlowLog [72], [85] was presented in two versions. The first version (for short v_1), was implemented on top of NetCore. The second version (v_2) uses Frenetic-OCaml for packet-handling. Both language were grouped in one single section because the most significant change has been in their syntax. The authors have implemented FlowLog as a finite state language, so it can be model checked efficiently. The language proposes a tierless abstraction for managing the three SDN tiers: Control Plane, Data Plane, and controller state. According to the authors, tierlessness simplifies the process of SDN programming and simultaneously enables cross-tier verification. FlowLog compiler presents a preprocessing function that identifies which rules should be sent to the controller (*Section III-A*). According to the authors, the only packets the controller needs are those that probably alter the controller state.

The syntax of FlowLog v_1 [72] was inspired by Datalog non-recursive with negation [83]. In this version, the programs consisted of a set of rules, where the programmer must explicit the controller internal state, *i.e.*, the rules to be learned ($+learned$) and forgotten ($-learned$) or a packet handling action (to send the rules to the switches). FlowLog v_2 [85] was inspired by the ANQL [146], changing its syntax to a SQL-like. This version allows the creation of internal tables (controller states) and external tables (callouts to external code). Programs are composed of a set of rules. When a new event is triggered, actions may be executed to $INSERT$ or $DELETE$ (similar operations to $+learned$ or $-learned$) rules into the controller state, or to DO an action such as forwarding a packet. Rules and triggers have an optional $WHERE$ clause, to additional constraints. An example of FlowLog program is shown in *Section III-C2*.

J. FatTire

FatTire [73] is a language for writing fault-tolerant network programs. The central feature of this language is a new abstraction that allows developers to specify the set of paths that packets may take through the network, as well as the degree of fault-tolerance required. This construct is implemented by a compiler that targets the in-network fast-failover mechanisms provided by OpenFlow v1.1+ standard. This mechanism automatically responds to link failures without controller intervention. FatTire uses regular expressions to allow programmers to declaratively specify sets of legal paths through the network, as one can see in the program presented in *Section III-D6*.

Compilation of a FatTire policy proceeds in four phases: It (1) normalizes the input policy to a union of atomic policies (non-overlapping predicates). Then, it (2) constructs a fault-tolerant forwarding graph for each atomic policy. After that, it (3) translates the forwarding graphs to policies in Frenetic-OCaml[12]. Last, it (4) compiles the resulting policies to OpenFlow using an extension of the Frenetic-OCaml compiler using fast-failover groups. Because FatTire compiles into Frenetic-OCaml, its programs can be used as ordinary Frenetic-OCaml programs, and can be combined using its parallel and sequential composition operators.

K. NetKAT

NetKAT [71], [118], [147], [148] is based on Kleene algebra with Tests (KAT) [149] for specifying, programming, and reasoning about networks. NetKAT semantics was inspired by NetCore, but it was extended to make it verifiable for KAT sound (*i.e.*, no false positives) and complete (*i.e.*, no bugs are missed). NetKAT was the firt language to use regular expressions to describe end-to-end network forwarding paths (*Section III-D6*).

The example below shows a simple firewall implemented in NetKAT. The firewall allows HTTP traffic between hosts with IP addresses 10.0.0.1 and 10.0.0.2. Any other traffic is dropped.

```
let firewall : policy =
  <:netkat <
  if ( ( ip4Src  =  10.0.0.1 &&
         ip4Dst  =  10.0.0.2 &&
         tcpSrcPort  =  80 )    ||
       ( ip4Src  =  10.0.0.2 &&
         ip4Dst  =  10.0.0.1 &&
         tcpSrcPort  =  80 ) )
  then $forwarding
  else drop
```

NetKAT can be used to specify virtual topologies (*Section III-D5*) [118]. NetKAT has replaced NetCore as the intermediate language in Frenetic-OCaml. It introduces a new operator

[12]In FatTire paper, authors has used NetCore as intermediate language but, the comments in the source repository specifies it was migrated to Frenetic-OCaml

for iteration, allowing the policy to be repeated zero or more times (*Section III-D3*). This operator can be used, for example, to encode the end-to-end behavior of the network (forwarding policies).

Ryan at. al. [150] have extended NetKAT with past-time temporal operators $last(a)$, which asks if a is true at the previous step in the history, and $ever(a)$, which asks if a was ever true at any point in the history. In a recent technical report, Foster et. al. have proposed the Probabilistic NetKAT [151], which enriches NetKAT semantics to denote functions that yield probability distributions on sets of packet histories. It adds primitives such as probabilistic choice, which makes it possible to handle scenarios involving congestion, failure, and randomized forwarding. McClurg et. al. [152] have introduced stateful features in NetKAT, allowing to specify static configurations and configuration changes triggered by events at the switches. Their proposal also incorporates event-triggered consistent updates motivated by the need for the network to react to events immediately, while avoiding expensive synchronization that would entail buffering of packets.

L. Merlin

Merlin [57], [116], [64], [153] is a framework for programming network policies. It was implemented on top of Frenetic-OCaml and it includes a declarative language, infrastructure for distributing, refining, and coordinating enforcement of policies, and a run-time monitor that inspects incoming and outgoing traffic on end-hosts. The declarative language provides native constructs for specifying bandwidth limit (max) and guarantee (min). One example of this abstraction is shown in *Section III-B*. As well as NetKAT, it also uses regular expressions to encode forwarding paths (*Section III-D6*).

Merlin is a pioneer language to provide forwarding decision control to tenants, specified as sub-policies (*Section III-D2*). Merlin also offers a mechanism for verifying if the sub-policies do not violate global constraints. Another innovation is to allow not only switches to be programmed but also the policies specified are compiled generating low-level instructions to program a variety of elements, including middle-boxes, and end-hosts.

M. PonderFlow

PonderFlow [65] is an extension of Ponder language [154]. Ponder is a language for specifying management and security policies for distributed systems. The Ponder language does not support flow abstraction, so PonderFlow extends it by allowing to define OpenFlow flow rules. PonderFlow provides only constructs to allow access control, implementing authorizations and obligations abstractions.

The authorization policies define what the members of a group may or may not do in the target objects. Essentially, these policies define the level of access that users possess to use an OpenFlow switches network. One example of this feature is shown in *Section III-D2*. Obligation policies allow specifying actions to be performed by the network administrator, or by the OpenFlow controller when certain events occur in the network and provide the ability to respond any change in circumstances.

N. NoF

Network Overlay Framework (NoF) [66] is a framework that enables networks to be defined according to the application (software) requirements. It provides a programming language to allow application specialists to program the network. NoF implements a module to translate NoF programs to any SDN language or controller's API. NoF also provides an interface to execute automatically commands to configure and read end-host information.

The NoF language is composed of sets of operations and services. Operations were divided into three groups: (1) Matching operations can be done on traditional network fields, and also on host information (e.g., hostname, system load, bandwidth usage, process name). (2) Timing operations allow to specify when (date/time) the services will be installed and for how long they will act on the network. (3) Query operations allow reading network state (e.g., link state, transmission errors and bandwidth usage). Services define the modifications that will be implemented on the network. The current implementation provides services to ensure bandwidth guarantees, to manage virtual overlay networks, and redirect specific flows through these overlaid networks.

```
NoF : {
operations(process_name = ("hadoop", "hdfs")),
services(virt_topo = ("spanning_tree_1"));
}
```

In this example, NoF configures the hosts to start monitoring new connections on processes named *hadoop* and *hdfs*. Upon these processes receive new connections, the network is reprogrammed to redirect, only these new flows, through the virtual topology called *spanning_tree_*1.

O. Kinetic

Kinetic [74] provides a language for expressing a network policy in terms of finite state machines (FSMs). According to the authors, FSM permits to concisely capture control dynamics in response to network events and are amenable to verification [155]. States correspond to distinct forwarding behavior, and events trigger transitions between states. Kinetic's event handler listens to events and triggers transitions in policy, which in turn update the Data Plane. Kinetic is based on Pyretic, inheriting its runtime features. Specifically, Kinetic uses Pyretic's composition operators to express larger FSMs as multiple smaller ones that correspond to distinct network tasks (e.g., authentication, intrusion detection, rate-limiting).

FlowLog [85] also uses finite state machines reacting to events, but it implements programs in a single FSM. Kinetic allows encoding generic FSMs that can be applied to any group of packets (e.g., all packets from the same host). Each group of packets has a separate FSM instance; packets in the same group will always be in the same state. These groups of packets were called as Located Packet Equivalence Class (LPEC). Kinetic instantiates copies of programmer-specified FSMs (one per LPEC); the Kinetic event handler sends incoming events, which can arrive either from external event hookups or from the Pyretic runtime (e.g., in the case of certain types of events such as incoming packets), to the appropriate FSMs.

```
1.  @transition
2.  def infected(self) :
3.      self.case(occurred(self.event), self.event)
4.  @transition
5.  def policy(self) :
6.      self.case(is_true(V('infected')), C(drop))
7.      self.default(C(forward))
8.  self.fsmdef = FSMDef(
9.      infected = FSMVar(type = BoolType(),
10.         init = False,
11.         trans = infected),
12.     policy=FSMVar(
13.         type=Type(Policy, {drop, forward}),
13.         init = forward,
14.         trans = policy))
15. def lpec(pkt) :
16.     return match(srcip = pkt['srcip'])
17. fsmpol = FSMPolicy(lpec, self.fsmdef)
```

This code is a Kinetic program that implements the simple intrusion detection. The FSM has two states, (1) host is infected, and its traffic is dropped, and (2) when it is not infected, and its traffic is forwarded. Each source IP address should have a distinct FML, so it is necessary to create an LPEC per source IP address (lines 15-16). This FSM has two variables: *infected* and the corresponding *policy*. When *infected* is *false*, the *policy* is set to *forward* the traffic from that host. Upon receiving an event saying the host is infected, the variable *infected* is set to *true* and the *policy* is changed to *drop* the host's traffic. When receiving another event saying the host is not infected, *infected* receives *false* and *forwarding* is assigned to the variable *police*, allowing the network to forward the traffic again.

P. Another Northbound Abstractions

This section presents some other works offering northbound abstractions. We have selected some relevant efforts that, despite not having the language as a main aspect, propose frameworks or APIs to raise the level of abstraction in different areas. These works are succinctly described, ordered by publication date, exposing their key features, suitable with this survey.

Gutz et. al. [156] have proposed network isolation at the language level. They have created a library in Python that allows to instantiate virtual topologies through Mininet[13], and to define isolated slices on top of these topologies. Different SDN applications can be associated with different slices. PANE [157], [158] provides a Northbound API for delegating privileges for users to request network resources (e.g., bandwidth or access control). It uses a hierarchy conflict resolution supplied by HFT [159].

Trema [160] is a framework covering the entire development cycle of programming, testing, debugging, and deployment. FRESCO [111] is another framework, but focusing on developing and deploying security services. It provides some security functions, such as firewalls, scan detectors, attack deflectors, and IDS. Maple [161] is also a framework that, through an API, allows the programmer to use a standard programming language for programming OpenFlow switches. Maple optimizer observes the algorithm execution traces, organizes these traces to develop a partial decision tree. After that, it compiles these trees into optimized flow tables for distributed switches. Maple includes the McNettle OpenFlow network controller [162] that efficiently executes user-defined OpenFlow event handlers on multicore CPUs. Maple could be considered as an imperative programming language just like Pyretic, but the authors have focused on other aspects, hindering us to classify it as a language.

[13]http://mininet.org/

NVP [44] is a network virtualization platform to manage virtual networks in multi-tenant data centers. It uses a declarative language called nlog for computing the network forwarding state. NVP users can not use this language, it was used only internally to develop NVP. Path-queries [163] is a query language implemented on top of Pyretic. It uses a regular-expression-based path language that includes SQL-like *groupby* constructs for counting aggregation. It provides some useful applications, including single path packet count, traffic matrix generation, and congested link detection.

Concurrent NetCore [164] was inspired in NetCore [69]. It is a programming language allowing to describes the layout of switches flow tables. Currently, the switches do not have programmable tables, but P4 has proposed them to be configured according to the user's needs [165]. The language acts mostly in CDPI layer, providing a small number of northbound operations for specifying packet processing, plus modular composition operators. The language introduces the concurrent compositor. While parallel compositor makes copies of packets, and then performs the actions on theses copies, the concurrent compositor allows two policies to act simultaneously on the same packet, so non-conflicting actions may be executed concurrently to reduce packet processing latency.

V. Summarization and Open Issues

In this section, we have summarized the SDN languages presented in this survey. Languages were categorized into two main groups as follows: (1) *specific purpose* languages, aiming to solve a particular problem, providing specific operations; and (2) *general purpose* languages that allow a more general and wider set of operations. Such a division has enabled us to better identify how languages have evolved, regarding the studied features.

Table IV presents, in chronological order, the main contributions introduced by SDN languages. These contributions have been discussed in details in *Section III*. The *specific purpose* languages: FatTire [73], Merlin [57], PonderFlow [65], and NoF [66], were identified with an asterisk ($*$) in this table.

FatTire has been created for writing fault-tolerant network programs. The language permits to specify allowable backup paths to be used in case of link failure. Merlin has been designed to allow programmers to set the intended behavior of the network in terms of legal paths and bandwidth requirements (limiting and guaranteeing). The language provides constructs to support time-based windows and aggregation. It also

TABLE IV
MAIN CONTRIBUTIONS INTRODUCED BY THE SDN PROGRAMMING LANGUAGES

Language	Main Contributions
FML [59]	Static policies External events Bandwidth limits Path selection
Nettle [60]	Parallel and sequential module compositions Dynamic policies
Frenetic [61]	Windowed packet/flow history Specific language to query the network (SQL-like)
Procera [62]	Windowed event history
Flog [63]	Event-driven programming paradigm
NetCore [69]	Proactive rule instalation Network slicing
Frenetic-Ocaml [70]	Consistent updates
Pyretic [53]	Imperative programming Abstract packet model (fields + stacking) Topology virtualization (virtual switches and slicing)
FlowLog v1 [72]	Finite-state language, amenable to model-checking
FatTire [73] *	Fault-tolerance to support link failure
NetKAT [71]	Uses Kleeve Algebra with Tests (KAT) Iteration compositor
Merlin [57] *	Abstractions to program end-hosts and middleboxes Access control and delegation
PonderFlow [65] *	Specify language to provide access control
NoF [66] *	Raise abstraction level to application specialists
Kinetic [74]	Generic Finite State Machines (FSMs) Located Packet Equivalence Class (LPEC) FSM composition

* *specific purpose* language

includes mechanisms for delegating sub-policies and verifying if delegated sub-policies do not violate global constraints.

PonderFlow has been designed to specifically provide access control. Its constructs enable programmers to define which users can access and modify switches information, as well as, add or remove rules in their flow tables. NoF provides abstractions to support the applications running on the top of the network. Its language provides constructs for reading and modifying host information, dealing with QoS priority, and redirecting flows through virtual topologies.

The *general purpose* languages were subdivided in three groups considering their programming paradigm: *event-driven*, *imperative* and *functional*.

The *event-driven* languages use the concept of states. The authors of these languages have stated that finite state machines are amenable to model checking, and also some Internet protocols [166], [167] have been implemented using state machines. Another important aspect is that the event-driven model is close to the lower level layer, and the translation to OpenFlow rules would be simpler. However, by using only finite state languages, as proposed by Flog [63] becomes impractical for two reasons: One reason is because these languages provide restricted expressive power. FlowLog [85] has dealt with this issue by combining elements from both restricted and full languages. It provides interfaces and abstractions for interacting with external programs. The second problem occurs as the size of the network increases; the large number of hosts, flows, network events, and policies may cause a state explosion. To deal with this situation, Kinetic [74] has introduced an abstraction called Located Packet Equivalence Class (LPEC). The programmer creates a generic FSM, uses LPEC to specify a division of the flow space (e.g., all flows from the same source MAC address), and maps the LPEC to the generic FSM. Kinetic instantiates multiple copies of the generic FSM, one per LPEC. Kinetic also allows composing FSMs by using the Pyretic's inherited modular compositors.

The *imperative* programming paradigm has been adopted by Pyretic language [53], [55]. This language has proposed network virtualization in terms of virtual switches and slicing. Pyretic also has innovated being the only language to propose an abstract packet model with virtual fields and packet header stacking (*Section III-D5*). Although the imperative paradigm is more prevalent for programming computers, in the context of networks, it does not seem to be the most appropriate choice, mainly due to be far from networks programming model.

The remaining languages have been defined as *general purpose* and *functional* (reactive), with one exception: FML [59]. FML was the first SDN programming language; it has included constructs for defining paths through the network, imposing bandwidth limits, and also allowing policies to be written by using external information sources. Besides FML has been implemented using the logic paradigm, we have fitted it with this group because it provides general purpose abstractions, and it has inspired some other languages in this group. Nettle [60] has introduced dynamic policies and modular composition. Frenetic [61], [106] has incorporated arbitrary wildcards, has proposed a separate language to query network information, and also has included windowed history queries. Procera [62] has extended Nettle by adding an interface to deal with external events. NetCore [69] has been at the forefront by presenting formal semantics and proofs of correctness. It has been used as the core language to express forwarding in Pyretic and Frenetic-OCaml [70]. NetCore has also innovated by proposing proactive flow installation. Frenetic-OCaml has been the first language to provide constructs to ensure consistency. Lastly, NetKAT [118] has incorporated virtual topologies, regular expression for describing legal paths through the network, and the iteration compositor. NetKAT has replaced NetCore in Frenetic-OCaml.

A. Open Issues and Future Directions

SDN programming languages have evolved, incorporating new concepts to increase the level of abstraction for network programmability. Although the languages papers do not present any clear and concise future work, there are still many open issues. This section presents some directions that we believe to be interesting as future research areas and may be implemented as new abstractions in the next generations of network programming languages.

- Programmable forwarding modes: Depending on the network topology, it may be interesting to dynamically change the forwarding mode [168]. For instance, in a hypercube topology, packets can be forwarded by stateless devices by just applying an XOR operation between source and destination addresses. If the switches support programmable forwarding modes, new routing techniques (not based on legacy protocols) can be deployed to take advantage of topology benefits [169]. Network programmers may be interested in deciding what (e.g. coding), where (bits) to insert information into the packets and how to process them (forwarding mode). Some existing works enable switches to modify the forwarding mode [165], [170]. For instance, a packet forwarding mode might be a stateless node that forwards packets using simple MOD operation [171], or a node that processes packets based on {offset, length} tuples[172]. SDN languages should be able to incorporate the evolution of these instructions sets at Data Plane for packets processing offering appropriate higher-level abstractions.

- Libraries: Many computer programming languages have become popular because they offer the standard library as well as community-contributed extensions. These languages can be extended by simply importing modules available in their repositories. The current SDN programming languages provide constructs to deal with specific problems, or offer fundamental constructs, forcing the programmers to write even basic SDN applications. The languages still do not provide an open interface (nor repository), to allow new modules to be developed and incorporated into the language.
- Network orchestration: The presence of congestion means that the load is greater than available resources. Researchers may be interested to use language constructs to scale-up (or out) the resources in a elastic way, for example by setting the routing of some flows to be mapped into multi-disjoint paths, or dynamically enabling a spare device or communication links typically used as backups. Another strategy might be to decrease the load on congested links, by controlling the traffic rate per-flow, per switch port (e.g. traffic shaping at network borders), or on extremely heavy-load conditions to refuse new connections (e.g. controller application for connection admission control).
- Network Function Virtualization (NFV): NFV implements network functions, such as firewalls, network address translation, intrusion detection, gateways, and caching on top of commodity servers, instead of using specialized hardware. NFV provide scalability and elasticity in managing the network. There exist efforts to join SDN with NFV [122], [173], but is still an open issue introducing, in a high-level language, constructs to manage the virtualized functions.
- Network forensics: Provides means to collect information on networks devices for the purpose of discovering evidence of crimes. This is difficult to achieve because the networks deal with volatile and dynamic information. We have found recent efforts providing past-time temporal operators [150], and some isolated works focusing this area [174], [175]. One can perceive that there still exists a lack of research in SDN languages to provide constructs to deal with this area.
- Application requirements: Network needs to be aware of communication requirements imposed by applications running on top of it. Usually, the applications have a known traffic pattern that may be characterized by bandwidth, delay, jitter, and loss. These patterns are difficult to be discovered due to several aspects such as the application behavior, time scale, and so forth. However, these parameters may be used to dynamically modify the network, optimizing it for applications that are running at a given time.
- OpenFlow updates: The languages could take advantage of the new features offered by OpenFlow protocol:
  - OpenFlow v1.1 has incorporated the group tables and, the only language to use this improvement is FatTire [73]. In addition, OpenFlow 1.1 includes other abstractions enabling multicast and multipath that could be offered as constructs in the languages.
  - OpenFlow v1.2 has changed the matching to a type-length-value (TLV) structure, called OpenFlow Extensible Match (OXM) [176]. Prior versions of OpenFlow specification used a static fixed length structure to specify matching, which prevents flexible expression of matches and prevents inclusion of new match fields. This OXM matching mechanism was not availed in any SDN language.
  - OpenFlow v1.3 has included the Tunnel-ID abstraction to support a wide variety of tunnel encapsulations (GRE, L2TP, PPTP). Languages may provide constructs to facilitate tunnel creation, or to enhance tunnel-based virtual topologies.
  - OpenFlow v1.4 has added support to configure optical ports properties, which could be extended to the language level. Also, it has included a bundle mechanism, enabling to apply a group of OpenFlow message as a single operation. The bundle mechanism enables the quasi-atomic flow table update, better synchronizing changes across a series of switches.
  - OpenFlow v1.5 has modified the bundle mechanism by adding scheduled bundles, atomic bundles, and ordered bundles. This mechanism can be used to improve the consistency abstractions provided by the languages.

VI. Conclusion

This paper has presented a comprehensive survey on SDN programming languages. We have focused on language-related aspects, in contrast to existing surveys that cover broader topics of the SDN architecture. First programming language dates from 2009, and we have found fifteen solutions since then, many recently published. This attests the network programmability has been an active research topic. We have presented a genealogy showing the chronological evolutions and relationships among these languages.

We have presented the SDN architecture, exposing the different programming levels, comparing, and commenting their differences. SDN applications were implemented to show some advantages in using a programming language. The drawbacks when the networks are programmed without a language were also considered.

The surveyed languages reveal a broad range of features, but at the same time, there are a considerable amount of intersections among them. We also notice the lack of a consensus on the terminology used in abstractions provided by these languages, which often makes extremely hard to compare and classify them. It has motivated us to carefully investigate the languages main features, in order to propose a taxonomy. We have observed a couple of central contributions, which have been incrementally extended, e. g., Proactive Flow Installation (*Section III-A*), Modular Composition (*Section III-D3*), Windowed History (*Section III-D7*), and so forth. We have identified nineteen different features, divided into ten categories. Our taxonomy has led to a coherent view on the state-of-the-art of network programming languages, as well as the open research issues. There is a wide scope for researchers to offer new abstractions and also to contribute to the advance of NBI standardization.

We hope our taxonomy may collaborate in the NBI standardization. This survey can be useful as support for users and researchers to better understand the existing SDN programming languages, allowing them to choose the language that best fit their requirements.

Acknowledgments

The authors would like to thank CAPES that is providing partial funding for this research, CNPq for the grant 456143/2014-9, and the Brazilian Ministry of Communications that is partly funding it through the project "Digital Inclusion: Technology for Digital Cities".

References

[1] ONF. (2012, April) Software-Defined Networking: The New Norm for Networks. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] ——. (2015, May) SDN Architecture Overview. [Online]. Available: https://www.opennetworking.org/images/stories\\/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf

[3] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 1, pp. 493–512, 2014.

[4] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, T. Turletti *et al.*, "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, 2014.

[5] F. Hu, Q. Hao, and K. Bao, "A survey on software-defined network and openflow: from concept to implementation," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 4, pp. 2181–2206, 2014.

[6] Y. Jarraya, T. Madi, and M. Debbabi, "A survey and a layered taxonomy of software-defined networking," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 4, pp. 1955–1980, 2014.

[7] K. Suzuki, K. Sonoda, N. Tomizawa, Y. Yakuwa, T. Uchida, Y. Higuchi, T. Tonouchi, and H. Shimonishi, "A survey on OpenFlow technologies," *IEICE Transactions on Communications*, vol. 97, no. 2, pp. 375–386, 2014.

[8] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A survey on software-defined networking," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 1, pp. 27–51, 2014.

[9] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[10] H. Farhady, H. Lee, and A. Nakao, "Software-Defined Networking: A survey," *Computer Networks*, vol. 81, pp. 79–95, 2015.

[11] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *Communications Magazine, IEEE*, vol. 51, no. 11, pp. 24–31, 2013.

[12] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer, "Survey on Network Virtualization Hypervisors for Software Defined Networking," *arXiv preprint arXiv:1506.07275*, 2015.

[13] S. Scott-Hayward, G. O'Callaghan, and S. Sezer, "SDN security: A survey," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*. IEEE, 2013, pp. 1–7.

[14] J. François, L. Dolberg, O. Festor, and T. Engel, "Network security through software defined networking: a survey," in *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications*. ACM, 2014, p. 6.

[15] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A Survey of Security in Software Defined Networks," vol. PP, no. 99, 2015.

[16] J. Proença, T. Cruz, E. Monteiro, and P. Simões, "How to use Software–Defined Networking to Improve Security–a Survey," in *Proceedings of the 14th European Conference on Cyber Warfare and Security 2015: ECCWS 2015*. Academic Conferences Limited, 2015, p. 220.

[17] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, "A Survey of Securing Networks Using Software Defined Networking," 2015.

[18] M. Yang, Y. Li, D. Jin, L. Zeng, X. Wu, and A. V. Vasilakos, "Software-defined and virtualized future mobile and wireless networks: a survey," *Mobile Networks and Applications*, vol. 20, no. 1, pp. 4–18, 2014.

[19] N. A. Jagadeesan and B. Krishnamachari, "Software-defined networking paradigms in wireless networks: a survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 27, 2014.

[20] C. Liang and F. R. Yu, "Wireless network virtualization: A survey, some research issues and challenges," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 1, pp. 358–380, 2015.

[21] P. Bhaumik, S. Zhang, P. Chowdhury, S.-S. Lee, J. H. Lee, and B. Mukherjee, "Software-defined optical networks (SDONs): a survey," *Photonic Network Communications*, vol. 28, no. 1, pp. 4–18, 2014.

[22] X. Chen and Y. Zhang, "Intelligence on Optical Transport SDN," *International Journal of Computer and Communication Engineering*, vol. 4, no. 1, p. 5, 2015.

[23] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.

[24] W. Gu, X. Zhang, B. Gong, and L. Wang, "A Survey of Multicast in Software-Defined Networking," in *Proceedings of the fifth International Conference on Information Engineering for Mechanics and Materials (ICIMM)*, 2015, pp. 1096–1100.

[25] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based comparison and selection of Software Defined Networking (SDN) controllers," in *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*. IEEE, 2014, pp. 1–7.

[26] S. K. Rao, "SDN and Its Use-Cases-NV and NFV," 2014.

[27] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," DTIC Document, Tech. Rep., 1990.

[28] N. Feamster and H. Balakrishnan, "Detecting bgp configuration faults with static analysis," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 43–56.

[29] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[30] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 13–24.

[31] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[32] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (netconf)," Internet Requests for Comments, RFC Editor, RFC 6241, June 2011.

[33] B. Pfaff and B. Davie, "The open vswitch database management protocol," Internet Requests for Comments, RFC Editor, RFC 7047, December 2013.

[34] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, "The locator/id separation protocol (lisp)," Internet Requests for Comments, RFC Editor, RFC 6830, January 2013.

[35] J. Yu, X. Wang, J. Song, Y. Zheng, and H. Song, "Forwarding programming in protocol-oblivious instruction set," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. IEEE, 2014, pp. 577–582.

[36] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, "Opflex control protocol," Working Draft, IETF Secretariat, Internet-Draft draft-smith-opflex-00, April 2014, http://www.ietf.org/internet-drafts/draft-smith-opflex-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-smith-opflex-00.txt

[37] A. Autenrieth, J.-P. Elbers, P. Kaczmarek, and P. Kostecki, "Cloud orchestration with sdn/openflow in carrier transport networks," in *Transparent Optical Networks (ICTON), 2013 15th International Conference on*. IEEE, 2013, pp. 1–4.

[38] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang, "Meridian: an sdn platform for cloud network services," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 120–127, 2013.

[39] V. Kotronis, X. Dimitropoulos, R. Klöti, B. Ager, P. Georgopoulos, and S. Schmid, "Control exchange points: Providing qos-enabled end-to-end services via sdn-based inter-domain routing orchestration," *LINX*, vol. 2429, no. 1093, p. 2443, 2014.

[40] C. Yin, T.-C. Kuo, T.-Y. Li, M.-C. Chang, and B.-H. Liao, "Mediating between openflow and legacy transport networks for bandwidth on-demand services," in *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*. IEEE, 2014, pp. 1–4.

[41] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[42] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. ACM, 2010, p. 8.

[43] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *Internet Computing, IEEE*, vol. 17, no. 2, pp. 20–27, 2013.

[44] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram *et al.*, "Network virtualization in multi-tenant datacenters," in *USENIX NSDI*, 2014.

[45] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[46] M. McCauley. (2015, December) About POX. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[47] D. Erickson, "The beacon openflow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 13–18.

[48] Ryu. (2015, April) Ryu SDN Framework. [Online]. Available: http://osrg.github.io/ryu/

[49] B. S. Networks. (2015, April) Project Floodlight. [Online]. Available: http://www.projectfloodlight.org/

[50] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller architecture," in *2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 1–6.

[51] H. Julkunen and C. E. Chow, "Enhance network security with dynamic packet filter," in *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*. IEEE, 1998, pp. 268–275.

[52] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[53] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN programming with Pyretic," *Technical Reprot of USENIX*, 2013.

[54] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending Networking into the Virtualization Layer," in *Eight ACM Workshop on Hot Topics in Networks (HotNets-VIII), HOTNETS '09, New York City, NY, USA, October 22-23, 2009*, 2009.

[55] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing Software Defined Networks," in *NSDI*, 2013, pp. 1–13.

[56] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography." *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[57] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the network with Merlin," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 24.

[58] D. Batory, *Feature models, grammars, and propositional formulas*. Springer, 2005.

[59] 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, pp. 1–10.

[60] A. Voellmy, A. Agarwal, and P. Hudak, "Nettle: Functional reactive programming for openflow networks," DTIC Document, Tech. Rep., 2010.

[61] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker, "Frenetic: a high-level language for OpenFlow networks," in *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. ACM, 2010, p. 6.

[62] A. Voellmy, H. Kim, and N. Feamster, "Procera: a language for high-level reactive network control," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 43–48.

[63] N. P. Katta, J. Rexford, and D. Walker, "Logic programming for software-defined networks," in *Workshop on Cross-Model Design and Validation (XLDI)*, vol. 412, 2012.

[64] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 213–226.

[65] B. L. A. Batista and M. P. Fernandez, "PonderFlow: A Policy Specification Language for Openflow Networks," *ICN 2014*, p. 215, 2014.

[66] C. Trois, M. Martinello, L. Bona, and M. Del Fabro, "From Software Defined Network To Network Defined for Software," in *Proceedings of the 2015 ACM Symposium on Applied Computing*. ACM, 2015.

[67] B. Salisbury, "OpenFlow: Proactive vs Reactive Flows," 2013.

[68] M. P. Fernandez, "Comparing openflow controller paradigms scalability: Reactive and proactive," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE, 2013, pp. 1009–1016.

[69] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 217–230, 2012.

[70] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger *et al.*, "Languages for software-defined networks," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 128–134, 2013.

[71] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 113–126, 2014.

[72] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "A balance of power: Expressive, analyzable controller programming," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 79–84.

[73] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 109–114.

[74] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable Dynamic Network Control," in *USENIX NSDI*, 2015.

[75] G. N. Stone, B. Lundy, and G. G. Xie, "Network policy languages: a survey and a new approach," *Network, IEEE*, vol. 15, no. 1, pp. 10–21, 2001.

[76] P. Van Roy *et al.*, "Programming paradigms for dummies: What every programmer should know," *New computational paradigms for computer music*, vol. 104, 2009.

[77] A. Mottola, "Design and implementation of a declarative programming language in a reactive environment," Ph.D. dissertation, Università degli Studi di Roma, 2005.

[78] J. Qadir and O. Hasan, "Applying Formal Methods to Networking: Theory, Techniques, and Applications," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 1, pp. 256–291, 2015.

[79] K. Sharan, *Harnessing Java 7: A Comprehensive Approach to Learning Java*. CreateSpace Independent Publishing Platform, 2012, vol. 1.

[80] J. W. Lloyd, *Foundations of logic programming*. Springer Science & Business Media, 2012.

[81] W. Clocksin and C. S. Mellish, *Programming in PROLOG*. Springer Science & Business Media, 2003.

[82] V. Lifschitz, "What Is Answer Set Programming?" in *AAAI*, vol. 8, 2008, pp. 1594–1597.

[83] N. Li and J. C. Mitchell, "Datalog with constraints: A foundation for trust management languages," in *Practical Aspects of Declarative Languages*. Springer, 2003, pp. 58–73.

[84] G. Ucoluk and S. Kalkan, *Introduction to programming concepts with case studies in Python*. Springer Science & Business Media, 2012.

[85] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," *NSDI, Apr*, 2014.

[86] S. Ferg, *Event-driven programming: Introduction, tutorial, history*. Autoedicion, 2006.

[87] J. Armstrong, *Programming Erlang*. Pragmatic Bookshelf, 2013.

[88] D. Rémy, "Using, understanding, and unraveling the OCaml language from practice to theory and vice versa," in *Applied Semantics*. Springer, 2002, pp. 413–536.

[89] S. Halloway, *Programming Clojure*. Pragmatic Bookshelf, 2009.

[90] S. Thompson, *Haskell: the craft of functional programming*. Addison-Wesley, 1999, vol. 2.

[91] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter, "A survey on reactive programming," in *ACM Computing Surveys*, 2012.

[92] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *ACM SIGPLAN Notices*, vol. 35, no. 5. ACM, 2000, pp. 242–252.

[93] E. Amsden, "A survey of functional reactive programming," *Unpublished*, 2011.

[94] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 411–422.

[95] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: a programming language for Ajax applications," in *ACM SIGPLAN Notices*, vol. 44, no. 10. ACM, 2009, pp. 1–20.

[96] A. Courtney, H. Nilsson, and J. Peterson, "The yampa arcade," in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. ACM, 2003, pp. 7–18.

[97] M. Shaw, "Abstraction techniques in modern programming languages," *IEEE software*, no. 4, pp. 10–26, 1984.

[98] C. Szyperski, J. Bosch, and W. Weck, "Component-oriented programming," in *Object-oriented technology ecoop99 workshop reader*. Springer, 1999, pp. 184–192.

[99] M. Casado, N. Foster, and A. Guha, "Abstractions for software-defined networks," *Communications of the ACM*, vol. 57, no. 10, pp. 86–95, 2014.

[100] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM, 2011, p. 7.

[101] S. Ghorbani and M. Caesar, "Walk the line: consistent network updates with bandwidth guarantees," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 67–72.

[102] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 49–54.

[103] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 20.

[104] A. Noyes, T. Warszawski, P. Černỳ, and N. Foster, "Toward synthesis of network updates," *arXiv preprint arXiv:1403.7840*, 2014.

[105] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 323–334.

[106] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, no. 9. ACM, 2011, pp. 279–291.

[107] Y. Feng, R. Guo, D. Wang, and B. Zhang, "Research on the Active DDoS Filtering Algorithm Based on IP Flow," in *Natural Computation, 2009. ICNC'09. Fifth International Conference on*, vol. 4. IEEE, 2009, pp. 628–632.

[108] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*. IEEE, 2010, pp. 408–415.

[109] K. Giotis, G. Androulidakis, and V. Maglaris, "Leveraging SDN for efficient anomaly detection and mitigation on legacy networks," in *Proceedings of the third European Workshop on Software Defined Networks (EWSDN)*. IEEE, 2014, pp. 85–90.

[110] K. Wang, Y. Qi, B. Yang, Y. Xue, and J. Li, "LiveSec: Towards effective security management in large-scale production networks," in *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*. IEEE, 2012, pp. 451–460.

[111] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks," in *NDSS*, 2013.

[112] A. Zaalouk, R. Khondoker, R. Marx, and K. Bayarou, "OrchSec: An orchestrator-based architecture for enhancing network-security using Network Monitoring and SDN Control functions," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–9.

[113] F. Klaedtke, G. O. Karame, R. Bifulco, and H. Cui, "Access control for SDN controllers," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 219–220.

[114] J. Matias, J. Garay, A. Mendiola, N. Toledo, and E. Jacob, "FlowNAC: Flow-based network access control," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 79–84.

[115] K. Liu and K. Xu, "OAuth based authentication and authorization in open telco API," in *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, vol. 1. IEEE, 2012, pp. 176–179.

[116] R. Soulé, S. Basu, E. G. Sirer, and N. Foster, "Scalable Network Management with Merlin," 2013.

[117] J. C. Mitchell, *Foundations for programming languages*. MIT press Cambridge, 1996, vol. 1.

[118] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha, "A Fast Compiler for NetKAT," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.

[119] ONF, "OpenFlow Switch Specification Version 1.5.0," December 2014.

[120] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *Communications Magazine, IEEE*, vol. 35, no. 1, pp. 80–86, 1997.

[121] N. M. K. Chowdhury and R. Boutaba, "A survey of network virtualization," *Computer Networks*, vol. 54, no. 5, pp. 862–876, 2010.

[122] J. Batalle, J. Ferrer Riera, E. Escalona, and J. A. Garcia-Espin, "On the implementation of NFV over an OpenFlow infrastructure: Routing Function Virtualization," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–6.

[123] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN," *Queue*, vol. 11, no. 12, p. 20, 2013.

[124] M. R. Nascimento, C. E. Rothenberg, M. R. Salvador, C. N. Corrêa, S. C. de Lucena, and M. F. Magalhães, "Virtual routers as a service: the routeflow approach leveraging software-defined networks," in *Proceedings of the 6th International Conference on Future Internet Technologies*. ACM, 2011, pp. 34–37.

[125] E. L. Fernandes and C. E. Rothenberg, "Openflow 1.3 software switch," *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC, pgs*, pp. 1021–1028, 2014.

[126] P. Floodlight. (2015, August) Indigo Virtual Switch. [Online]. Available: http://www.projectfloodlight.org/indigo-virtual-switch/

[127] Z. Bozakov and P. Papadimitriou, "Autoslice: automated and scalable slicing for software-defined networks," in *Proceedings of the 2012 ACM conference on CoNEXT student workshop*. ACM, 2012, pp. 3–4.

[128] H. Yamanaka, E. Kawai, S. Ishii, and S. Shimojo, "AutoVFlow: Autonomous virtualization for wide-area OpenFlow networks," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 67–72.

[129] S. F. Python. (2015, August) Data Structures – Python 2.7.10 documentation. [Online]. Available: https://docs.python.org/2/tutorial/datastructures.html

[130] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, "Requirements for Traffic Engineering Over MPLS," Internet Requests for Comments, RFC Editor, RFC 2702, September 1999.

[131] N. Wang, K. Ho, G. Pavlou, and M. Howarth, "An overview of routing optimization for internet traffic engineering," *Communications Surveys & Tutorials, IEEE*, vol. 10, no. 1, pp. 36–56, 2008.

[132] G. R. Ash, "Traffic Engineering & QoS Methods for IP, ATM, & TDM-Based Multi-service Networks," in *Internet-Draft, Work in Progress*, 2001.

[133] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao, "Overview and Principles of Internet Traffic Engineering," Internet Requests for Comments, RFC Editor, RFC 3272, May 2002.

[134] H. Kim, J. R. Santos, Y. Turner, M. Schlansker, J. Tourrilhes, and N. Feamster, "Coronet: Fault tolerance for software defined networks," in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*. IEEE, 2012, pp. 1–2.

[135] F. Botelho, A. Bessani, F. Ramos, and P. Ferreira, "On the design of practical fault-tolerant SDN controllers," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 73–78.

[136] B. Chandrasekaran and T. Benson, "Tolerating SDN application failures with LegoSDN," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 22.

[137] H. Long, Y. Shen, M. Guo, and F. Tang, "LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks," in *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE, 2013, pp. 290–297.

[138] H. E. Egilmez, S. Civanlar *et al.*, "An optimization framework for QoS-enabled adaptive video streaming over OpenFlow networks," *Multimedia, IEEE Transactions on*, vol. 15, no. 3, pp. 710–715, 2013.

[139] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Adding regular expressions to graph reachability and pattern queries," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 39–50.

[140] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: traffic matrix estimator for OpenFlow networks," in *Passive and active measurement*. Springer, 2010, pp. 201–210.

[141] Y. Yu, C. Qian, and X. Li, "Distributed and collaborative traffic monitoring in software defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 85–90.

[142] N. L. Van Adrichem, C. Doerr, F. Kuipers *et al.*, "Opennetmon: Network monitoring in openflow software-defined networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–8.

[143] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "Payless: A low cost network monitoring framework for software defined networks," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–9.

[144] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker, "Expressing and enforcing flow-based network security policies," *University of Chicago, Tech. Rep*, 2008.

[145] H. Kim and N. Feamster, "Improving network management with software defined networking," *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 114–119, 2013.

[146] C. M. Rogers, "ANQLAn Active Networks Query Language," in *Active Networks*. Springer, 2002, pp. 99–110.

[147] D. Kozen, "NetKATA Formal System for the Verification of Networks," in *Programming Languages and Systems*. Springer, 2014, pp. 1–18.

[148] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson, "A coalgebraic decision procedure for NetKAT," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2015, pp. 343–355.

[149] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 3, pp. 427–443, 1997.

[150] R. Beckett, M. Greenberg, and D. Walker, "Temporal NetKAT," 2015, pLVNET.

[151] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic NetKAT," Cornell University, Tech. Rep., 2015.

[152] J. McClurg, H. Hojjat, N. Foster, and P. Cerny, "Specification and Compilation of Event-driven SDN Programs," *arXiv preprint arXiv:1507.07049*, 2015.

[153] R. Soule, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: Programming the Big Switch," *The Open Networking Summit (ONS14)*, 2014.

[154] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Policies for Distributed Systems and Networks*. Springer, 2001, pp. 18–38.

[155] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[156] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: A slice abstraction for software-defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 79–84.

[157] A. D. Ferguson, A. Guha, J. Place, R. Fonseca, and S. Krishnamurthi, "Participatory networking," *Proc. Hot-ICE*, vol. 12, 2012.

[158] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An API for application control of SDNs," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 327–338.

[159] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Hierarchical policies for software defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 37–42.

[160] H. Shimonishi, Y. Takamiya, Y. Chiba, K. Sugyo, Y. Hatano, K. Sonoda, K. Suzuki, D. Kotani, and I. Akiyoshi, "Programmable network using OpenFlow for network researches and experiments," in *Proc. 6th International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2012)*, 2012, pp. 164–171.

[161] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 87–98.

[162] A. Voellmy and J. Wang, "Scalable software defined network controllers," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 289–290.

[163] S. Narayana, J. Rexford, and D. Walker, "Compiling path queries in software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 181–186.

[164] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent NetCore: From policies to pipelines," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. ACM, 2014, pp. 11–24.

[165] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[166] J. Lang, "Link Management Protocol (LMP)," Internet Requests for Comments, RFC Editor, RFC 4204, October 2005.

[167] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," Internet Requests for Comments, RFC Editor, RFC 4271, January 2006.

[168] G. L. Vassoler, M. H. Paiva, M. Ribeiro, and M. E. Segatto, "Twin Datacenter Interconnection Topology," *Micro, IEEE*, vol. 34, no. 5, pp. 8–17, 2014.

[169] R. Ramos, M. Martinello, and C. Esteve Rothenberg, "SlickFlow: Resilient source routing in Data Center Networks unlocked by OpenFlow," in *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, Oct 2013, pp. 606–613.

[170] R. Vencioneck, G. Vassoler, M. Martinello, M. Ribeiro, and C. Marcondes, "FlexForward: Enabling an SDN manageable forwarding engine in Open vSwitch," in *Network and Service Management (CNSM), 2014 10th International Conference on*, Nov 2014, pp. 296–299.

[171] M. Martinello, M. Ribeiro, R. de Oliveira, and R. de Angelis Vitoi, "Keyflow: a prototype for evolving SDN toward core network fabrics," *Network, IEEE*, vol. 28, no. 2, pp. 12–19, March 2014.

[172] H. Song, "Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 127–132. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491190

[173] H. Masutani, Y. Nakajima, T. Kinoshita, T. Hibi, H. Takahashi, K. Obana, K. Shimano, and M. Fukui, "Requirements and design of flexible NFV network infrastructure node leveraging SDN/OpenFlow," in *Optical Network Design and Modeling, 2014 International Conference on*. IEEE, 2014, pp. 258–263.

[174] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, "Let SDN be your eyes: Secure forensics in data center networks," in *Proceedings of the NDSS Workshop on Security of Emerging Network Technologies (SENT14)*, 2014.

[175] I. E. Achumba, K. C. Okafor, G. N. Ezeh, and U. H. Diala, "OpenFlow Virtual Appliance: An Efficient Security Interface For Cloud Forensic Spyware Robot," *International Journal of Digital Crime and Forensics (IJDCF)*, vol. 7, no. 2, pp. 31–52, 2015.

[176] ONF, "OpenFlow Switch Specification Version 1.2," 2011.