I.J. Wireless and Microwave Technologies, **2024, 1, 52-72**
Published Online on February 8, 2024 by MECS Press (http://www.mecs-press.org/)
DOI: 10.5815/ijwmt.2024.01.05

Programming SDNs: A Compass for SDN Programmer

Suhail Ahmad*
Department of Computer Science & Engineering, University of Kashmir, India
E-mail: suhail.sam008@gmail.com
ORCID iD: https://orcid.org/0000-0002-4579-9237
*Corresponding Author

Ajaz Hussain Mir
Department of Electronics & Communication Engineering, National Institute of Technology Srinagar, India
E-mail: ahmir@rediffmail.com
ORCID iD: https://orcid.org/0000-0001-9777-0850

Received: 21 June, 2023; Revised: 04 August, 2023; Accepted: 12 September, 2023; Published: 08 February, 2024

Abstract: The modern communication networks have evolved from simple-static systems to highly flexible and adaptive systems facilitating dynamic programmability and reconfiguration. This network evolution has influenced the lowest level of packet processing in data plane to highest level of network control and management functions. It has also influenced the overall network design and architecture which is clearly evident from the emergence of SDN and NFV. With the wide-spread acceptance of SDN, a novel networking paradigm, the network programmability has re-appeared as a top research area in networking and numerous programming languages have been proposed. In this paper, we present a systematic review of various state-of-the-art SDN programming languages used to program different network planes. We follow a top-down approach, starting with the high-level or top-tier programming languages followed by the data plane or bottom-tier programming languages. We have provided an in-depth analysis of various top-tier and bottom-tier programming languages and compared them in terms of most prominent features and supported abstractions. In addition to it, we have elaborated various programming models used in different bottom-tier programming languages which provide necessary abstractions for mapping diverse functionalities of data plane algorithms splendidly onto the specialized hardware like ASICs. Lastly, we have highlighted the research challenges in SDN programming languages like cross platform programming, necessary language libraries, support for network verification, NFV, stateful and inline packet processing, which need to be incorporated into existing programming languages to support diverse functions required in next generation networks.

Index Terms: Programmability in SDN, Domain Specific Languages, Programmable Networks, Network Programming Languages, OpenFlow, P4

1. Introduction

The two initiatives of mid-90s with primary objective to make networks programmable were programmable networks [1] and active networking [2]. The former promoted separation of control and data plane whereas the later investigated insertion of code into data packets termed as capsules to achieve intelligent networking. Afterwards, with the advent of the large-scale data centers, 5G mobile standards, ubiquitous IoE, and massive big data and machine learning applications, the network operators had to embrace new novel techniques to architect next generation networks. As a result, the edge computing, network virtualization, Software Defined Networking, and service chaining find their way into modern communication networks.

In communication networks, the dynamic traffic flows, link/device status change or congestion in the network modifies the overall network state. The network control plane monitors and verifies the network state and modifies the forwarding behaviour as per the state change. However, conventional devices supported limited approaches or protocols in the control plane and require numerous middle-boxes to monitor and regulate the traffic flows. In next generation networks, such limited approaches fail to provide desired Quality of Service (QoS) and necessitates use of advanced approaches for efficient network monitoring, verfication and debugging. Moreover, the impending application requirements force the device vendors to support ever-evolving heterogenous protocols and at the same time provide

basic functions like L4/L3/L2 packet processing, tunneling and more advanced features like QoS, congestion control, intrusion detection, etc., in the network devices. Providing all these features at a desired level of flexibility, efficiency and performance requires extensive and careful engineering effort which often results in complex and costly designs, and manufacturing of a dedicated hardware [3]. Such a dedicated hardware results into two major problems: (i) *inhibiting innovation:* adding novel features involves rapid device upgrade cycles and cost-effective development process, hence forces vendors to roll-out new features only when it becomes most widely desired feature; (ii) *inefficient resource utilitzation:* supporting every single protocol or feature may lead to wastage of resources like CPU cycles, memory space and silicon chips as most of these features may never be used.

To address these challenges, the traditional SDN [4] initially emerged with the idea of control plane programmability and limited flexibility in data plane by allowing network operators to define flow rules using well defined APIs like OpenFlow [5]. The complex and compute-intensive algorithms are defined in software and were commonly executed on a SDN Controller, having bird-eye view of the entire network. Therefore, the intricate control plane algorithms which were designed for distributed legacy devices are replaced by centralized, simpler algorithms. Such a centralized control environment is beneficial for high demand use cases like 5G networks or multi-tenant data centers to achieve flexibility, scalability and efficiency in network control. However, this traditional approach of SDN offers programmability in top-tier (control and management plane) and data plane devices are still irreplaceable and under the control of device vendors. This constraint was resolved with programmable data planes wherein the users can define their own algorithms and can build customized network equipment without compromising the overall network performance, speed, or scalability. Numerous top-tier and bottom-tier programming languages and frameworks have been proposed and in this manuscript our goal is to analyze these programming languages and frameworks with stress on potential future trends that may promote further research in SDN programming.

1.1 Related Work and Contributions

With the introduction of OpenFlow [5] and advances in SDN paradigm, the network programming has re-emerged as a buzzword in computer networking. Till now, numerous surveys have addressed different facets of SDN, including design challenges and research issues [6-8], SDN controllers [9, 10], fault management [11], traffic engineering [12], security [13, 14], and applications of this radical paradigm in diverse domains [15-18]. However, SDN programming frameworks and languages which enable network programmers to dynamically control and manage the network control logic with extraordinary efficiency and define the packet processing pipeline have been least surveyed by the research community.

The authors in [19] have laid the foundation of basic programming concepts and programming contructs required in top-tier programming or high-level SDN programming languages. Further, they have classified the SDN programming languages on the basis of Feature Oriented Domain Analysis (FODA). However, they have only focused on top-tier programming languages and over the last few years these programming languages have been extensively extended to address various limitations which necessitates a fresh analysis of all such languages. On the other hand, authors in [20, 21] have addressed the data plane programmability. In [20], the authors have presented anatomy of a programmable SDN switch and have surveyed the architectures and abstractions proposed for a programmable switch. The authors in [20] have discussed data plane programming languages as a subsection with primary focus on data plane flexibility and programmability. They have highlighted the limitations of OpenFlow and ForCES architectures and based on such limitations, they have provided a summary of approaches used to address these problems. In [21], the authors have only focused on P4 [22] and have provided a comprehensive overview and introduction of P4 with applications of P4 in diverse domains.

In the last one decade, there has been tremendous development with respect to network programming which initially began with top-tier programming languages followed by bottom-tier programming frameworks in the last few years. We have explained network programmability and how SDN simplifies network programming with separation of network control logic from packet processing functionality. In this manuscript, we have attempted to cover all the advancements and innovations related to network programming. We have provided a systematic and elaborate analysis of various top-tier and bottom-tier programming languages. The programming languages discussed in this manuscript are shown in Fig. 1 along with the year of inception. The contributions made are as follows:

- *Network programmability:* We have explained the network programmability and elaborated how SDN simplifies network programming in next generation networks.
- *Top-tier programming language features:* We have presented the various SDN programming language features and have discussed each feature in detail.
  Top-tier programming languages: We have analyzed more than twenty top-tier programming languages in terms of following features: network policy definition, flow installation, programming paradigm and supported abstractions. We have highlighted the necessary extensions made to these programming languages to address various programming issues.

- *Bottom-tier programming models:* To map packet processing algorithms onto specialized hardware, different programming languages use specific programming models to express packet processing logic in an abstract way. We have presented three such models and mapped various bottom-tier programming languages to these models.
- *Bottom-tier programming languages:* We have analyzed more than ten bottom-tier programming languages with emphasis on their objectives, supported models, mechanisms employed and problems addressed.
- *Future research perspectives:* We have highlighted various future research challenges in terms of necessary abstractions, features and programming constructs to be incorporated into existing programming languages to support diverse functionalities in next generation networks.

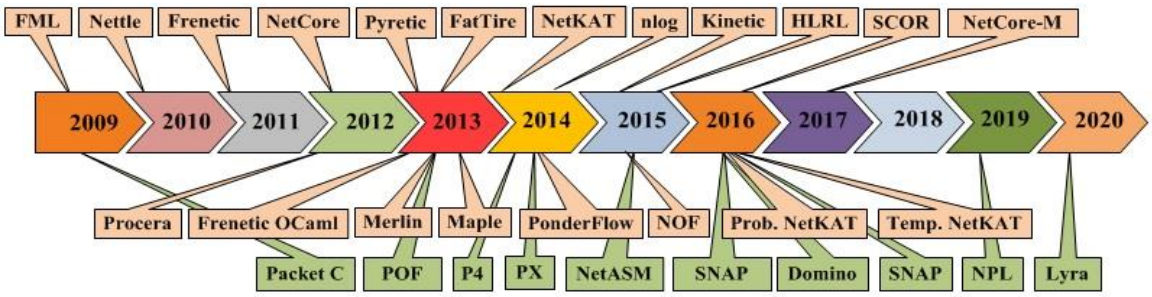

Fig. 1. Top-tier and bottom-tier programming languages analyzed in this manuscript

1.2 Paper Outline

The rest of this manuscript is organised as follows: section 2 introduces network programmability along with programmability in SDN. In section 3, we have first presented top-tier programming language features and then various such languages are analyzed in detail. Section 4 begins with the introduction of data plane programming models followed by in-depth survey of various bottom-tier programming languages. Section 5 highlights the future research perspectives in network programming and finally the paper is concluded in section 6.

2. Network Programmability

The term programmability in general refers to the ability of hardware or software to accomplish processing of an externally-defined procedure. Therefore, programmable entities are distinct from re-configurable ones, as the former allows to carry-out externally defined tasks whereas the latter allows to change certain parameters of a well defined internal algorithm. Likewise, network programmability refers to the ability of execution of externally defined procedure on packet processing nodes such as routers, switches, or load balancers. In this section, we have illustrated how SDN enables flexible, efficient and programmable network control. In addition to it, we have elaborated how the latest trends change the conventional OpenFlow based packet processing in data plane to more flexible, programmable and dynamically defined packet processing pipeline.

2.1 Programmability in SDN

SDN is a scalable, programmable, agile and centrally controlled network architecture which promotes flexible and automated network control. The three layer architecture of SDN as proposed by ONF [23] is shown in Fig. 2. The bottom layer or forwarding plane includes network devices managed and controlled by a remote disjoint control plane. The central layer or control plane consists of logically centralized controllers which monitor and manage the entire network. The controllers provide bird-eye view of entire network at a single central point, hence simplifies network programming by hiding intricate control logic. The control and data plane communicate over a standard open interface termed as southbound interface. The logically centralized controllers use east/westbound SDN interfaces to share network state information with each other or legacy distributed control [24]. The top layer or management plane comprises of various applications which are designed to implement specific functions like load balancing, routing, firewalls, etc. These management applications communicate with the control plane over the northbound interface as shown in Fig. 2.

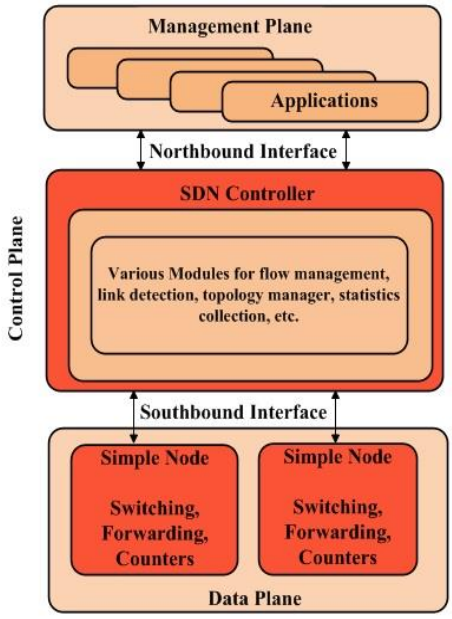

Fig. 2. SDN Architecture

Unlike conventional devices wherein device functionality is defined by the device vendor, the algorithms for different SDN planes can be defined by the users or network operators. This novel concept of requirement based algorithm implementation in both control and data plane is basically the true network programmability. Therefore, SDN fosters innovation by allowing both the end users and the equipment vendors to build networks as per their needs. SDN traditionally evolved with the control plane programmability (referred to as top-tier programmability in this manuscript), however, immediately afterwards the researchers realized the need of data plane programmability (referred to as bottom-tier programmability in this manuscript). The SDN controller may implement high-level complex control programs, but without data plane performance optimization, the per-packet processing may result in low performance. In the following two sub-sections, we have elucidated both top and bottom-tier SDN programmability along with their benefits.

2.1.1 Top-tier Programmability in SDN

The SDN control plane includes one or more SDN controllers which implement basic control logic necessary to manage and operate the network. Most of the SDN controllers [25-28] allow network operators to define their own customized control programs or to bypass in-built procedures, making the network control highly programmable and extensible. Such user defined algorithms could directly communicate with the data plane devices using standard APIs like OpenFlow [5]. Traditionally, SDN involved vendor specific data plane devices which were remotely controlled by the SDN controllers by installing flow rules into flow tables using standard APIs as shown in Fig. 3(a). This programmable, centralized, network-wide control has proved very productive in use cases like multi-tenant data centers.

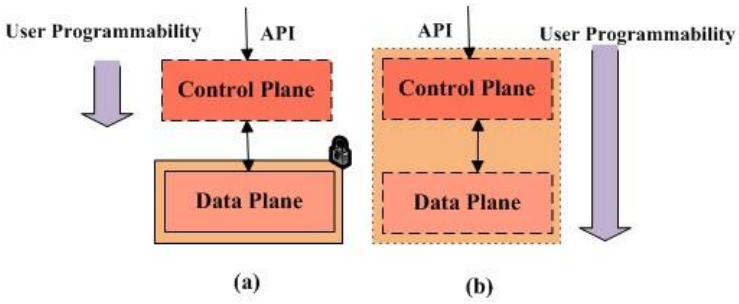

Fig. 3. Programmability in: a) Control Plane; and b) Control and Data plane

The major challenge here is to translate the high-level policy directives into flow rules. Handling complex policy directives with low-level languages may result in conflicting flow rules and in the worst-case inconsistent network state. Numerous programming languages have been proposed to address this issue and these top-tier programming languages provide better, flexible and less error prone alternative to program SDNs.

Initially, the standardization efforts for southbound API resulted in the development of most widely used SDN protocol OpenFlow. At that time it was realized that the flexible and effective API will be enough to manage the data

plane devices or regulate their behaviour. However, very soon it became evident that the OpenFlow based data plane devices have resulted in a specific, pre-defined packet processing structure which could not be altered. This realization led to the beginning of bottom-tier programmability or data plane programming which is discussed in the next sub-section.

2.1.2 Bottom-tier Programmability in SDN

In traditional SDN, the data plane devices perform limited number of operations on the received packets. The packet processing begins with the packet parsing, in which the data plane devices determine certain packet bytes to obtain the control information. This control information defines the series of packet processing operations to be performed on the packet. Such operations can be some computation (like evaluation of header checksum), updating certain bytes of packet (e.g., updating the checksum into the suitable packet fields), or saving state or monitoring information (e.g., writing certain counters). Finally, the packet may be forwarded on specific port(s) or may be dropped.

The data plane or bottom-tier programmability means that the end user can define the forwarding functionality or packet processing logic as per the requirement using reconfigurable ASICs. Therefore, a programmable switch exposes the low-level interface by which the progression and semantics of packet processing operations are defined, modified or reconfigured systematically and rapidly. Unlike the traditional fixed function network devices, the programmable switch allows to modify packet parsing logic, to provide arbitrary header field matching or introduction of new forwarding constructs or novel protocols for experimentation or innovation. Such a programmable switch can be realized with the help of re-configurable hardware and programming languages [29]. Initially, hardware based architectures were mainly used to implement packet processing nodes, however, with the advances in processing capability of conventional computer systems, software based switches are also used.

In a fully programmable network architecture, the user can define new control plane algorithms and can accordingly design the data plane processing logic to build a custom network as shown in Fig. 3 (b). Thus, programmability in control and data plane offers flexible, network customization which is having high implications in commercial appliances, e.g., data centers or 5G networks or can be utilized for rapid prototyping in production and academic research.

3. Top-Tier Programming

In SDN control plane, the programmers have to ensure seamless network operations and separation of various tasks concurrently like forwarding, access control, mirroring, etc. Programming such intricate tasks with low-level languages may result in conflicting flow rules and in the worst-case inconsistent network state. To address this issue, numerous top-tier programming languages have been proposed which provide diverse set of functionalities. These programming languages raise level of abstraction by specifying the network policies in a flexible application development environment. The majority of these languages have primarily focused on translation of high-level network policy requirements into flow table rules. In this section, we have first introduced top-tier programming language features and then various programming languages and frameworks are analyzed in detail.

3.1 Programming Language Features

The features of top-tier SDN programming language are depicted in Fig. 4. An overview of all these features is provided in this sub-section and in the subsequent section various programming languages are compared with respect to these features.

i. Network Policy Definition: A network policy is formally defined as a collection of policy rules [30]. Each rule is a set of conditions and a related set of actions. The conditions express which policy rule is applicable and accordingly corresponding set of actions are initiated. The network policies can be triggered either statically or dynamically. Static policies apply actions in a predetermined way, according to predefined set of rules like firewall filters. On the other hand, dynamic policies change as per the network state. Majority of languages have dynamic capability except few which work statically.

ii. Flow Installation: It refers to the way flow rules are installed on the forwarding devices. The controller installs and manages flows on the forwarding devices either proactively or reactively. In proactive approach, flow rules are communicated in advance by the controller to the switches as per the topology information whereas in reactive approach, whenever a new flow packet arrives at a switch, it is forwarded to the controller. Another possibility is a hybrid approach which combines benefits of both these approaches. In the hybrid approach, some specific rules are installed proactively whereas rest of the flow rules are installed reactively. Proactive approach being more agile and reduces delay in processing packets at the cost of inefficient use of switch memory whereas reactive approach being more flexible and applicable in dynamic environments, but may overwhelm a controller if the flow setup requests are very rapid.

iii. Programming Paradigm: It refers to the way of building elements and overall structure of a computer program. There are two general approaches: imperative and declarative. The former can be viewed as a conventional programming model in which programmer specifies all the steps to solve a particular problem. The later just specifies

what a program must do, without specifying how to do it [31, 32]. For instance, in SQL we specify the query without bothering about how it will be executed on the database engine. The sub-paradigms of declarative are logic programming, event-driven, functional and functional reactive.

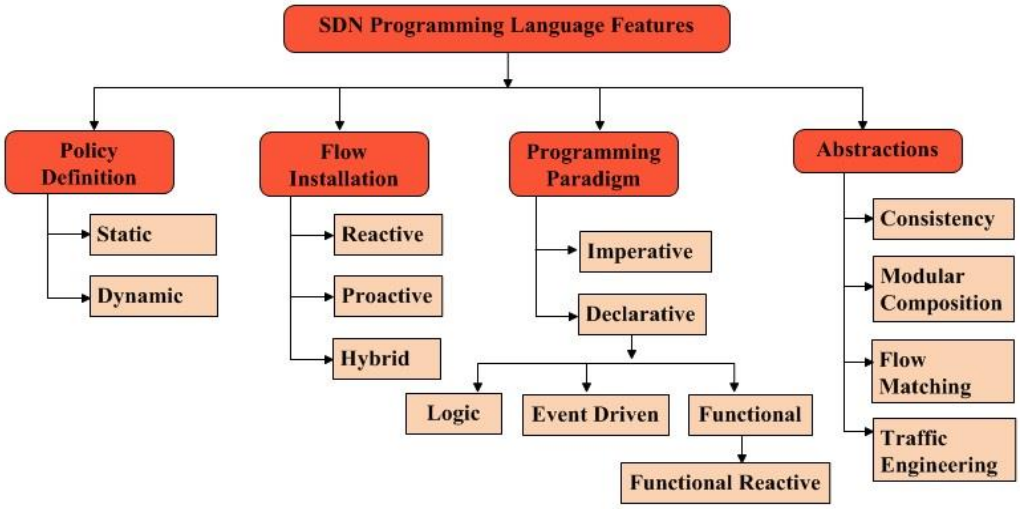

Fig. 4. Top-tier SDN Programming Language Features

In logic programming, a language compiler employs an algorithm which scans all probable patterns in a set of stated inference rules to postulate and resolve query. In event driven programming, the program responds to events. Whenever an event is received, an action is triggered automatically. The actions can be some computation or initiation of another event. In functional programming, some mathematical functions are evaluated and state changes are avoided. Such programming languages are stateless and provide referential transparency which means the output of a function will remain same for given parameters irrespective of when and where it is evaluated. The reactive programming provides abstractions that facilitate programs to respond to external events. The amalgamation of functional and reactive programming results in functional reactive programming that models reactive behavior in functional programming languages.

iv. Support for Abstractions: Abstraction means simplified description of a system, suppressing unwanted details while emphasizing some system required details. Abstractions help in handling complexity by emphasizing user required details and suppressing immaterial or diversionary details. Different languages provide different abstractions, and in this manuscript, we have analyzed top-tier SDN programming languages in terms of following four abstractions: consistency, modular composition, flow matching and traffic engineering.

Consistency ensures a packet is managed either with an old configuration or with a new configuration but not by a mix of two [33]. Hence, this abstraction ensures that a packet never traverse a path during the transition of network state from old to new. Modular composition is beneficial in handling complexity of programs. In modular composition, a program is decomposed into distinct modules with accurately identified boundaries and strictly controlled interactions. These modules have packet processing roles; they consume a packet history and produce a new packet history [34]. In order to group these modules, there are following three compositors: sequential, parallel and iteration. In sequential, one module operates on a packet produced by another module. Parallel enables multiple copies of same packet being operated concurrently by multiple policies. Iterative composition enables recursive or repeated processing of packets.

In SDN, forwarding devices have flow tables populated with flow entries, with each entry comprising of match, action and counters. A match occurs if the received packet header fields match a predefined flow table entry. For matching, two abstractions are provided by SDN programming languages: basic and arbitrary matching. If a language supports standard matching operations specified by the OpenFlow, it is termed as basic matching. On the other hand, in arbitrary matching, a match field may be wildcarded wherein bitmasks are defined as follows: 0 indicates a don't care condition and 1 indicates an accurate match. OpenFlow 1.0 supports exact match and prefix wildcards, whereas OpenFlow 1.1+ supports arbitrary wildcards using bitmasks.

On the other hand, the basic aim of traffic engineering is to optimize performance of networks or simplify reliable network operations. The common traffic engineering abstractions provided by programming languages are link failure, QoS and path selection. Link failure abstraction ensures reliable network operation in case of link failure by changing the forwarding behavior. QoS refers to a set of service requirements (e.g., bandwidth, delay, etc.) to be met by the network while transporting data flows. Path selection abstraction specifies which path a particular flow must or must not traverse within a network.

3.2 Top-tier Programming Languages

In this subsection, more than twenty top-tier SDN programming languages have been described. Some languages have been extended to achieve a particular goal or address certain limitation. We have presented all such extensions along with the features introduced.

Frenetic [35] is a declarative, high-level language with modular design for programming distributed collection of network devices. Frenetic is based on the following two key abstractions: first, a set of operators for construction and manipulation of network traffic streams and second, a run-time environment that handles all minutiae of installing and uninstalling low-level rules on switches. Most of the existing SDN programming languages are derived from Frenetic. It is based on following key principles: single-tier programming, race-free semantics and cost control. It is a functional reactive language, providing arbitrary wildcard matching and using combinatory library for defining high-level packet forwarding policies.

Flow-based Management Language (FML) [36] is a high level, declarative, reactive, policy description language for data flows using non-recursive datalog with negation [37]. It is the first SDN programming language and was initially termed as Flow-based Security Language (FSL) [38]. FML policy statements comprise of if-then relationships that specifies constraints for matching flows. This language statically defines which flow to allow or deny and has provisions for specifying performance parameters like latency, bandwidth and jitter.

Likewise, NetCore [39] is also a declarative language that eases process of expressing packet-forwarding policies. It enables efficient processing of majority of packets inside switches instead of controller by installing flow rules proactively rather than reactively. Using NetCore, the controller with new compilation algorithms and a new run-time system issues rule installation commands and traffic-statistics queries to switches. It is considered as a replacement of Frenetic [35] for defining forwarding policies. The authors in [40] have extended NetCore, termed as NetCore-M to implement multi-policy combination and packet drop action. Further, the authors have extended multi-policy combination algorithm to detect policy conflicts using pyretic [41] project.

Pyretic [41] is an imperative, domain-specific language embedded in Python which defines: (i) composition operators and a library of policies for forwarding and querying network traffic; (ii) a policy to operate on an abstract topology that implicitly constrains what a module can visualize and perform; and, (iii) a new abstract packet model that allows the programmers to extend packets with virtual fields which may be used to associate packets with high-level meta-data. Unlike Frenetic, which only supports sequential composition, Pyretic supports both sequential and parallel composition. Further, Pyretic provides topology abstraction which can represent a single underlying switch as multiple virtual switches or vice-versa.

Procera [42] is a policy layer on top of the SDN controller, which is event-driven, reactive and extensible. It provides flow constraint output functions that simplify the role of the network controller by enabling separation of high-level control from low level networking issues. It applies the principles of functional reactive programming (FRP) [43] and provides a declarative, expressive and compositional framework for describing reactive and temporal behaviors. It is customizable with a collection of primitive event streams and does not have default access to OpenFlow events, such as flow request events which may lead to functional division. It permits scalable implementations wherein OpenFlow events are processed in parallel while as non-OpenFlow events are processed sequentially.

Nettle [44] is a domain-specific language, embedded in Haskell which enables programming of OpenFlow networks in an elegant and declarative style. It is based on the principles of functional reactive programming (FRP), with both continuous (time-varying) and discrete (event-based) abstractions leveraged in the design. The discrete events may arise from the OpenFlow switches and continuous behavior enables creation of dynamic policies like load balancing, traffic engineering, etc. It provides both sequential and parallel composition.

NetKAT [45] is a network programming language which is based on canonical and mathematical structure known as Kleene algebra with tests (KAT). NetKAT supports primitives for filtering, modifying, and transmitting packets. It also includes union, sequential composition operator and an operator termed as Kleene star operator that iterates programs. It enables programmers to create compositional and expressive network programs and reason effectively about their semantics. It was the first SDN programming language to use regular expressions for defining end-to-end forwarding routes. Authors in [46] have extended NetKAT, termed as Probabilistic NetKAT which introduced functions to determine probability distribution on collections of packet histories. This enhanced version of NetKAT handles scenarios like faults, congestion and randomized forwarding. Likewise, authors in [47] have proposed Temporal NetKAT which is based on the semantics of NetKAT and offers an interesting blend of linear temporal logic and Kleene (network) algebra with tests. The Temporal NetKAT is a network programming language which supports network verification, runtime monitoring, and debugging.

[48] is another declarative language for computing network forwarding state. It separates logic specification from the state machine that implements the logic. The logic is defined in a declarative manner with a function mapping. The controller input to output and state transitions are handled by the language compiler that handles the event processing code and provides a runtime environment responsible for consuming the input events and re-computing of all affected outputs. On the other hand, FatTire [49] is also a declarative, regular expression-based language for writing fault tolerant network programs. It uses natural and orthogonal programming constructs to provide fast-failover mechanisms in OpenFlow networks. The distinguishing feature of this language is that it allows programmers to specify both forwarding paths and backup paths that the packets may take while traversing through the network.

Maple [50] simplifies SDN programming by using high level declarative constructs to define network policies. It involves two important components: (i) optimizer that offloads work to switches and invalidates cached policy executions due to environmental changes and; (ii) run-time scheduler which provides Maple scalable execution of policy "misses" generated by many switches in a large network on multi-core hardware. Authors in [51] have extended Maple called as Maple++, to implement a reactive programming paradigm which supports heterogeneous data plane devices with flow table pipeline. Maple++ provides a data structure (forwarding tree) maintained by the SDN controller to manage forwarding rules of switches in an integrated and centralized way. The forwarding tree is compressed and divided into multiple sub-trees based on the configuration. Afterwards, the sub-trees are mapped to multiple flow table pipelines.

Likewise, Merlin [52] simplifies network management by allowing network operators to specify network policies using declarative language, comprising of logical predicates and regular expressions. Like NetKAT, it uses regular expressions to specify forwarding routes and uses partitioning approach to convert global policies into smaller sub-policies for sub-domains of the network which are later on assigned to different tenants. It uses high-level abstractions, constraint solver and parameterizable heuristics to distribute resources such as paths and bandwidth (specifying max. bandwidth and assured min. bandwidth). Further, it proposes mechanism for verifying delegated sub-policies of tenants so that they do not violate global constraints.

PonderFlow [53] is an extension of Ponder language to specify security and management policies in distributed systems. It is a declarative language with its own grammar and uses ANother Tool for Language Recognition (ANTLR) [54] framework for lexical analysis, parsing and for defining Abstract Syntax Trees. It also provides constructs to define access control and authorization. On the other hand, Kinetics [55] is a structured language for defining network policies in the form of finite state machines (FSMs), which capture dynamics and are verifiable. States of FSMs define distinct forwarding behavior and event trigger causes transition between states. It also incorporates an event handler that listens to events and on occurrence of an event, triggers transition in policy which in turn updates the devices in data plane.

Frenetic-OCaml [56] is a new form of Frenetic [35] to define queries for reading information about the traffic statistics and topology changes. It maintains a consistent network state with a run-time system which polls switch counters, respond to events and get aggregated flow statistics. It supports proactive rule installation and is the only language which ensures that packets of a flow are forwarded either with a new policy or with an old policy but never with an amalgam of both. Further, it supports both parallel and sequential composition.

Network Overlay Framework (NOF) [57] language provides services and set of operations to define network behavior as per the application requirements. It provides assured bandwidth to traffic flows, creates and manages virtual overlays and specifies flows to redirect traffic to these virtual overlays. The operations supported by NOF language can be classified into three categories. First, matching operations done on packet header fields. Second, timing operations which specifies which service to be installed and how long it will last in the network. Finally, query operations allow reading network statistics like bandwidth usage, link states and transmission issues. Likewise, Software Defined Constrained Programming (SCOR) [58] provides QoS in SDN traffic management. It defines two layers at the northbound interface in which the top layer uses constrained programming language and the bottom layer provides QoS and traffic engineering services. SCOR is a declarative, constrained programming language, implemented in Minizinc [59].

In [60], authors have proposed a high-level rule-based language (HLRL) for defining network policy using XML. This XML based policy description is converted into java program with the help of XML translator which comprises of two phases: lexical analyzer and syntax analyzer. The former tokenizes the input and matches the tags, identifiers, attributes, constant values, etc, using regular grammar. On the other hand, syntax analyzer performs syntactic analysis of the input and if everything works well it generates java source code. This java program is forwarded to floodlight controller, which in turn define low-level instructions for forwarding devices.

Summary: All the SDN programming languages discussed in this sub-section are summarized in table 1 along with their prominent features and key highlights. The top-tier programming languages simplify programming in SDNs by providing different set of capabilities, functionalities and restrictions. There is significant intersection among the features of these languages and only few languages take advantage from new OpenFlow versions. It is clear from table 1 that most of these programming languages lack consistency measures and traffic engineering abstractions. Such features play a critical role in seamless network operations and automated network management.

Table 1. Summary of Top-tier SDN Programming Languages

Language	Implemented in	Extensions (if any)	Features							Main Highlights of Programming Language
			Network Policy Definition	Flow Installation	Programming Paradigm	Abstractions				
						Consistency Support	Composition	Flow Matching	Traffic Engineering Support (QoS/Link Failure/Path	
Frenetic [35]	python	-	Dynamic	Reactive	FR	No	S	B+ AW	-	Provides SQL like constructs to query network
FML [36]	C++, Python	-	Static	Reactive	L	No		B	QoS (L) - Using LC	Expresses and enforces static policies, supports bandwidth limits and path selection in enterprise networks.
NetCore [39]	Haskell	NetCore-M [40]	Dynamic	Both	FR	No	S	B + AW	-	Declarative language that allows network slicing.
Pyretic [41]	Python	-	Dynamic	Both	I	No	S/P	B + AW	-	It is an imperative, domain-specific language which supports virtualization.
Procera [42]	Haskell	-	Dynamic	Reactive	FR	No	S/P	B	-	Event-driven, reactive and extensible policy layer on top of a SDN controller
Nettle [44]	Haskell	-	Dynamic	Reactive	FR	No	S/P	B	-	Using FRP for specifying dynamic policies and supports sequential and parallel composition.
NetKAT [45]	OCaml	Probabilistic [46] and Temporal [47] NetKAT	Dynamic	Both	F	No	S/P/I	B + AW	- - Using RE	Uses KAT operators for filtering, modifying, and transmitting packets. Supports iterative composition
nlog [48]	C++	-	Dynamic	Both	E	No	S/P	B + AW	QoS, Link Failure, Using FHA	Used to log queries over multiple tables resulting in immutable tuples for detection and propagation of updates.
FatTire [49]	OCaml	-	Dynamic	Reactive	F	No	S/P	B	- Link Failure, Using RE	Regular expression based declarative language for defining fault tolerant network paths.
Maple [50]	-	Maple ++ [51]	Dynamic	Both		No	S/P	B + AW	-	Provides high level declarative constructs to define network policies.
Merlin [52]	OCaml	-	Dynamic	Reactive	F	No	-	B	QoS (L/G)	Provides access control and QoS with guaranteed bandwidth.
Ponderflow [53]	Java	-	Dynamic	Reactive	E	No	-	B	-	Extension of Ponder language for OpenFlow to provide access control.
Kinetics [55]	Python	-	Dynamic	Both	E	No	S/P	B + AW	QoS (L) -	Structured language using FSMs to define network policies.
Frenetic-OCaml [56]	OCaml	-	Dynamic	Both	FR	Yes	S/P	B + AW	-	Provides consistent updates.
NOF [57]	Python	-	Dynamic	Reactive	E	No	-	B	-	Provides abstractions to application developers.
SCOR [58]	MiniZinc [59]	-	Dynamic	Reactive	E	-	-	-	QoS (L/G) - Using LC	Provides QoS and traffic engineering interface for assured bandwidth, path selection etc.

Programming Paradigm: F: Functional, FR: Functional Reactive, E: Event-driven, I: Imperative, L: Logic; **QoS:** L: limit, G: Guarantee; **Modular Composition:** I: Iterative, P: Parallel, S: Sequential; **Flow Matching:** B: Basic, AW: Arbitrary Wildcard ; **Path Selection:** LC: Language Construct, RE: Regular Expression, FHA: Flow Hash Algorithm

4. Bottom-Tier Programming

Majority of the top-tier SDN programming languages are designed for network policy management and have predominantly considered OpenFlow-based data plane devices. Such programming languages do not support protocol-oblivious forwarding, multi-table pipeline processing and per-packet stateful processing. Moreover, the first-generation network programming was divided into following two parts: (i) stateless packet processing which could be examined statically, compiled and finally installed on OpenFlow devices, and (ii) stateful component managed by the controller.

However, this two-step programming model cannot support efficient stateful tasks like detection of DNS amplification, SYN flood attacks, etc., as packets must go back and forth to the controller.

Traditional SDN data plane lacks programmability and flexibility. As per the authors in [61], flexibility in data plane refers to: (i) *flow steering*: forwarding element is flexible if it supports both forwarding and copying of elements; (ii) *function placement*: ability to dynamically alter the position of network functions during operation; (iii) *function scaling*: regulating scalable resources as per the requirement; and; (iv) *function operation*: ensures seamless operation during configuration of programming devices. On the other hand, the data plane programmability depends on its depth and implementation [62]. The depth reflects the management activities like reserving, transcoding, provision for new protocols, etc., beyond packet forwarding. Implementation of a data plane determines degree of programmability like data plane device implemented completely in software is considered fully programmable whereas the one implemented in hardware offers limited programmability. In contrast to it, the researchers in [63] are of the opinion that the data plane programmability can be achieved with re-configurable hardware along with the programming languages. Hence, flexibility to regulate flows, functions and resources in data plane can be achieved with the help of programmability.

In the last few years numerous bottom-tier programming languages have been proposed which follow different programming models to map specific algorithms splendidly onto specialized hardware like ASICs. The bottom-tier programming models provide necessary abstractions for mapping diverse functionalities of data plane algorithms. In this section, we have first introduced the bottom-tier programming models and afterwards various programming languages are examined in detail.

4.1 Bottom-tier Programming Models

The user-defined packet processing logic/algorithms are expressed by using bottom-tier programming languages. To map such algorithms onto specialized hardware, different programming languages use specific programming models to express packet processing logic in an abstract way. In this subsection, we have discussed the three commonly used packet processing models in bottom-tier programming.

i) Data-Flow Graph Abstraction: The Data Flow Graph (DFG) expresses the packet processing logic in terms of a graph, in which vertices represent computation phases and edges represent the data flow from one phase to another. This abstraction allows the programmer to express the well-defined processing phases into meaningful programming constructs following graph-oriented base model. These programming constructs are defined only once and can be re-used as per the requirement, hence provides rapid development and modular functionality. Therefore, this abstraction simplifies the design of expressing packet processing logic, however with increased heterogeneity it becomes complex to generate overall abstractions and hinders network-wide performance optimizations [64].

The programmable switches that follow this programming model are click [65], Berkeley Extensible Software Switch (BESS) [66], Vector Packet Processing (VPP) [67], and NetBricks [68].

ii) *Match-Action Abstraction:* The match-action abstraction or Protocol Independent Switching Architecture (PISA) uses a sequence of tables arranged into a hierarchical organization [5, 69]. More precisely, it is based on re-configurable match-action tables (RMTs) [70] and disaggregated re-configurable match-action tables (DRMTs) [71]. The multi-table pipeline processing begins with packet parser followed by match-action pipeline and ends with deparser. These programmable components perform the following functions:

- The *parser* helps programmers to declare different header fields along with a finite state machine which specifies the order of such fields within a data packet. Further, it transforms serialized packet header fields into well-organized form.
- The multi-table *match-action pipeline* involve multiple match-action tables to perform packet field matching and corresponding actions are initiated on the matching packet. Intrinsically, it uses SRAM or TCAM to store key values and the related action data. The action may be an arithmetic operation or a field modification which is implemented with the help of an arithmetic logic unit. Further, the stateful operations within the programmable switch are implemented using counters, registers or meters which are stored in SRAM. The control plane controls the runtime switch behavior by defining entries in the match-action pipeline.
- The *deparser* allows programmer to control packet serialization.

A data packet usually involves packet payload and metadata. The metadata consists of packet headers, intrinsic and user-defined metadata. The intrinsic metadata involves the packet related fixed function component whereas the user-defined metadata is developer friendly information which may be used in the entire packet processing pipeline. The metadata is transient which means it is discarded when a packet departs from the processing pipeline. All the aforesaid three programmable components process the metadata and not the payload which travels separately.

The widely used OpenvSwitch implements match-action pipeline with comprehensive flow-caching founded data path. However, to achieve line-rate software switching specialized template-based data-path compilation is introduced in ESwitch.

iii) *Hybrid Abstractions:* Over the last few years, the difference between the match-action abstraction and data-flow graph has become blurry. The software/hardware switches have emerged that involve the combination of these two abstractions. Such hybrid architectures combine the concepts from multi-processor design and distributed systems. Nowadays, the packet processing logic is implemented using graphics processing units (multi-threaded hardware) or extends the RMT towards a more adaptable architecture [71].

4.2 Bottom-tier Programming Languages

The network programs which manage data packets in forwarding plane need to operate at line-rate and are typically implemented in low-level or assembly language to control vendor-specific network processors. The authors in [72] attempted to simplify and streamline this software development process and free network programmers from managing low level hardware details. They used packetC [73], a high-level language and applied it on general parallel packet processing model. This model implements SPMD parallelism to relieve programmer from process thread management. The packetC language supports new data types and operators to abstract and encapsulate the well-known packet processing operations and data sets. The model comprises of global/private memories, data stores and host system to store packet programs, application domain information (like protocol, session, search data, etc.) and calculates layer offset, respectively. The packetC language is based on following imperatives: (i) uses operators, conditional statements of C99 variant of C language; (ii) removes address operators, pointers and dynamic memory allocation to evade runtime crashes; and (iii) uses strict typing rules (avoids promotions or implicit type conversions) to increase reliability.

The authors in [74] argue that the traditional SDN using OpenFlow does not sufficiently decouple the control and data plane as the forwarding elements require protocol-specific knowledge and innovations/extensions by user-defined protocols still require tunneling or overlay models. More specifically, the OpenFlow lacks stateful packet processing and offers limited expressivity which hampers the forwarding plane programmability. They have proposed Protocol Oblivious Forwarding (POF) in which data plane devices extricate search keys from the packet headers as per the controller's instructions and such keys are searched inside the table to execute the pre-installed instructions. These instructions are defined in Flow Instruction Set (FIS) language which allows to define keys in terms of packet headers, contents of flow tables and counters for statistics. Therefore, this FIS based definition of fields/protocols for processing data flow packets using search key tuples in the data plane enables programmability and flexibility in the data plane. However, with such key tuples it is difficult to handle variable header length, variable header sequence and consequently dynamic multi-table pipeline for packet processing [75].

One of the most popular and widely used data plane programming language is P4 [21, 22, 76]. It was introduced to achieve protocol independence, switch reconfigurability and independence from underlying forwarding hardware. It revolutionized the data plane programmability by enabling programmers to define data plane operations like parsing, field matching and actions flexibly using programming constructs. Till now, the Language Design Working Group (LDWG) of P4 has standardized two distinct standards of P4: $P4_{14}$ and $P4_{16}$. The timeline of these two standards is shown in Fig. 5 and the major differences can be observed in Fig. 6.

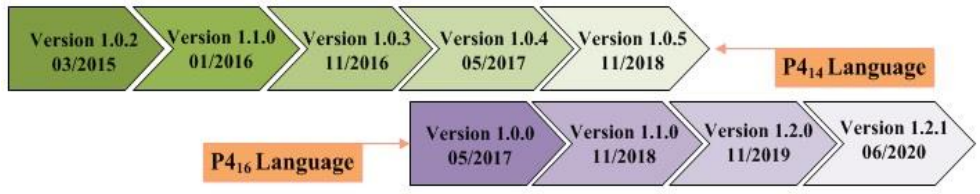

Fig. 5. Chronological history of $P4_{14}$ and $P4_{16}$

The $P4_{14}$ language allows programmers to use combination of both imperative and declarative constructs to define data plane functions like meters, counters, header checksum, etc. It provides in excess of 70 keywords and follows a specific packet processing model based on PISA. However, it fails to provide mechanisms to handle various architectures, involves loose semantics and weak typing. It provides limited support for modularity and supports relatively low-level constructs. To address these limitations, $P4_{16}$ separates core language constructs from the essentials of a specific architecture, as shown in Fig. 6. It introduces architecture-specific features in a library usually provided by the target vendor to define interface, structure and capabilities of a specific data plane packet processing pipeline. The core language components are also refined into a core language library and small set of constructs. In contrast with $P4_{14}$, $P4_{16}$ provides enhanced data structures, better modularity, removed declarative constructs, hence making it flexible and effective to better reason about the data plane programs. Consequent to such advantages of $P4_{16}$, $P4_{16}$ is mainly used and $P4_{14}$ has become obsolete.

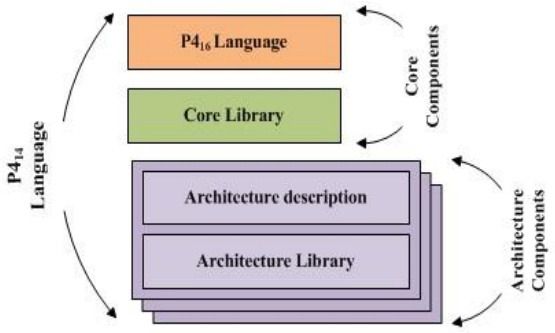

Fig. 6. Comparison of $P4_{14}$ and $P4_{16}$

The overall elaboration and implementation of P4 programs can be expressed as shown in Fig. 7 [77]. P4 targets or nodes can be either software based or specialized hardware which implements packet processing pipelines with the help of fixed function or programmable components. The pipeline structure is target specific described by a specific architecture model. The user-defined P4 programs are implemented as per the architecture model and are compiled as per the target specification. Both the architecture model and P4 compiler is provided by the node manufacturer. The compiler also defines the southbound API which will be used by the control plane to control and manage the run-time behavior of the target. Stratum [78] is an open-source NOS which includes implementation of P4 runtime and OpenConfig interface. Currently, the most prominent P4 targets are: (i) *software-based:* T_4P_4S [79] (uses Data Plane Development Kit (DPDK) [80] or Open Data Plane (ODP) [81]), PISCES [82] (converts Open vSwitch (OVS) [83] into software P4 target), Ripple [84] (based on DPDK) and P4rt-OVS [85]; (ii) *FPGA-based:* Netcope P4 [86], P4FPGA [87]; (iii) *ASIC-based:* Intel Tofino, Pensando Capri [88].

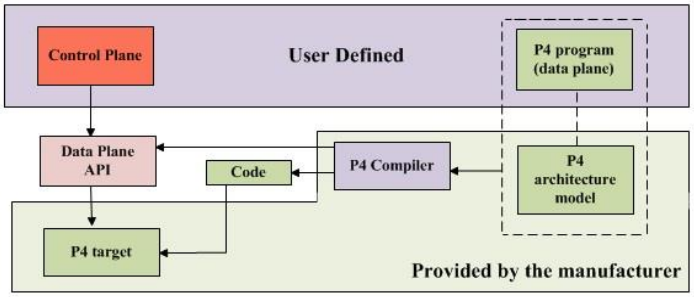

Fig. 7. Implementation of P4 as per [77]

More specifically, P4 is a programming language for unrestrained packet processors. The primary focus of P4 is programmable data plane and basic motivation is to make data plane switches reconfigurable, protocol oblivious and hardware independent. The P4 program consists of following constructs: i) *Headers:* which specify the series of fields with constraints on values and widths, ii) *Parsers:* which ensure how to determine valid header sequence within a packet, iii) *Tables:* to define match and action fields, and iv) *Control Programs:* to specify order of match-action tables.

Stateful Network-Wide Abstractions for Packet Processing (SNAP) [89] is a network programming language with two novel features stateful network programs and switch virtualization. It enables programmers to write programs for virtualized switches which follow an algebraic pattern as observed in top-tier NetKAT/NetCore languages. In SNAP, each program comprises of one or more policies and predicates. Its semantics are defined with the help of an evaluation function "eval" which determines how a packet should be processed by SNAP program. The input to eval function is a packet and a starting state and it yields a packet or set of packets and an output state. It uses extended forwarding decision diagrams (xFDD) to detect program errors like race conditions and generates mixed integer linear program from xFDD to decide state array placement and network routing to reduce network traffic congestion and at the same time satisfying constraints essential for network transactions. It supports both sequential and parallel composition.

The authors in [90] have proposed Domino, a C like language with numerous constraints for deterministic performance. It is an imperative programming language which allows programmers to write programs in terms of packet transactions. Such packet transactions are indivisible successive code blocks which specify how to process state and packet headers. The packet transactions are executed on a new machine architecture termed as Banzai to achieve line-rate packet switching. The Banzai system comprises of an ingress and egress pipeline. Packet parsing takes place before entering the pipeline for processing and is out-of-scope of the Banzai system. The packet processing pipeline involves many stages which contain atoms (programmable units for packet processing). The Domino compiler consists of three phases: (i) *pre-processing:* converts packet transactions into three address code; (ii) *pipelining:* transforms three address code into an intermediate representation which has no resource or computational limits; and (iii) *code generation:* converts the intermediate code into Banzai machine configuration.

The researchers in [91] have presented an intermediate bottom-tier programming language to implement intricate programming constructs of languages like P4 on different hardware platforms. The primary goal of researchers was to have an intermediate language which must be expressive enough to support high-level, front-end programming constructs and at the same time must be functional enough to support various hardware targets. The intermediate language termed as NetASM in [91] provides instructions which are executed concurrently or sequentially on varied level of persistent packet states at the packet processing pipeline level. These instructions are 25 in number and enables to perform: (i) data storing, (ii) data loading, (iii) calculation, (iv) branching, (v) packet header related operations, and (vi) distinct operations like checksum and hash. With the help of packet related state, the NetASM language can parallelize the packet processing pipeline.

The authors in [92] present a high-level declarative language PX to program FPGA-based computing platform. PX follows object-oriented semantics like C++ which allows programmers to define packet processing logic with component objects and communication among such objects. The component objects are of two types: engine and system. The engine performs functions like parsing, lookup, editing and specialized computations whereas the system acts as communication engines or/and subsystems. The communication takes place with two well defined interface objects packets and tuples. The former carries packets between components whereas the later transports data tuples between components. Engine objects operate upon tuples and packets which are transferred over engine APIs.

In brief, the forwarding systems can include numerous engines with intricate communication patterns. The PX compiler guarantees flow and timing control for the information exchange among components. The compiler generates RTL based customized reconfigurable architecture in VHDL or Verilog which satisfies both operational and functional necessities. To update functions dynamically at runtime, the reconfigurable architecture supports custom firmware.

Another community driven data plane programming language is NPL (Network Programming Language) [93]. NPL is a specialized language which effectively addresses the different requirements of programming forwarding planes. It involves necessary programming constructs to implement robust networking behavior which can take advantage of advanced programmable hardware. The basic building blocks include data types which allow the design of distinct control signals to high-level constructs that allow interfacing with intricate hardware blocks [94]. It is not innately bound to a particular hardware architecture and can be used to program FPGAs, ASICs, programmable NICs and software switches. It involves two step compilation and associated tools in which the front-end compiler transforms NPL program into intermediate representation (IR) and the back-end compiler maps IR to individual hardware objects. The back-end compiler also specifies the API which the SDN controller uses to control the run-time switch behavior.

With the help of NPL, the users can specify flow table details and other necessary objects in data plane to realize the required behavior. The core abstractions in NPL include data types, parser, logical bus, logical table, header editor, strength resolution, control functions and metadata for match-action table. With these abstractions and other sophisticated features, a programmer can define customized flow-table pipelines, advanced logical table capabilities, parallelism, intelligent actions to achieve simple and instinctive control flow. It also provides constructs to include component libraries which implement static task hardware blocks. Such feature enables programmers to define simple table-based architectures to more complex and advanced architectures which involve well-organized and effective building blocks.

Lyra [95] proposes high-level abstractions for data center networks to achieve extensibility, portability and composition in data plane programming. It provides one-big pipeline abstraction to programmers to express the packet processing logic in a target-agnostic and chip-neutral way. This programming abstraction provides simple and expressive mechanism to define how traffic flow packets involving different fields will be managed along a chain of algorithms. Every algorithm follows a tree-like structure to express packet processing logic (read/write/arithmetic operations) with simple if-else statements. With Lyra language, the programmers can explicitly specify the scope of an algorithm on target candidate switches which is a useful feature to direct final compilation/deployment through high-level intents. Lyra compiler combines lyra program, low-level ASIC details and algorithm scopes to generate target specific code as per the topology information.

Unlike other approaches [89, 96] which use Integer Linear Programming (ILP) for resource allocation, the authors in Lyra compiler encode all packet processing logic and flow constraints into satisfiability modulo theories (SMT) and use SMT resolver to determine the best implementation and deployment scheme for a high-level program in a target network. This process involves following three steps: (i) translation of lyra program into context-aware Intermediate Representation (IR); (ii) transforms context-aware IR into conditional language specific implementation onto corresponding target switches and; (iii) uses SMT formula to encode all resource and placement restraints in order to determine the target-specific implementation and algorithm placement, simultaneously. The authors have evaluated it and have observed that it generates less lines of code and requires less hardware resources.

Summary: All the bottom-tier SDN programming languages discussed in this section are summarized in table 2. The bottom-tier programming languages simplifies packet processing logic design with new and advanced features, constructs and abstractions. PacketC [72] and POF [74] mainly focused on simplifying packet processing programming and protocol oblivious forwarding respectively, however these approaches lack measures to handle complex packet parsing required in DPI, variable header length, variable header sequence and consequently dynamic multi-table pipeline. Likewise, SNAP [89], Domino [90], NetASM [91], PX [92] have addressed a particular issue of packet processing programming. On the other hand, P4 provides a complete design and architecture to express headers, packet parsers, tables and control programs to achieve reconfigurable, protocol oblivious and hardware independent data plane forwarding. However, P4 mainly focuses on simplifying programming of FPGA-based devices and lacks features to define complex network functions which require to maintain per-flow state. Although approaches like NPL [93], P4 [21, 22] and Lyra [95] have introduced advanced features and have addressed numerous issues but we believe that the bottom-tier programming is still in early stage of development which requires further research contribution to address various open issues which we have highlighted in the next section.

5. Future Research Perspectives in SDN Programming

With the advancements in SDNs, several programming languages have been proposed to simplify design, operation and management of resources in diverse network planes. Initially, the researchers addressed the network management issues of conventional networks with centralized SDN control plane. Subsequently, first generation programming languages were proposed with the aim to express policy directives using simple programming constructs and then compile such programs statically to generate flow table rules effectively for each data plane device. However, subsequently the network operators realized the need of flexible solutions which can adapt to changing network requirements like telemetry features and novel protocols. This forced the network designers to introduce programmability in SDN data plane as well. Consequently, numerous bottom-tier programming languages were proposed to define packet processing logic and control functions which can flexibly and dynamically regulate resources in SDN data plane. Although few top-tier and bottom-tier programming languages introduced new abstractions, novel programming constructs and advanced features, however there are still many open issues and research challenges in SDN programming which we have highlighted in this section.

5.1 Adaptable cross-platform programming

The logically centralized network control in SDN has simplified intricate global routing by flexible and advanced traffic engineering and fast re-routing. The success of SDN is commonly attributed to control plane innovations; however, with new reconfigurable switch designs and rich data plane programmability along with their APIs can provide promising ways to define packet processing logic in data plane and management of resources in control/management plane. The current trends in data plane programmability involve hardware description languages (like P4, NPL, etc.) and programming languages which implement packet-processing algorithms directly on specialized hardware. The former involves complicated compilation and arduous development process whereas the later achieves function operation flexibility but at the cost of destination platform dependency. The two open problems in bottom-tier programming are:

i. the existing data plane programming languages are not fully proficient to abstract the details of majority of destination platform that would ensure freedom from the destination hardware platform and enables defining intricate forwarding processes with granularity, and
ii. the lack of sophisticated and adequate constructs in current data plane languages which would allow comprehensive programmability of every data plane process.

In brief, to achieve high-level, portable and cross-platform programmability in data plane we believe an integrated approach of P4 [21, 22] and ODP [81] is required. P4's capability of defining packet processing pipelines along with ODP's APIs can provide standard features across diverse platforms like ARM, Power PC, and x86. P4 can hide complexity of data plane programming where as ODP can provide scalable and portable data plane with hardware acceleration features. Therefore, merging ODP APIs with P4 primitives and abstractions is one promising future research direction which can simplify data plane programmability.

On the other hand, the top-tier programming languages (like Frenetic, Pyretic, Procera, etc.) have mainly focused on a specific data plane device (mainly OpenFlow complaint devices) and the programs written in such languages can hardly handle diverse data plane devices. The main problem with such languages is monolithic compilation and lack of modularity [97]. To address this issue the high-level programs should be either translated into an intermediate code and then the device specific code should be generated from that intermediate code [97] or an intermediate abstraction layer should be introduced between the control and data plane [98]. However, both these solutions may increase the control latency in dense networks.

5.2 Programming language libraries

Table 2. Summary of Bottom-tier SDN Programming Languages

Language/ Versions	Programming Model	Programming constructs	Prominent Features	Main Highlights
PacketC [72]	SPMD	C99 variant of C language	uses strict typing rules and dynamic memory allocation	It relieves programmer from process thread management and supports new data types and operators to abstract and encapsulate the packet processing
POF [74]	DFG	Generic Flow Instruction Set	- Search keys = {offset, length} tuples - Dynamic multi-table pipeline	The controller defines instructions in Flow Instruction Set (FIS) language which allows to define keys for packet processing
P4 [21,22] / $P4_{14}$, $P4_{16}$	PISA	Headers, parsers, match-action controls, match action tables and deparser	-flexible and extensible architecture agnostic programming language - translates P4 programs to target-specific code using the corresponding architecture model.	Enables programmers to define data plane operations like parsing, field matching and actions using simple programming constructs
SNAP [89]	OBS One Big Switch	eval function	- Persistent global arrays - network virtualization - consistent network transaction	Provides stateful network programs written for one-big-switch using abstract topology
Domino [90]	DFG	C like constructs with restrictions	Constrained programming with three stages for line-rate switching -pre-processing -pipelining -code generation	Enable programmers to express packet processing logic in terms of packet transactions which are indivisible, isolated and sequential code blocks.
NetASM [91]		Twenty-five specialized instructions	-Introduces packet level state to parallelize the pipeline processing	An intermediate language to support high-level programming constructs and functional enough to support various hardware targets.
PX [92]	PISA	Object-Oriented semantics	-component objects (engines and system) and -communication among them	The forwarding device includes numerous engines with intricate communication patterns
NPL [93]		data types, parsers, logical bus, logical table, header editor, strength resolution, etc.	-basic constructs and advanced constructs -Special functions -intelligent actions -Instrumentation and -Runtime programmability	NPL is a construct rich programming language which provides sophisticated features like parallelism and special functions to design highly efficient forwarding pipeline architectures.
Lyra [95]	One Big Pipeline	High level constructs	-IR instructions -Instruction dependency and -Deployment constraints	Lyra provides a one-big pipeline abstraction to data plane programmers to express the packet processing logic in a target-agnostic and chip-neutral way.

Majority of the top-tier programming languages support OpenFlow 1.0 with limited code reusability and modularization, and need to be extended to provide support for latest OpenFlow versions. Likewise, most of the bottom-tier programming languages are in early stage of development and provide elementary constructs and negligible code reusability. Many high-level programming languages like java, python, etc., have received widespread acceptance due to the standard library support and community-driven extensions. Inclusion of new features in such programming languages is very simple and is usually achieved by importing modules into repositories. However, the existing SDN programming languages lack such features and have been designed to address a particular network problem like fault tolerance in FatTire [49], access control and QoS in Merlin [52], end-to-end stateful monitoring function in SNAP [89], etc. The SDN programming languages do not provide a well-supported and feature rich interface to simplify network programming wherein new modules can be developed and integrated with the existing code. Further, we believe that SDN programming language libraries need to developed and existing libraries need to be extended and updated regularly.

5.3 Stateful and in-line packet processing

The transition from rigid OpenFlow based data plane programming model to more flexible and programmable packet processing architectures (like P4, NPL, etc.) have not only revolutionized the packet processing functionality but is envisioned to open new dimensions in critical packet processing operations like dynamic load balancing, multipath routing, traffic flow isolation, etc. To scale network systems for dynamic operations like fluctuating traffic demands or congestion avoidance and massive workloads, forces network designers to push network state information into data plane devices. However, these in-line network computations open doors for novel distributed consensus protocols [99], consistent updates [100], network accelerated key-value stores [101], automated traffic regulation with neural networks [102], etc. The existing bottom tier programming languages like P4 provide limited constructs for stateful functions [103]. The stateful abstractions are in early stage of development and the major challenge here is tradeoff between consistency and performance. The primary performance issue in state-of-the-art computing systems is reading and writing to the memory [104]. Currently, following three approaches are used for memory abstraction: application-based, executor-based and consistency-based.

The application-based state abstraction exposes state to common abstractions found in network applications. OpenState [105] follows application-based approach in which flow state is defined in addition to global state. The executor-based abstraction exposes a simplified view of underlying architecture to programmer. The packetC [72] uses this approach by partitioning memory into global and processor-local. On the other hand, the consistency-based memory abstraction organizes memory as per the consistency requirement which provide either strong, eventual or no consistency. For instance, Domino [90] offers only strong consistency at the cost of restraining supported procedures at line rate. All the three approaches follow different trade-offs. For instance, the application-based memory abstraction simplifies program development by providing access to most of the applications at the cost of limited support for experimentation with novel solutions whereas the executor-based provides most flexibility to programmer at the cost of reducing portability and increased programming complexity.

Another possibility is to keep data plane stateless and store the switch-state into an external data-store [106]. The data-store maintains the data plane state and a forwarding device can fetch state information dynamically from it. This external data-store architecture enables scale-out state migration, replication and restoration. However, for line-rate operations some state information is still required to be stored locally into a local cache and programming abstractions are required to handle this cache. Apart from state management, another interesting research direction is the integration of network functions with high-level SDN programming to achieve efficient and adaptive resource utilization and real-time performance aware routing (like contra [107]). However, to realize unified programming (combining top-tier programming with network function state), the following open issues/research challenges need to be addressed by the research community:

- How to integrate SDN programming with network function state.
- How to achieve efficient and accurate dependency tracking in unified SDN programming.
- How to construct correlated and consistent routes using network state information flexibly.

5.4 Network verification

The network verification is a critical issue in network management. Network verifications ensures automatic network correctness both in data plane [108-112] and control plane [113-117]. The control plane verification mechanisms mainly focus on protocol misconfiguration detection whereas the data plane verification checks the actual packet forwarding behavior, therefore detects a broad range of bugs in software and hardware.

Numerous data plane verification mechanisms allow network operators to verify the forwarding accuracy in real-time [109, 111, 112]. For real-time analysis, the data plane verification process very often divides packets into different equivalence classes and maintains forwarding behavior models for each such class. When a forwarding behavior update takes place, the verifier updates the model and verifies the updated model for correctness. Some verification frameworks [109, 112] take sub-millisecond time for network verification but assume simple network models considering only forwarding function of data plane devices. However, real data plane devices perform number of functions, simultaneously. On the other hand, some verification models [110, 111] consider many network functions but are very difficult to extend. In brief, in a dynamically programmable network architecture wherein we can define on-demand packet processing logic and control logic, we require new scalable models and verification algorithms which can handle various real data plane devices.

5.5 Abstractions for IBN (Intent-Based Networking)

The IBN is an important development towards shaping and managing networks in terms of high-level business objectives, and let the network handle low-level concerns in a flexible, automated, secure and agile way. The IBN involves translation of high-level business policies, implementation of these policies, awareness of network state and assurance of desired state [118].

Although the advances in top-tier programming languages is envisioned to realize the vision of IBN by introducing well-organized, effective language constructs and modular composition [41, 50, 55], however, it is still unclear how to expose forwarding functionalities to the network operator offering highest programming liberty while hiding the underlying intricacies effectively. The need of the hour is to have an intent based bottom-tier compiler which would abstract the data plane details to provide effective and efficient resource utilization, built with the vision of optimizing forwarding programs and overall network performance [119]. On the other hand, there should be upward mapping which will enable real-time monitoring in control plane and verify the operations of the data plane. For this closed control-data plane loop, new abstractions and special programming constructs are required in both bottom and top-tier programming which permit the intent layer to determine the operational context including overall network load/state, to validate operations precisely, to discover performance anomalies which may originate from malicious actions.

5.6 Abstractions for Network Function Virtualization

SDN and Network Function Virtualization (NFV) greatly complement each other, as the former takes away network control logic from forwarding functions whereas the latter advocates network functions (intrusion detection, firewall, network address translation, etc.) on commodity servers instead of specialized hardware. Numerous studies [120-126] have proved how SDN's dynamic implementation of packet processing logic in data plane and

programmability in control plane can improve fault tolerance and elasticity in NFV systems. Integration of network functions like extraction of application layer header information with SDN programming can make adaptive and cross-layer SDN control plausible [127]. However, to fully realize the benefits of SDN programmability in NFV systems new programming constructs and novel abstractions are required in SDN programming languages [128].

5.7 Programming support for hybrid networks

The hybrid networks or transitional networks comprise of amalgamation of programmable and legacy devices [129]. These networks are envisioned to pave-way for deployment of open programmable forwarding devices into traditional networks or progressively changing from legacy distributed control to logically centralized SDN control [9]. However, to the best of our knowledge, neither a single top-tier programming language nor a bottom-tier programming language has been proposed for hybrid networks. We believe that such programming languages can greatly simplify policy definition, packet processing and efficient resource management in hybrid networks.

6. Conclusion

Modern network systems have advanced to incorporate immense agility, reconfiguration and programmability. Numerous approaches, techniques and implementations over the years have resulted into SDN paradigm which provides strong isolation of different network planes along with centralized network control and high-level of programmability. The programmability in SDN offers flexibility in defining various functions and procedures in diverse network planes without modifying the other layers. Numerous top-tier and bottom-tier programming languages have been proposed in the last one decade which offer different features, properties and capabilities. In the first portion of this manuscript, we have introduced the top-tier programming language features and have analyzed various such programming languages. Afterwards, in the second half of this manuscript, we have evaluated the data plane programming languages in terms of supported programming models, programming constructs and features.

The top-tier SDN programming languages have mainly focused on network policy management. We have observed that these languages provide better and less error prone alternative to specify high-level network policy directives in a flexible application development environment. However, most of these languages have only considered OpenFlow based data plane devices and have taken limited advantage from latest versions of OpenFlow. We observed that these languages lack community and library support, hence provide only basic constructs to developers, who often have to develop an application from scratch. Further, majority of top-tier programming languages lack consistency measures and traffic engineering abstractions. On the other hand, bottom-tier or data plane programming languages involve hardware description languages (like P4, NPL, etc.) and programming languages which implement packet-processing algorithms directly on specialized hardware. The former involves complicated compilation and arduous development process whereas the later achieves function operation flexibility but at the cost of destination platform dependency. We believe that the bottom-tier programming languages are still in early stage of development and face numerous challenges in terms of cross platform support, dynamicity, stateful packet processing and efficiency.

References

[1] A.T. Campbell, H.G.D. Meer, M.E. Kounavis, K. Miki, J.B. Vicente and D. Villela, " A Survey of Programmable Networks" , ACM SIGCOMM. Computer Communication Review, vol. 29, no. 2, pp 7-23, 1999.

[2] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall and G.J. Minden, "A Survey of active network research", IEEE Communication Magazine vol.35, no. 1, pp: 80-86, 1997.

[3] N. McKewon, "Programmable Forwarding Planes are Here to Stay", in the proceeding of NetPL, 2017.

[4] "Software Defined Networking: The New Norm for Networks", ONF White Paper, April 13, 2012.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks", SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[6] Y. Jarraya, T. Madi, and M. Debbabi, "A Survey and a Layered Taxonomy of Software Defined Networking", IEEE Communications Surveys and Tutorials, vol. 16, no. 4, pp. 1955–1980, 2014.

[7] A. Hakiri, A. Gokhale, P. Berthou, D. C. Schmidt, and T. Gayraud, "Software Defined Networking: Challenges and Research Opportunities for Future Internet", Computer Networks, volume 75, no. Part A, pp. 453– 471, Dec. 2014.

[8] H. Farhady, H. Lee, and A. Nakao, "Software Defined Networking: A Survey", Computer Networks, volume 81, no. supplement C, pp. 79–95, Apr. 2015.

[9] S. Ahmad, A. H. Mir, "Scalability, Consistency, Reliability and Security in SDN Controllers: A Survey of Diverse SDN Controllers", Journal of Network and Systems Management, Springer, vol. 29, No. 9. ,doi.org/10.1007/s10922-020-09575-4, 2020

[10] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," IEEE Communications Surveys & Tutorials, vol. 20, no. 1, pp. 333–354, 2018.

[11] Y. Yu, X. Li, X. Leng, L. Song, K. Bu, Y. Chen, J. Yang, L. Zhang, K. Cheng and X. Xiao, "Fault Management in Software Defined Networking: A Survey", IEEE Comm. Surveys & Tutorials, vol. 21, No. 1, pp. 349 – 392, Sept. 2018.

[12] A. Mendiola, J. Astorga, E. Jacob, and M. Higuero, "A Survey on the Contributions of Software Defined Networking to Traffic Engineering", IEEE Comm. Surveys & Tutorials, vol. 19, no. 2, pp. 918– 953, 2017.

[13] S. S. Hayward, S. Natarajan and S. Sezer, “A Survey of Security in Software Defined Networks”, IEEE Comm. Surveys & Tutorials, vol. 18, no. 1, pp. 623– 654, 2016.
[14] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti, “A Survey on the Security of Stateful SDN Data Planes”, IEEE Comm. Surveys & Tutorials, vol. 19, no. 3, pp. 1701–1725, 2017.
[15] R. Alvizu, G. Maier, N. Kukreja, A. Pattavina, R. Morro, A. Capello and C. Cavazzoni and E. Keller, “Comprehensive Survey on T-SDN: Software-Defined Networking for Transport Networks”, IEEE Communication Surveys & Tutorials, vol. 19, no. 4, pp. 1701–1725, 2017.
[16] A. C. Baktir, A. Ozgovde, and C. Ersoy, “How Can Edge Computing Benefit from Software-Defined Networking: A Survey, Use Cases & Future Directions”, IEEE Comm. Surveys & Tutorials, vol. 19, no. 4, pp. 2232 - 2283, June, 2017.
[17] K. M. Modieginyane, B. B. Letswamotse, R. Malekian, and A. M. Abu-Mahfouz, “Software Defined Wireless Sensor Networks Application Opportunities for Efficient Network Management: A Survey”, Computers & Electrical Engineering, vol. 66, pp. 274–287, Feb. 2018.
[18] S. Bera, S. Misra, and A. V. Vasilakos, “Software Defined Networking for Internet of Things: A Survey”, IEEE Internet of Things Journal, vol. 4, no. 6, pp. 1994–2008, Dec. 2017.
[19] C. Trois, M. D. D. Fabro, L. C. E. de Bona, “A Survey of SDN Programming Languages: Towards a Taxonomy”, IEEE Communication Surveys & Tutorials, vol. 18, no. 4, 2016.
[20] E. Kaljic, A. Maric, P. Njemcevic, and M. Hadzialic, “A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking,” IEEE ACCESS, vol. 7, 2019.
[21] F. Hauser, M. Haberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank and M. Menth, “A Survey on Data Plane Programming with P4: Fundamentals, Advances and Applied Research”, arXiv:2101.10632v1 [cs.NI], 2021.
[22] P. Bossharty, D. Daly, G. Gibby, M. Izzardy, N. McKeownz, J. Rexford, C. Schlesinger, D. Talaycoy, A. Vahdat, G. Varghesex, D. Walker, “P4: programming protocol-independent packet processors**”,** ACM SIGCOMM Computer Communication Review, vol. 44, no. 3, July 2014.
[23] ONF, “Open Networking Foundation”, (online) Available:https://www.opennetworking.org/
[24] S. Ahmad, A. H. Mir, “SDN Interfaces: Protocols, Taxonomy and Challenges”, International Journal of Wireless and Microwave Technologies, DOI: 10.5815/ijwmt.2022.02.02, 2022.
[25] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, and G. Parulkar, “Onos: Towards an open, distributed sdn os”, in Proceedings of the third ACM Workshop on Hot Topics in SDN, USA, pp. 1–6, 2014.
[26] Floodlight Project. [Online]. Available: http://www.projectfloodlight.org/
[27] OpenDayLight Project. [Online]. Available: http://www.opendaylight.org/
[28] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, “Onix: A distributed control platform for large-scale production networks”, in Proceedings of the Ninth USENIX Conference on Operating Systems Design and Implementation, pp. 1–6, 2010.
[29] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, “Reconfigurable Network Systems and Software-Defined Networking”, in the Proceedings of the IEEE, vol. 103, no. 7, pp. 1102–1124, July 2015.
[30] G. N. Stone, B. Lundy, and G. G. Xie, “Network policy languages: A survey and a new approach”, IEEE Network, vol. 15, no. 1, pp. 10–21, Jan/Feb 2001.
[31] J. Qadir and O. Hasan, “Applying formal methods to networking: theory, techniques, and applications”, IEEE Comm. Surveys & Tutorials, vol. 17, no. 1, pp. 256–291, Mar. 2015.
[32] A. Mottola, “Design and implementation of a declarative programming language in a reactive environment,” Ph.D. dissertation, Dept. Comput. Eng., Universit àdegli Studi di Roma, Rome, Italy, 2005.
[33] M. Casado, N. Foster, and A. Guha, “Abstractions for Software-Defined Networks”, ACM Commun., vol. 57, no. 10, pp. 86–95, 2014.
[34] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha, “A fast compiler for NetKAT,” in Proc. 20th ACM SIGPLAN International Conf. on Funct. Program., Vancouver, BC, Canada, pp. 328–341, 2015.
[35] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: a network programming language*”,* SIGPLAN, 2011.
[36] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, “Practical Declarative Network Management”, in Proceedings of First ACM workshop on Research on Enterprise Networking, 2009.
[37] N. Li and J. C. Mitchell, “DATALOG with constraints: A foundation for trust management languages,” in Practical Aspects of Declarative Languages. Heidelberg, Germany: Springer, pp. 58–73, 2003.
[38] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker, “Expressing and Enforcing Flow-based Network Security Policies”, Univ. of Chicago, Chicago, IL, USA, Tech. Rep., Mar. 2008. [Online]. Available: https://www.cs.uic.edu/hinrichs/papers/hinrichs2008design.pdf
[39] C. Monsanto, N. Foster, R. Harrison, and D. Walker, “A Compiler and Run-Time System for Network Programming Languages”, SIGPLAN Notices, vol. 47, no. 1, pp. 217–230, Jan. 2012.
[40] B. He, L. Dong, T. Xu, S. Fei, H. Zhang, W. Wang, “Research on network programming language and policy conflict for SDN”, Concurrency Computation: Pract. Exper. Wiley, 2017.
[41] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing Software-Defined Networks”, in Proceedings of the tenth USENIX conference on Networked Systems Design and Implementation. Berkeley, CA, USA, 2013.
[42] A. Voellmy, H. Kim, and N. Feamster, “Procera: a language for highlevel reactive network control”, in Proceedings of the first workshop on Hot topics in Software Defined Networks, ACM, 2012.
[43] Z. Wan and P. Hudak. “Functional Reactive Programming from first principles”, in Proceedings of ACM Conference on Programming Language Design and Implementation, 2000.
[44] A. Voellmy, A Agarwal and P. Hudak, “Nettle: Functional reactive programming for OpenFlow Networks”, DTIC Document, Tech. Report 2010.

[45] C. J. Anderson, et al., "NetKAT: Semantic foundations for networks" , SIGPLAN Notices, vol. 49, no. 1, pp. 113–126, Jan. 2014.
[46] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic NetKAT," in Proceedings of Twenty Fifth Eur. Symp. On Program. Lang. Syst. (ESOP), Eindhoven, The Netherlands, pp. 282–309, 2016.
[47] R. Beckett, M. Greenberg and D. Walker, "Temporal NetKAT", in the Proceedings of Thirty Seventh Annual ACM SIGPLAN conference on Programming Lang. Design and Implementation, pp. 13-17 June, 2016.
[48] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network virtualization in multi-tenant datacenters" , in proceedings of eleventh USENIX Symposium on NSDI, Seattle, WA, April, 2014.
[49] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for Software Defined Networks", in Proceedings of the second workshop on Hot topics in SDNs, ACM, 2013.
[50] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: simplifying SDN programming using algorithmic policies", in Proceedings of the ACM SIGCOMM conference, 2013.
[51] J. Wang, S.Cheng and X. Fu, "SDN Programming for Heterogeneous Switches with Flow Table Pipelining", Scientific Programming, Hindawi, DOI: https://doi.org/10.1155/2018/2848232, 2018.
[52] R. Soule, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the Network with Merlin", in Twelfth ACM Workshop on Hot Topics in Networks, College Park, MD, November 2013.
[53] B. L. A. Batista and M. P. Fernandez, "PonderFlow: A policy specification language for OpenFlow networks", in Proceedings of ICN, CA, USA, 2014.
[54] T. Parr, "ANTLR: ANother Tool for Language Recognition" , Online Available: http://www.antlr.org
[55] H. kim, J. Reich, A. Gupta, M. Shahbaz , N. Feamster, R. Clark, "Kinetic: Verifiable dynamic network control", in Proceedings of USENIX Network Systems Design and Implementation, Oakland, CA, USA, pp. 59–72, 2015.
[56] N. Foster, M. J. FreedMan, A. Guha, et al., "Languages for software-defined networks," IEEE Communication Magazine, vol. 51, no. 2, pp. 128–134, Feb. 2013.
[57] C. Trois, M. Martinello, L. Bona, and M. Del Fabro, "From software defined network to network defined for software," in Proc. ACM Symposium on Appl. Comput., Spain, pp. 665–668, 2015.
[58] S. Layeghy, F. Pakzad, and M. Portmann, "SCOR: Constraint Programming-Based Northbound Interface for SDN", in Twenty Sixth International Telecommunication Networks and Applications Conference (ITNAC), pp. 83–88, Dec 2016.
[59] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a Standard CP Modelling Language", in Principles and Practice of Constraint Programming, Ed. Berlin, pp. 529–543, 2007.
[60] M. Mohammadi, A. Al-Fuqaha, Z. J. Yang, "A High-Level Rule-based Language for Software Defined Network Programming based on OpenFlow", in Proceedings of GENI Engineering Conference 22 (GEC 22), 2015.
[61] W. Kellerer, A. Basta, and A. Blenk, "Using a flexibility measure for network design space analysis of SDN and NFV," in the proceedings of IEEE International Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 423–428, April 2016.
[62] H. Farhad, H. Lee, and A. Nakao, "Data plane programmability in SDN," in the proceedings of IEEE Twenty Second International Conference on Network Protocols, pp. 583–588, Oct 2014.
[63] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and software-defined networking," Proceedings of the IEEE, vol. 103, no. 7, pp. 1102–1124, July 2015.
[64] B. Li, K. Tan, L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, E. Chen, "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware", ACM SIGCOMM 2016.
[65] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router", ACM Transactions on Computer Systems (TOCS), vol. 18, 2000.
[66] "BESS: Berkeley Extensible Software Switch," http://span.cs.berkeley.edu/bess.html,
[67] "VPP/What is VPP?" https://bit.ly/2mrxVGE,
[68] A. Panda, K. Jang, M. Walls, S. Ratnaswamy, S. Shenker, "NetBricks: Taking the V out of NFV", in the proceedings of USENIX OSDI, 2016.
[69] L. Molnar, G. Pongracz, G. Enyedi, Z. L. Kis, et. al., "Dataplane Specialization for High performance OpenFlow Software Switching", in ACM SIGCOMM, 2016.
[70] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN", ACM SIGCOMM Conference, vol. 43, 2013.
[71] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "DRMT: Disaggregated Programmable Switching," in ACM SIGCOMM Conference, 2017.
[72] R. Duncan and P. Jungck, "packetC Language for High Performance Packet Processing", in the proceedings of 11th IEEE International Conference on High Performance Computing and Communications, 2009.
[73] CloudShield Technologies. "packetC Programming Language Speci- fication. Rev. 1.128, October 10, 2008.
[74] 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, USA: ACM, pp. 127–132, 2013.
[75] X. Wang, Y. Tian, M. Zhao, M. Li, L. Mei, X. Zhang, "PNPL: Simplifying Programming for Protocol-Oblivious SDN Networks", Computer Networks, vol. 147, pp. 64-80, 24 Dec. 2018.
[76] P4, [online] Available: http://p4.org/
[77] P4 16 Language Specification v.1.2.1, [online] Available: https://p4.org/p4-spec/docs/ P4-16-v1.2.1.html
[78] B. O'Connor, Y. Tseng, M. Pudelko, C. Cascone, A. Endurthi, Y. Wang, A. Ghaffarkhah, D. Gopalpur, T. Everman, T. Madejski, J. Wanderer, and A. Vahdat, "Using P4 on Fixed-Pipeline and Programmable Stratum Switches," in P4 Workshop in Europe (EuroP4), 2010.
[79] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel, "High speed packet forwarding compiled from protocol independent data plane specifications," in *ACM SIGCOMM Conference*, 2016.

[80] Data Plane Development Kit (DPDK), [online] Available: https://www.dpdk.org/.
[81] Open Data Plane (ODP), [online]. Available: https://opendataplane.org/.
[82] M.Shahbaz,S.Choi,B.Pfaff,C.Kim,N.Feamster,N.McKeown,and J. Rexford, "PISCES: A Programmable, Protocol-Independent Software Switch," in ACM SIGCOMM Conference, 2016.
[83] OpenvSwitch, [online]. Available: https://www.openvswitch.org/.
[84] X. Wu, P. Li, T. Miskell, L. Wang, Y. Luo, and X. Jiang, "Ripple: An Efficient Runtime Reconfigurable P4 Data Plane for Multicore Systems," in International Conference on Networking and Network Applications, 2019.
[85] T. Osiński, H. Tarasiuk, P. Chaignon, M. Kossakowski, "P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4", IEEE IFIP Networking Conference (Networking), 22-26 June, 2020.
[86] Netcope P4, [online]. Available:https://www.netcope.com/Netcope/media/content/ NetcopeP4_2019_web.pdf
[87] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A Rapid Prototyping Framework for P4," in ACM Symposium on SDN Research (SOSR), 2017.
[88] A. Seibulescu and M. Baldi, "Leveraging P4 Flexibility to Expose Target-Specific Features," in the proceedings of P4 Workshop in Europe (EuroP4), 2020.
[89] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford and D. Walker, "SNAP: Stateful Network-Wide Abstractions for Packet Processing", ACM SIGCOMM, August 22-26, Brazil, 2016.
[90] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakr- ishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in Proceedings of the 2016 ACM SIGCOMM Conference, New York, NY, USA: ACM, pp. 15–28, 2016.
[91] M. Shahbaz and N. Feamster, "The case for an intermediate representation for programmable data planes," in Proceedings of the first ACM SIGCOMM Symposium on Software Defined Networking Research, NY, USA: ACM, pp. 3:1–3:6, 2015.
[92] G. Brebner and W. Jiang, "High-speed packet processing using reconfigurable computing", IEEE Micro, vol. 34, No. 1, pp. 8–18, Jan 2014.
[93] Network Programming Language, [online]. Available: https://nplang.org/
[94] NPL Speicifications ver 1.3, [online]. Available:https://nplang.org/npl/specifications
[95] J. Gao , E. Zhai ,H. H. Liu ,R. Miao ,Y. Zhou, B. Tian, C. Sun, D. Sai, M. Zhang, M. Yu, " Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs", ACM SIGCOMM 2020, August 10–14, Virtual Event, NY, USA, https://doi.org/10.1145/3387514.3405879.
[96] A. Voellmy, S. Chen, X. Wang and Y. R. Yang, "Magellan: Generating Multi-Table Datapath from Datapath Oblivious Algorithmic SDN Policies", ACM SIGCOMM, August 22-26, Brazil, 2016.
[97] H. Li, P. Zhang, G. Sun, C. Hu, D. Shan, T. Pan, and Q. Fu, "An intermediate representation for Network Programming Languages", in the Proceedings of 4th ACM Asia Pacific Workshop on Networking, Aug. 3-4, Korea, 2020.
[98] S. Geissler, S. Herrnleben, R. Bauer, S. Gebert, T. Zinner, M. Jarschel, "TableVisor 2.0: Towards full-featured, scalable and hardware-independent Multi Table Processing", in the Proceedings of KuVS-Fachgesprach Fog Computing, Germany, March, 2018.
[99] H. T. Dang et al. "Paxos Made Switch-y". In: ACM SIGCOMM Comput. Commun. Rev., vol. 46, no. 2, May 2016.
[100] S. luo, H. Yu, L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes", in the proceedings of OSR'17, April 3–4, Santa Clara, CA, USA, 2017.
[101] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, M. J. Freedman, "Be Fast, Cheap and in Control with SwitchKV", in the proceedings of USENIX NSDI 2016.
[102] G. Siracusano, R. Bifulco "In-network Neural Networks", in CoRR abs/1801.05731, 2018. arXiv: 1801.05731.
[103] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici and G. Bianchi, " FlowBlaze: Stateful Packet Processing in Hardware", in the proceedings of Sixteenth USENIX Symposium on Networked Systems Design and Implementation, February 26–28, 2019.
[104] P. Bosshart et al. "Forwarding Metamorphosis: Fast Programmable match-action Processing in Hardware for SDN". In: ACM SIGCOMM 2013.
[105] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," SIGCOMM Computer Comm. Review, Vol. 44, No. 2, pp. 44–51, April 2014.
[106] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, S. Shanker, "Elastic Scaling of Stateful Network Functions", in the proceeding of fifteenth USENIX Symposium on NSDI, 2018.
[107] K. F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tamanna, B. Walker, "Contra: A Programmable System for Performance-Aware Routing", in the Proceedings of Seventeenth USENIX Symposium on Networked Systems Design and Implementation, Feb. 25–27, USA, 2020.
[108] N. Bjorner, J. Padhye, A. Agrawal, A. Bhargava, P. A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnam-raju, A. Power, N. M. Raje, and P. Sharma, "Validating datacenters at scale", in ACM SIGCOMM, 2019.
[109] A. Horn, A. Kheradmand, and M. R. Prasad. "Delta-net: Real-time network verification using atoms", in the proceedings of USENIX Networked Systems Design and Implementation, 2017.
[110] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks", In USENIX NSDI, 2014.
[111] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates", IEEE/ACM Transactions on Networking, vol. 24, no. 2, pp. 887–900, 2016.
[112] H. Yang and S. S. Lam. "Scalable verification of networks with packet transformers using atomic predicates", IEEE/ACM Transactions on Networking, vol. 25, no. 5, pp. 2900– 2915, 2017.
[113] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification" In ACM SIGCOMM, 2017.
[114] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation" In ACM SIGCOMM, 2016.
[115] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression", In ACM SIGCOMM, 2018.

[116] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, “Tiramisu: Fast and general network verification”, In USENIX NSDI, 2020.
[117] S. Prabhu, K.Y. Chou, A. Kheradmand, P. Godfrey, and M. Caesar “Plankton: Scalable network configuration verification through model checking”, In USENIX NSDI, 2020.
[118] B. Butler. What is intent-based networking? https://www.networkworld.com/article/3202699/lan-wan/what-is-intent-based-networking.html. 2017.
[119] Y. Li, J. Gao, E. Zhai, M. Liu, K. Liu and H. H. Liu, “Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling”, in the Proceedings of the 19th USENIX Symposium on NSDI April 4–6, USA, 2022.
[120] J. Schulz-Zander, C. Mayer, B. Ciobotaru, S. Schmid, et al. “Unified Programmability of Virtualized Network Functions and Software-Defined Wireless Networks” , in IEEE Transactions on Network and Service Management, 2017.
[121] R. F. Moyano, D. Fernandez, N. Merayo, C. M. Lentisco and A. Cardenas, “NFV and SDN-Based Differentiated Traffic Treatment for Residential Networks”, IEEE Access, Feb 2020.
[122] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, “E2: A Framework for NFV Applications”, In Proceedings of the 25th Symposium on Operating Systems Principles, SOSP ’15, pages 121–136, New York, NY, USA, 2015. ACM.
[123] R. Gandhi, Y. C. Hu, and M. Z. Yoda, “A Highly Available Layer-7 Load Balancer”, in Proceedings of the Eleventh ACM European Conference on Computer Systems, pp. 21:1–21:16, New York, USA, 2016.
[124] Jeongseok Son, Yongqiang Xiong, Kun Tan, Paul Wang, Ze Gan, and Sue Moon. “Protego: Cloud-Scale Multitenant IPsec Gateway”, In the proceedings of USENIX Annual Technical Conference (USENIX ATC 17), pages 473–485, Santa Clara, CA, 2017.
[125] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, “HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane,” in IEEE International Conference on Computer Communications and Networks, 2017.
[126] C. Zhang, J. Bi, Y. Zhou, and J. Wu, “HyperVDP: High-Performance Virtualization of the Programmable Data Plane,” IEEE Journal on Selected Areas in Communications, vol. 37, 2019.
[127] K. Gao, T. Nojima, and Y. R. Yang, “TRIDENT: Towards a Unified SDN Programming Framework with Automatic Updates”, in the Prodeedings of ACM SIGCOMM Conference, Budapest, Hungary, 20-25 Aug. 2018.
[128] E. O. Zaballa, D. Franco, M. S. Berger, and M. Higuero, “A Perspective on P4-Based Data and Control Plane Modularity for Network Automation,” in the proceedings of P4 Workshop in Europe, 2020.
[129] M.Canini, A. Feldmann, D. Levin, F. Schaffert, S.Schmid, “Software-defined networks: incremental deployment with panopticon”, IEEE Computer vol. 47, no. 11, pp: 56–60, 2014.

Authors’ Profiles

Suhail Ahmad Mir is Assistant Professor in Department of Computer Science and Engineering, University of Kashmir. He received B. Tech. degree in Computer Science & Engineering from the University of Kashmir, M. Tech. and Ph. D degree from National Institute of Technology, Srinagar, Jammu & Kashmir. His research interests include Network Security, MPLS, Network Programming and Software Defined Networks.

Ajaz Hussain Mir is Professor and Head of Electronics and Communication Department, National Institute of Technology, Srinagar, Jammu & Kashmir. He received the Ph. D and M. Tech degree from IIT-Delhi. His current research interests include Network Security, Mobile Networks, IOT, Next Generation Networks, Software Defined Networks and Image Processing.

How to cite this paper: Suhail Ahmad, Ajaz Hussain Mir, "Programming SDNs: A Compass for SDN Programmer", International Journal of Wireless and Microwave Technologies(IJWMT), Vol.14, No.1, pp. 52-72, 2024. DOI:10.5815/ijwmt.2024.01.05