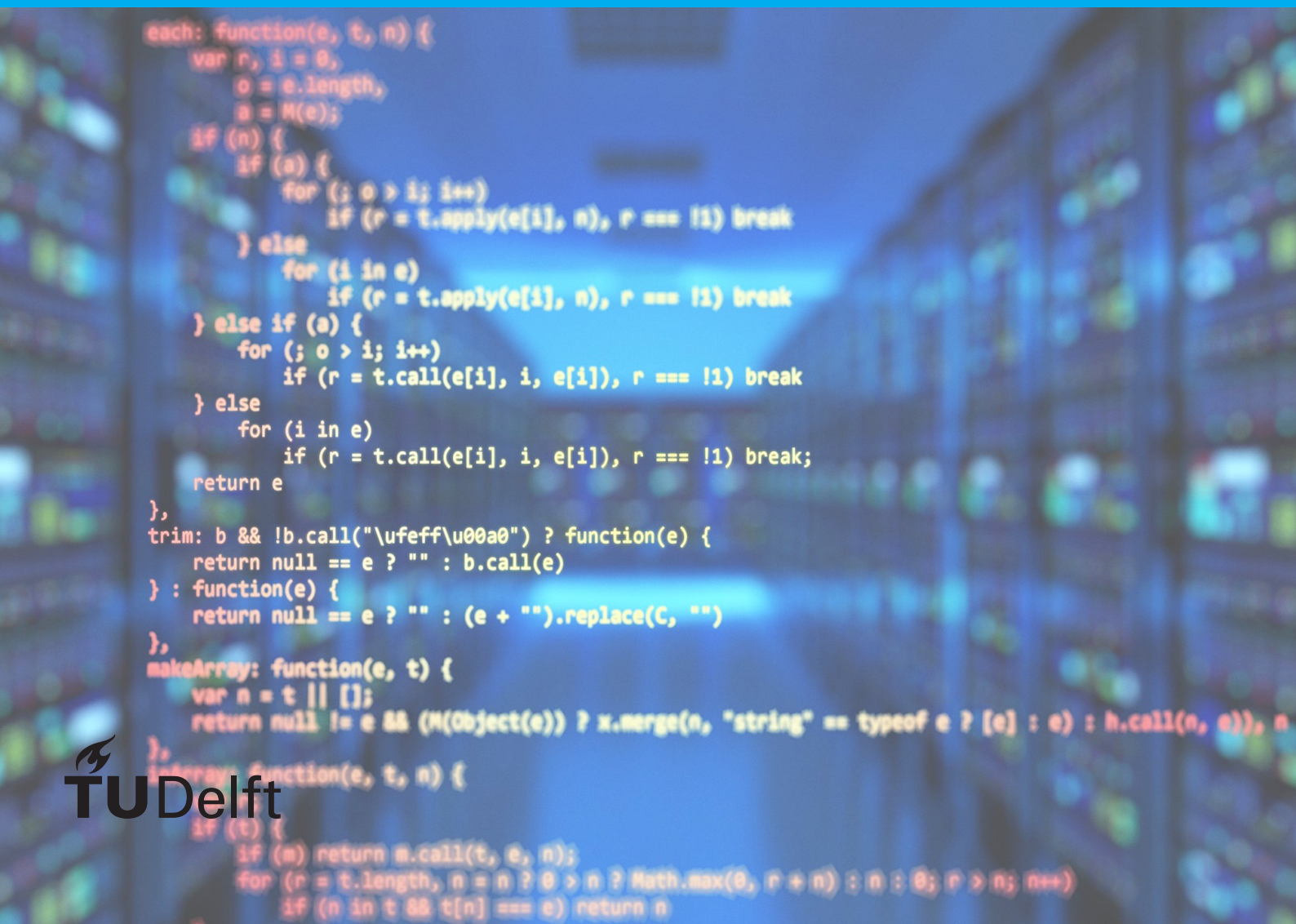
Intent-Based Networking with Programmable Data Planes

Mohammad Riftadi



TU Delft

Intent-Based Networking with Programmable Data Planes

by

Mohammad Riftadi

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 8, 2019 at 10:00 AM.

Student number:	4743792	
Project duration:	September 19, 2018 – August 8, 2019	
Thesis committee:	Dr. ir. F. A. Kuipers,	TU Delft, supervisor
	Dr. M. Nasri,	TU Delft
	Dr. C. Hauff,	TU Delft

TU Delft

Preface

This thesis report document lying before you serves as the final artifact of my 2-year journey as a master student in Computer Science at the Delft University of Technology. I have spent a major part of my second year working on this thesis, during which I have learned and grown a lot. This work would not be possible without the support of the people that I would like to mention here.

First, I would like to express my deepest gratitude and appreciation to my supervisor, *Dr. Fernando Kuipers*, for allowing me to realize my dream of working on a thesis related to artificial intelligence in my favorite domain of computer networking. Without his guidance and patience, I would surely not be able to finish this thesis work. He made me felt very welcome in Delft, even already guided me before I got accepted into the university. On a more special note, he also introduced me to the wonderful world of research and also gave me opportunities to disseminate the knowledge that I gained during the process. If I am going to be a successful researcher one day, I will surely attribute my success on his name.

I would also like to thank my collaborator, *Jorik Oostenbrink*, who has helped me a lot during the work on the GP4P4 chapter of this thesis. His amazing insight has widened my perspective while looking into a problem. Special thank also goes out to *Belma Turkovic* for the discussion and the brainstorming of ideas during the initial phase of this work.

Further, I would like to thank my friends for their companionship and support throughout my endeavor in Delft: *Gilang*, *Haris*, *Enreina*, *Francisco*, and many others that would be too long to be named here. I am also deeply grateful to my wife, *Syifa*, for her support and care she had provided me during this work, and my daughter, *Sheryl*, for providing a cheerful atmosphere at home. I would also like to express my sincerest gratitude to my parents for their never-ending love, affection, and good wishes to me. Last but not least, I would also like to thank *the Ministry of Communication and Information Technology of Indonesia* for providing me the funding that enabled me to pursue my study at the Delft University of Technology.

Mohammad Riftadi
Delft, 8 August 2019

Contents

1 Introduction

1.1. Background

The Internet has become an essential part of modern life. Popular user applications like social media and video sharing platforms promote the usage of the Internet by billions of people every day and generating massive amounts of user traffic. From another perspective, this phenomena can be viewed by the network operators as a new challenge: *how to manage a network that serves millions or even billions of users while minimizing the operating cost?*

Traditionally, network devices were rigid, fixed-function devices developed by network device manufacturers. This approach was necessary as the software and hardware components needed to be developed hand-in-hand to be able to produce highly-performing devices that can cater to the high throughput required by the operators. With this approach, the functionality of each device is essentially "locked" and the operators are only left with the option to configure and tune the parameters exposed by the manufacturers. It is also for this reason that network devices are traditionally referred to by the name of their functionality, e.g. switch, router, firewall, load-balancer, etc.

The situation changed as of recent advances on-chip designs have enabled programmable hardware to achieve terabit speeds [7, 39]. This development enables network vendors to build high-performing network devices that are also user-programmable. With this tool in hand, we are now ready to challenge the status quo of the *bottom-up* approach in which the hardware dictates what a network can do, into a more promising *top-down* approach, in which the network controller governs the behavior of the forwarding hardware.

Further, the introduction of P4 data-plane abstraction language [8] made the development of customized network functions possible for everyone. It hides the complex low-level mechanisms and the native configuration languages of various programmable network devices. Network programmers only need to learn about the abstraction model of the P4 language and let the P4 compiler take over and translate the program into a target device-specific configuration.

Yet, P4 does not come without drawbacks. P4 is written in the form of a procedural language, requiring the network operators to learn how to program in it before they can reap the benefit of custom network functionality. In contrast, most networks today are run by individuals that have deep knowledge of networking protocols and strong skills in configuring network devices while possessing little to no exposure to programming. Furthermore, the operators also have to face the problem of managing various software artifacts along with other complexities associated with software management. It is impractical to expect a network operation department to possess a skill set similar to that of a software engineering department. These two reasons effectively create a barrier in the adoption of programmable switch technology.

The "problem" of removing the barrier of having to learn a procedural programming language is a recurring theme within this work. We develop a solution to this problem based on the philosophy of Intent-Based Networking (IBN). Instead of having to "tell" the network de-

vices what to do, the operators can instead just state their intended behavior to the network controller, after which the network carries out the implementation by itself.

The concept of IBN is analogous to the self-driving car concept. Instead of manually driving the car, the passenger can just state the destination and, optionally, some other constraints, e.g. taking the most scenic route, minimizing the fuel utilization, etc. For the passenger, this alleviates the burden of learning how to drive a car. Translated to the context of network management, we want the network operator to be able to "drive" the network without having to master the mechanics of how the network devices operate.

1.2. Problem Definition

While P4 provides more flexibility by enabling operators to implement their required network functions in the form of a procedural program, it also has some drawbacks associated with it as well. The disadvantage mainly comes from the fact that now network operators have to learn a new programming language and write code before they can run the network. As the skill set needed to operate traditional network devices and new programmable devices are different, this might pose some difficulties for most network operators.

We propose a solution to this problem by using the Intent-Based Networking (IBN) approach. Instead of requiring operators to code the network functions in the form of a procedural language, we give the network operators a tool to control their network based on their intentions, i.e. network goals. The network should then find a way to satisfy the expressed network goals by searching for a network condition that fulfills the network goals and then implementing this certain condition automatically.

1.3. Research Question

Before the IBN concept can be realized, we have to find the answers to the following research questions:

- **RQ1. In what language can we express our network intent?** One of the key requirement of this intent language is that it should be easier for the network operators to learn compared to a procedural programming language like P4.
- **RQ2. How to design and implement a system that can generate a P4 program based on user intent?** The system should be able to find a correct P4 program from the supplied intents and the current network conditions.
- **RQ3. How is the efficacy of such a system?** To be able to replace the current approach on data-plane programming, the system should be able to generate P4 program with reasonable performance, i.e. in a short time. Therefore, it is mandatory to evaluate the performance of our solution.

1.4. Thesis Outline

Apart from this introductory chapter, this thesis report is written with the following structure:

- **Chapter 2: Theoretical Background** aims to give the readers the necessary background knowledge required to understand the concepts and techniques described in this article. We start by describing the concept of data-plane programmability and also P4 the most popular language in the domain of network data-plane programming. We continue with delineating the concept of Intent-Based Networking before finally closing the chapter with a description of Genetic Programming, a machine learning technique that can be utilized to build imperative/procedural programs using a concept borrowed from evolutionary biology.
- **Chapter 3: P4I/O: Intent-Based Networking with P4** describes a framework that is capable to generate P4 programs based on parameterized templates to satisfy certain network intents. This chapter is based and adapted from our publication with the same title [56].

- **Chapter 4, GP4P4: Enabling Self-Programming Networks** defines a framework that can generate a certain part of a P4 program based on some network behavioral rules using a Genetic Programming technique. This chapter is based and adapted from our submitted paper with the same title.
- **Chapter 5, Conclusion** summarizes the conclusions that could be made from the design and evaluation of the frameworks that have been described in Chapter 3 and Chapter 4. We close the chapter with some directions for future research.

1.5. Publication

Part of the work described within this thesis report has led into one publication and one paper that is currently under review. The publication and paper details are as follows:

1. Mohammad Riftadi and Fernando A. Kuipers. 2019. P4I/O: Intent-Based Networking with P4. In Proceeding of 2019 2nd International Workshop on Emerging Trends in Softwarized Networks (ETSN 2019 at NetSoft). Paris, France.
2. Mohammad Riftadi, Jorik Oostenbrink, and Fernando A. Kuipers. 2019. GP4P4: Enabling Self-Programming Networks. Submitted to ANCS 2019: The 15th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. Cambridge, UK.

2

Theoretical Background

This chapter provides the necessary background on various concepts and techniques that are used throughout this manuscript. We start by introducing the concept of data-plane programmability and the P4 language, continuing with intent-based networking and closing it with a description of genetic programming.

2.1. Data-Plane Programmability

This section describes network data-plane programmability. We begin by describing the definition of network data-plane, then describing the programmable switches that are used to implement the notion of data-plane programmability, and finally closing the section by delineating P4, the most popular language used in data-plane programming.

2.1.1. Network Data-Plane

Network *data-plane* is defined as the part of the network that does the actual packet forwarding. This is in contrast with the *control-plane*, which gives certain instructions to the forwarding elements for it to be able to forward a packet to the correct destination while satisfying any constraints at the same time. Traditionally, the data-plane and control-plane of a network device reside in the same hardware, although the control-plane is usually implemented as a piece of software running on top of general-processing CPU. The network devices then "learn" the information of the network, e.g. topology, link utilization, by communicating with each other using a set of networking protocols. The networking protocols used by the traditional network devices can be categorized into various layers in the OSI framework. Table 2.1 provides examples of the networking protocols used by traditional network devices to gather information over the network.

The notion of network control- and data-planes is analogous to how the brain and muscles work in nature. The "muscles" or the forwarding devices perform the packet forwarding activities, while the "brain" or the software controller instructs the forwarding elements from a distance.

Software-Defined Networking changed this approach, by physically separating the control-plane software and the data-plane forwarding elements in the network [40]. The centralized

Table 2.1: Network Protocol Examples

OSI Layer	Protocols
Physical layer	Bidirectional Forwarding Detection (BFD) [34]
Data-link layer	Spanning Tree Protocol (STP) [1], Link Layer Discovery Protocol (LLDP) [2]
Network layer	Routing protocols: OSPF [47], BGP [54], IS-IS [50]; Multicast protocols: PIM-DM [4], IGMP [12]

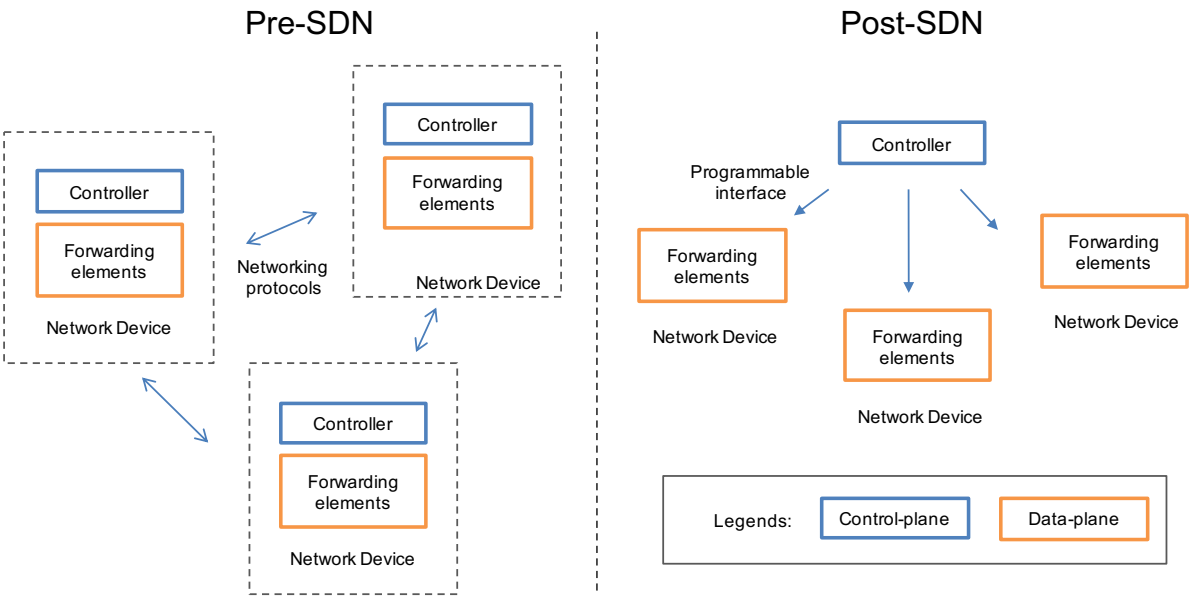


Figure 2.1: Pre- and post- SDN data- and control- planes separation.

control-plane software, or usually termed as the network controller, has full visibility over the network and is able to communicate with the forwarding elements over a vendor-agnostic interface, such as OpenFlow [44]. The SDN philosophy was born with the main mission of giving network developers, operators or programmers more flexibility to manage and control the network. Figure 2.1 illustrates the pre- and post-SDN definitions of the network control- and data-planes.

2.1.2. Programmable Switches

From the network data-plane perspective, a trade-off is often needed between flexibility and speed. The network data-planes are traditionally realized on top of specialized hardware, such as ASICs. They give high performance, up to terabit speed, at the cost of sacrificing flexibility – they are specifically designed for fixed functionality. On the other hand, there are also network data-planes built on top of general-purpose CPU, e.g. OpenvSwitch [52], Click [36], RouteBricks [19]. These systems are characterized by their high flexibility, as they can be programmed to perform any kind of network functions, but at the cost of slower forwarding speed.

On the other hand, the proliferation of bandwidth-hungry applications (e.g. big data processing applications, multimedia content platforms) in data-center networks creates a torrent of network traffic. New network virtualization technologies, like NVGRE, VXLAN, and STT, are invented to help network operators manage these networks with massive scale. These technologies keep evolving at a fast pace, while the network device manufacturers still operate with a long development cycle resulting in a gap in possible improvements and their realization.

We refer to this as a “bottom-up” approach, as this approach lets the capability of the forwarding hardware (e.g. ASICs) govern what a network can do. This traditional approach of waiting for the network device manufacturers to build devices that are able to run the latest technologies needs a major revision.

Bosshart et al. [8] demonstrated another example of the disadvantage of fixed-functionality hardware that is observable in the evolution of the OpenFlow protocol. In the beginning, OpenFlow [44] version 1.1 only supported matching a dozen of protocol header fields (e.g. MAC addresses, IP addresses, TCP/UDP port numbers, etc.). But over the years, as the user requirements grew more complex, the OpenFlow protocol specification also became more complex. For example, OpenFlow version 1.4 already needs to support recognizing 41 different protocol headers. This means that the data-plane hardware should also be able to

Table 2.2: OpenFlow protocol evolution.

Version	Date	Fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

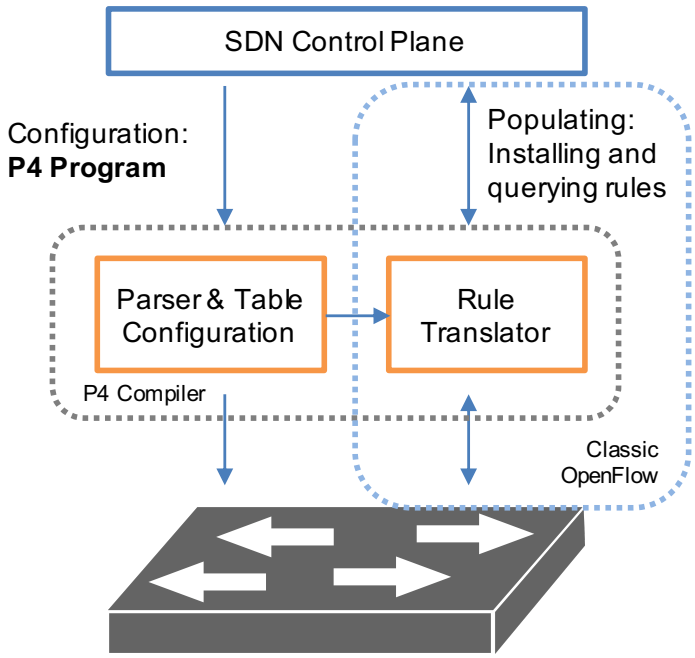


Figure 2.2: P4 overview.

recognize and parse these protocol headers each time the specification is revised, posing challenges for fixed-functionality hardware to keep up. Table 2.2 depicts the evolution of the OpenFlow protocol in more detail.

Only recently, the trade-off of flexibility and performance was proven wrong. Recent advances on-chip designs have enabled several programmable forwarding hardware to achieve terabit speeds [7, 39]. This development enables network vendors to build devices that can be programmed by the users. Finally, it was possible to change the aforementioned “bottom-up” approach with a “top-down” approach, in which the network controller is able to tell the forwarding hardware what it should do.

Regardless, programming custom network functions on these chips is by no means an easy task. Each of these programmable devices has a different set of low-level functionalities, not to mention the different languages used to implement them [8]. P4 [8] proposes to solve this data-plane programming challenge by providing a generic abstraction language that can be used to write forwarding behavior for theoretically any kind of forwarding hardware.

2.1.3. P4 Language

P4, which stands for Programming Protocol-independent Packet Processors, is a data-plane programming abstraction language, proposed by Bosshart et al. [8]. It was designed with the main goal of enabling network end-users to be able to write their network functions. The language defines various components of the data-plane which can be used to govern the way a network device processes incoming packets.

While OpenFlow gives a tool for network operators to customize forwarding behavior based on matching a set of protocol headers and determining what action(s) to take for various kind

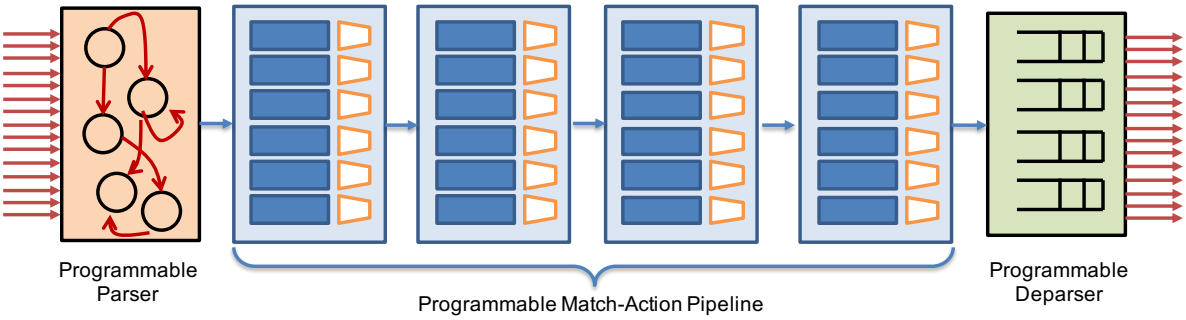


Figure 2.3: P4 data-plane abstraction model. Reproduced from the P4 language tutorial [51].

of header combinations, P4 enables network operators to govern all data-plane functionalities that a programmable device can support, including defining (new) network headers and the corresponding (customized) actions. In other words, P4 enables a network controller to tell the switch *how to operate* [8]. It enables countless new possibilities and gives the agile character of the software development cycle in the context of network data-plane development. Figure 2.2 illustrates the high-level working of P4 and its differences compared to OpenFlow.

To provide a high degree of flexibility, P4 was designed with 3 main goals in mind:

- **Reconfigurability.** The network operator should be able to adapt the network functions as needed. This implies that these changes should also be able to be implemented on-the-fly.
- **Protocol independence.** We should not limit the switch to predefined supported packet headers, but instead, we have to be able to define any network headers that we need. Further, we can build a customized match-action table based on the header values.
- **Target independence**. The P4 programmer does not need to know the details of the *target switch*. Instead, the compiler should translate the written P4 program into a form of software that is understood by the target hardware.

To write a program in the P4 language, the network programmers should understand the concept of P4's abstract forwarding model in advance. Derived from OpenFlow, the abstract forwarding model generalizes the packet forwarding process from various network devices (e.g., L2 switches, L3 routers, firewall, load-balancers) and by different technologies (e.g., fixed-function ASICs, NPUs, FPGAs, software switch). The model, as shown in Figure 2.3, allows to build a target-independent program that can be mapped into a variety of network devices.

The abstraction model comprises 3 main components:

- **Programmable Parser.** This component abstracts the declaration of the packet headers that should be recognized by the switch along with their order in the packet.
- **Programmable Match-Action Pipeline.** This component defines the packet processing algorithm and also the match-action tables used to decide which actions to take based on the value available in the packet headers.
- **Programmable Deparser.** This component is used to define how a processed packet should look when transmitted back on the wire.

Further, the P4 language has several important elements:

- **Parsers.** This element provides the state machine needed to do bit field value extraction from the packet headers.
- **Controls.** This element contains the match-action table definitions that are used to determine which action to take. We can use branching control flow statements, i.e. `IF-THEN-ELSE` statements, to implement more complex functionality.

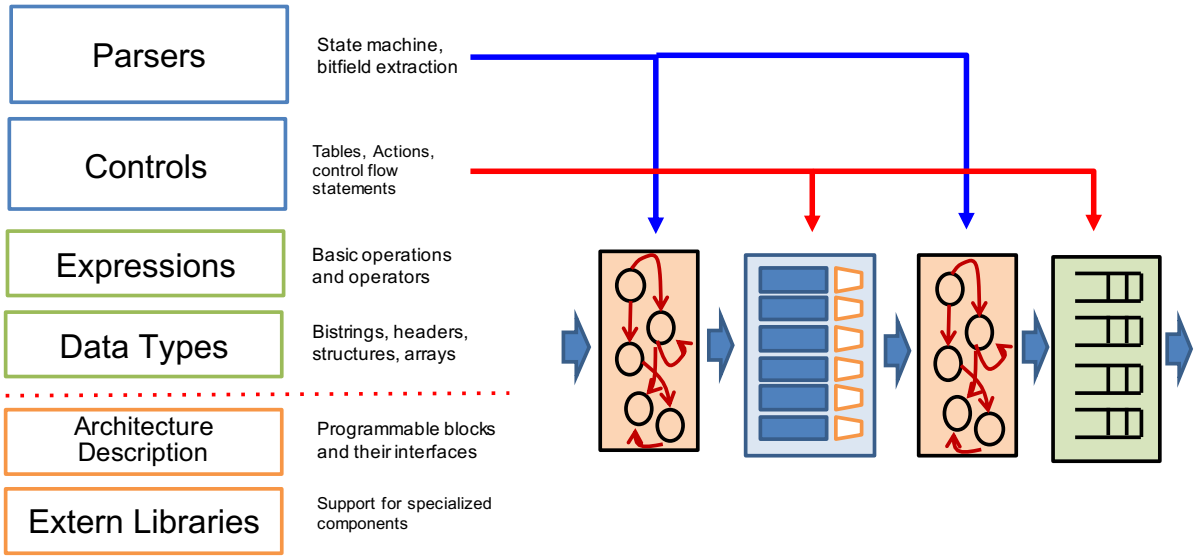


Figure 2.4: P4 language elements. Reproduced from the P4 language tutorial [51].

- **Expressions.** The expression element defines the operators and operands available in the P4 language.
- **Data Types.** P4 comes with several basic data types, such as strings, Boolean, integer. We can also define new data structures by combining the basic data types, just like in other imperative languages.
- **Architecture Description.** It specifies a particular configuration of programmable parsers, control blocks, and perhaps vendor-specific blocks.
- **External Libraries.** P4 also supports a library of "externs". The external functions provide some functionality that is *not* implemented in P4 but can be invoked from the P4 programs. An example of an external function is the hashing function, which is usually performed on specific hardware.

Figure 2.4 illustrates the P4 language components and their corresponding locations, if any, in the abstraction model. The complete specifications of the P4 language can be found in the $P4_{16}$ language specification document [61].

2.2. Intent-Based Networking

The basic idea of Intent-Based Networking (IBN) is to manage the network through describing the network services as *what goals we would like to achieve* instead of *how the services should be built* [5]. The network controller can then work to implement the intended goal by computing what kind of resource and configuration that is needed to provide such a service. The controller should go as far as guaranteeing that the intended service is being enforced correctly in the network.

While the concept of IBN is relatively still new, the notion of network management by describing policies is not. The concept of Policy-Based Network Management (PBNM) far predates the concept of IBN. Even though the main ideas between these two approaches are similar, the IBN is fundamentally different in which intents are independent of the network topology, technology, and vendor-specific features [28]. This independence allows us to move intents between one network to another conceptually without requiring any forms of modification or adaptation.

2.2.1. Intent Definition

There is no standard definition of intent in the context of IBN, but it is commonly understood as business or system-level policies used to govern network services [28]. A business-level

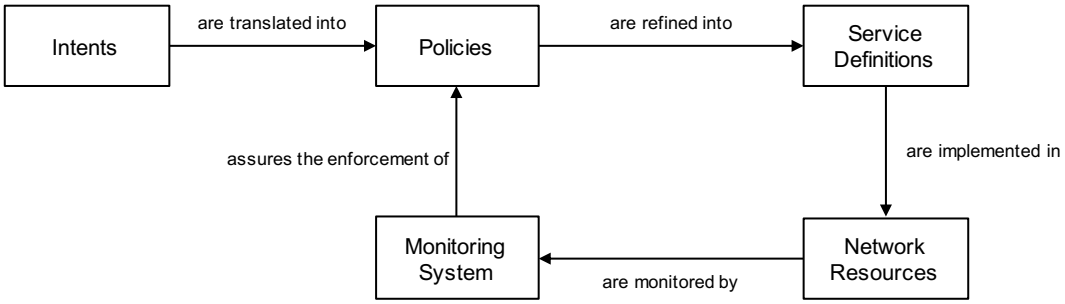


Figure 2.5: Intent Based Networking Architecture.

policy is usually described in a natural language, employing business domain vocabularies. For example, a network intent in a business level could be stated like *every gold level users should always get an uninterrupted session of high-resolution video streaming at all time*. In this example, the definition of some entities such as gold level users and a high-resolution video streaming is still not explicitly defined. The technical definition of what the entities are could be drilled down further in the controller by means of defining network policies, i.e. the formal definitions of what the entities are. The network policies can vary from network to network, but the service intent does not need to be modified.

2.2.2. Intent-Based Networking Architecture

Many works have been done in attempts to define the architecture of IBN. One of the earliest attempts was [66], which proposed a general policy-based administration framework adapted from the IETF/Distributed Management Task Force (DMTF) policy framework. More recently, Cohen et al. [15] introduced an intent-based network virtualization reference architecture. They realized it using an intent-based North-Bound Interface (NBI) and a network overlay technique called DOVE. However, they did not specifically describe the detail regarding their intent-based NBI or the design they used to implement the system.

Some works also brought up the idea of end-to-end intent-based networking deployment, although many of them only worked on developing a simple deployment to evaluate the feasibility of IBN. Cerroni et al. [13] developed simple intent-based network management for provisioning end-to-end network services over several network domains, but did not attempt to standardize any part of IBN management. Han et al. [28] took a different approach of a layered architecture for managing an intent-based virtualized network. They argued that each layer should give an abstraction for the next layer. Their architecture consisted of five layers: protocol adaptation, abstraction, virtualization, virtual abstraction, and intent layer.

There are also some efforts in defining the architecture of IBN in the industry. Cisco Systems [14] defined an intent-based network as a network that should fulfill 3 functions: translation of intent, deployment of the computed intent to the network, and assurance of the deployed intent using continuous monitoring. Veriflow Systems [25] shared a similar view and defined an intent-based network control loop consisting of 5 functions: intent description, translation, implementation, awareness, and verification.

From the aforementioned works, we can infer a general architecture of an intent-based network. The general intent-based management architecture is illustrated in Figure 2.5. The service definition starts with a form of intent description. The intent description is further translated into network policies by formalizing the intent into some form of explicit network policies. These policies are then refined, i.e. checked for consistency and usability, into some form of service definitions which are ready to be implemented in the network. After the service is deployed, a specialized monitoring system will monitor the network resource to assure that the policies are enforced correctly by the network.

The following sections are organized based on the entities in this general architecture.

2.2.3. Intent Extraction

One main challenge in the domain of intent extraction is how to translate intent from a natural language to formal policies. Jacobs et al. introduced Nile [31], a high-level intent definition language that is used as an abstraction layer between a natural language and the associated network policy. Nile is human-readable, as it closely resembles the English language, while still providing the conciseness that is required for further computation.

In Nile, the entities from the user's utterance are extracted using DialogFlow [30], a natural language processing framework developed by Google, which in turn utilizes machine learning techniques to perform the entities extraction. The intent is then presented, in the format of Nile language, to the user, who will give feedback regarding the translated intent, e.g. confirm or reject the translation of the intent. The feedback will be used to re-train the learning model that they used, improving the entities extraction accuracy of the model.

Attempts in standardizing network intent in the form of a language have also been made. One notable example is the Nemo project [49]. The project aims to develop a standard NBI API which allows applications to use intent-based policy to create virtual networks. Further extension on the Nemo language was also proposed in [64], which appended the functionality of conditional operators into the language.

2.2.4. Policy Refinement

Policy refinement is defined as the problem of translating a high-level abstract notation to the language, e.g. configuration, used in various policy enforcement points [17]. There are a lot of works that have been done in the domain of network policy, with a special emphasis on the problem of policy refinement. The problems tackled by these works also vary from the problem of policies reconciliation [3, 53], selecting a set of NFVs required to implement a service chain [59], up to where to implement encryption in the network [60].

Policies Reconciliation

The problem of policies reconciliation can be defined as how to combine several network policies with differing requirements, to the extent that the policies might be conflicting with each other. This problem is possible when several users of the network specify their intent. For example, the network owner has the policy of disallowing all traffic on TCP port 23, while on the other hand, a network user needs to use TCP port 23 for her application to work. PGA [53] solved this problem by using a graph abstraction to reconcile high-level policies. However, PGA only dealt with access control list policy.

Janus [3] extended the graph-based abstraction introduced in PGA by adding support for QoS constraints and dynamic intent-based policies. The dynamic policies can be categorized as two main types: (1) temporal policies which include the notion of time into the policy, e.g. limit bandwidth at working hours, and (2) stateful policies that depend on the state of the network and/or application, e.g. direct traffic from a specific host to IDS when there are too many TCP SYN packets sourced from the host. In this work, the authors also developed some heuristic to maximize the number of policies being enforced by the mean of policy negotiation, i.e. degrading service for applications with loose QoS requirements.

Service Chaining

Scheid et al., in their work INSpIRE [59], offered another perspective on how intent-based networking can be utilized. They proposed a solution to refine intents in the form of a controlled natural language to do Virtual Network Functions (VNFs) selection based on a Non-Functional Requirements (NFRs) in the intent. NFRs are informal specifications of a service, which are usually based on empirical observation from stakeholders. An example of an intent which contains NFR is *financial traffic from the marketing division should be secured and confidential*. The notions of secure and confidential are not explicit, leaving room for interpretation.

INSpIRE solved the NFR definition problems using the technique of Softgoal Interdependency Graph (SIG) and machine learning algorithm to do clustering, e.g. K-mean clustering. It produces the output of the set of VNFs that should be employed to fulfill the intended specification. The order of the VNFs involved in the service chaining itself was outside of their scope of work.

Determining Encryption Location

Intent-based networking has also been adopted in the domain of network security. Szyrkowiec et al. [60] used policies based management to decide where best to implement encryption. They argued that there are 3 layers to implement encryption: physical layer (using hardware-based optical encryptor), link-level layer (using MACsec), and logical layer (using IPsec). As each of these layers has different performance characteristics, i.e. latency, throughput, flexibility, the policy compiler should decide where best to implement the encryption, based on the performance requirement policies predefined by the users.

2.2.5. Policy Assurance

Policy assurance is the process of guaranteeing that the deployed policies are indeed running correctly. The policy guarantee mechanism is implemented by monitoring various metrics from the deployed service and taking appropriate action whenever the policies are violated or predicted to be violated. The domain of policy assurance still could benefit from more research as not many works have been done in this domain. The question of how to best assure the deployed policies are enforced correctly in the network remains an open question.

Some of the works that have been done in this domain are too specific, leaving us with no general framework on how to tackle this problem. Examples of such works are [43] and [58]. Marsico et al. [43] demonstrated that more intents can be deployed in the network if negotiation of intent is possible. They did it by offering a degraded version of the intended service or by re-arranging the policy enforcement location of the existing deployed services. Sanvito et al. [58] extended ONOS[1] by implementing an off-platform application, i.e. not a part of the ONOS controller itself, that does the function of optimizing and re-routing for the deployed intent-based services. The resulting optimized network configuration is then communicated back to the controller via an API.

2.3. Genetic Programming

Gulwani et al. [26] states that Genetic Programming (GP) [38] can be regarded as an extension of Genetic Algorithms (GA) [29], specifically for the computer program synthesis domain, with the aim to generate computer programs. GP searches for a program by “evolving” an initial population of random programs and “breeds” them into new generations of programs via the means of genetic operations inspired from Darwin’s theory of natural selection. The genetic operations consist of the crossover, mutation, duplication and deletion operations [37].

The original GP proposal [38] evolves structures containing computer programs built on top of a set of primitives, i.e. functions, and terminal values. This set of primitives and terminal values together form the hypothesis (search) space of possible programs. The GP algorithm first creates a population of random programs with a certain size, after which it does the crossover and mutation genetic operators in order to produce new program variants. The crossover operation swaps a part of a program with a part from another program with the intention to combine “useful” subprograms, while the mutation operation introduces random changes in a program. It then evaluates how well a program complies to a certain specification by using a certain fitness evaluation function. It then expresses the fitness of the evaluated program in the form of a numerical value. This value is then used to asses the suitability of the program. The program with the highest fitness value is then promoted as the solution.

The GP algorithm works on top of a set of terminals and primitives that are specific to the application domain. This set defines the complete search space of possible programs. To be able to produce a program that satisfies the specification, there are 2 requirements for the search space:

1. a program that satisfies the specification should exist in the search space, and
2. each function (primitive) should be able to accept a value returned by another function or a terminal value as its argument.

[1] https://onosproject.org/

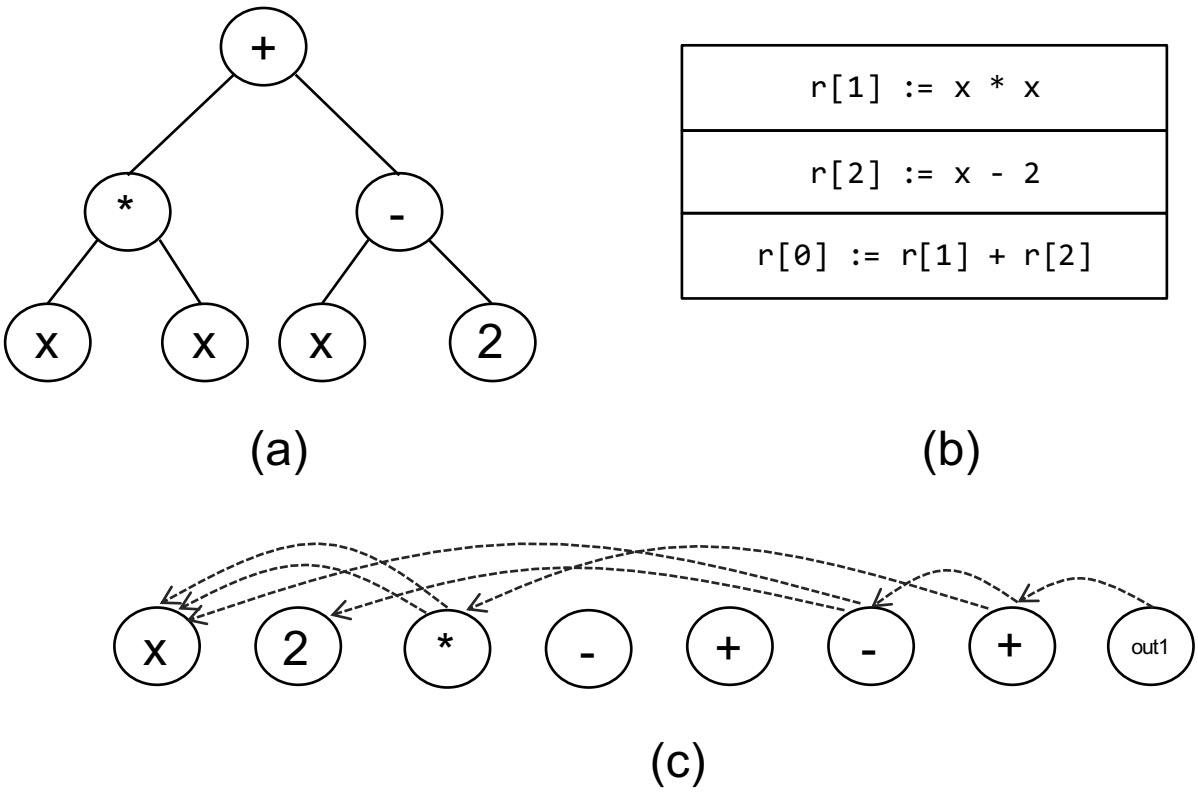


Figure 2.6: GP program examples in: (a) *tree-based* representation, (b) linear representation, (c) *graph-based* representation. All representations shows the program that computes the value of $f(x) = x^2 + x - 2$, with x as input and $f(x)$ as output.

2.3.1. Genotype Representation

There are three gene representations in Genetic Programming:

- **Tree-based representation.** It represents the genes in the form of a tree data structure, in which the functions reside in the internal nodes, while the terminal values reside in the leaves of the tree. The outputted value is always the evaluated value at the root of the tree. The original GP proposal [38] uses the tree-based representation.
- **Linear representation.** It represents the genes linearly as an imperative program [11]. Each line of the program consists of one function and its input values, which can be obtained from either another function or terminal value. Each program line modifies the content of an array of registers. The outputted value is conventionally stored in register number 0.
- **Graph-based representation.** It represents the genes in the form of a graph. The terminal values and functions are represented in the form of nodes, which then contain links to other functions or terminal values. The output is obtained by back-propagating the chain of nodes and links from a set of output nodes. One example of a GP variant that uses a graph-based representation is Cartesian Genetic Programming [45].

Figure 2.6 gives an illustration on the three GP representations for the program that computes the value of $f(x) = x^2 + x - 2$.

2.3.2. Workflow

To design a GP workflow, we need to define 4 main steps:

1. define a set of functions/primitives and the terminal values,

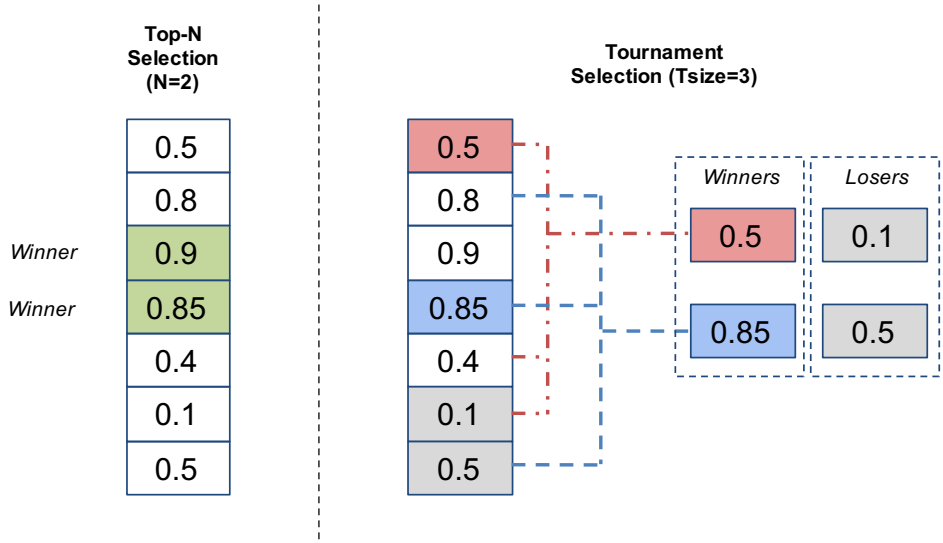


Figure 2.7: Selection methods: (left) top-N values selection (N=2), (right) tournament selection. Each box represents a fitness value.

2. define a fitness measurement function to assess the suitability of a program, i.e. how "close" the program is to our intent,

3. define the values of search parameters that guide the evolutionary process, such as the population size, the number of available primitives, the chance to do reproduction, crossover, mutation, deletion, etc., and

4. define the termination condition to end the evolutionary process and to return the best program.

The GP algorithm first needs to create a random population of initial programs, in which each individual is generated randomly from the set of terminal values and functions. Each program also needs to be syntactically correct in order to participate in the evolutionary process. It then measures the fitness of each program in the population using a fitness value evaluation function. The method of how to compute fitness value measurement is domain-specific and problem-dependent. Insight and creativity are often needed to build a good fitness value evaluation function. Some examples of fitness measures are counting the number of correct input-output value pairs, measuring the deviation of the program output and the desired output, measuring the properties generated after program execution (e.g. game score, energy, time, etc.) or the combination of these measurements. We can expect that the initial population contains programs with poor fitness values. However, some programs will have better fitness values than others. The discrepancy in these fitness values is our guidance to select which program to reproduce using the crossover and mutation operations.

2.3.3. Selection Mechanism

The GP algorithm then selects two individuals based on certain criteria from the population. The selection process is usually performed by comparing the fitness values of the programs. The most straightforward method is to just select two individuals with the highest and second-highest fitness values. However, the selection of best programs only might make the GP process stuck in a local minimum as it is likely that the gene pools will be limited to a few select individuals. Therefore, there also exist some alternatives for the program selection mechanism, like the tournament selection. The tournament selection does not necessarily select the programs with the highest fitness values, but instead, it introduces randomness into the process by comparing only a subset of programs. Each subset of programs will take part in a tournament. Each tournament returns one winner and one loser, which are the

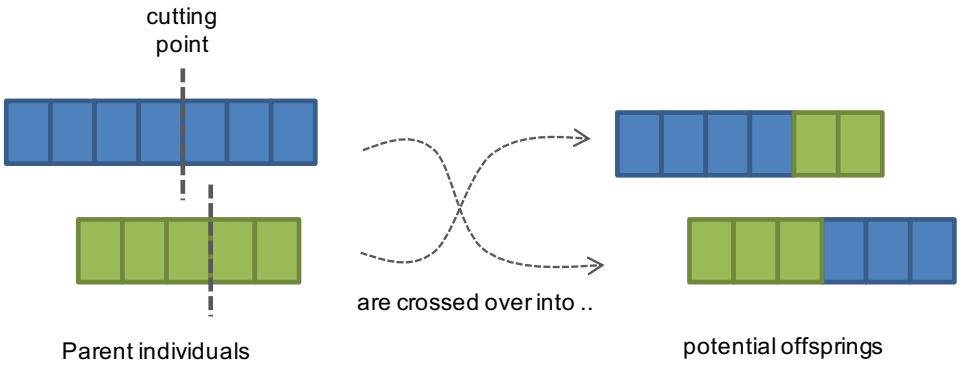


Figure 2.8: Crossover genetic operation example.

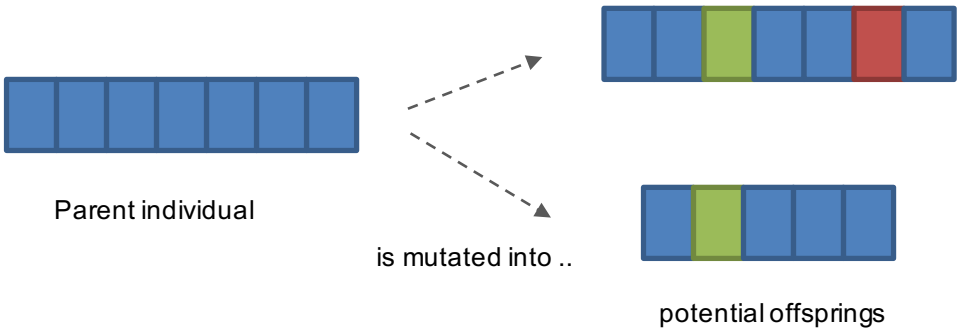


Figure 2.9: Mutation genetic operation example.

programs with the highest and lowest fitness values, respectively. Figure 2.7 illustrates the top-N and tournament selection mechanisms.

2.3.4. Crossover and Mutation

The crossover operation is then performed to the designated parent individuals. It selects a block of genes within one parent individual to be swapped with the ones selected from the other parent individual. This operation can be repeated multiple times to produce a set of "child" programs. Figure 2.8 gives an illustration of the crossover operation.

The mutation operation is performed to a select program in order to introduce changes within the program. The changes can be in the form of modification, addition or even removal of some genes. Figure 2.9 gives an illustration of the mutation operation. There is also the reproduction operation, which is used to duplicate a program and then insert the clone in the new population.

2.3.5. End Condition

The GP algorithm then repeats the whole process with the newly generated population. Each pass of this population modification is called a generation. It iterates through generations of programs until the end condition is reached. The end condition can be:

- when a perfect solution has been found,
- a predefined generation limit has been reached, or
- no significant improvement in the fitness value of the programs after several consecutive generations.

After the process terminates, the best program, the program with the highest fitness value, is returned as the result. The process is deemed successful when the returned program fulfills all specifications.

Needless to say, the success of each GP run depends on the design of the fitness evaluation function. A good evaluation function can tell, in fine granularity, how suitable a program is and express it in a numerical value. The design of a good fitness value requires deep knowledge in the domain and also a handful of creativity.

3

P4I/O: Intent-Based Networking with P4

Abstract. *Switches that can be (re)programmed through the network programming language P4 are able to completely change – even while in the field – the way they process packets. While powerful, P4 code is inherently static, as it is written and installed to accommodate a particular network requirement. Writing new P4 code each time new requirements arise may be complex and limits our agility to deal with changes in network traffic and services.*

In this chapter, we present P4I/O, a new approach to data-plane programmability based on the philosophy of Intent-Based Networking. P4I/O provides an intent-driven interface that can be used to install and/or remove P4 programs on the switches when needed and which is easy to use. In particular, to realize P4I/O, we (1) describe an extensible Intent Definition Language (IDL), (2) create a repository of P4 code templates, which are parsed and merged based on the intents, (3) provide a technique to realize the resulting P4 program in a programmable switch, while accommodating intent modifications at any time, and finally (4) implement a proof-of-concept to demonstrate that intent modifications can be done on-the-fly.

3.1. Introduction

Recent advances in data-plane programmability have enabled the implementation of customized high-speed network functions directly into programmable switches. Network operators can specify any pipeline processing logic by expressing their will in the form of a pipeline abstraction language, which was made famous by P4 [8].

The applications are virtually limitless, with examples ranging from performing network telemetry on the switches [27, 48], to fast congestion detection on the data-plane [65], to offloading distributed consensus algorithms to the data-plane [18].

Unfortunately, to reap the benefits of data-plane programmability, network operators have to face the burden of learning its associated abstraction languages, such as P4 [8], to be able to express their desired data-plane functionality. This translates to a steep learning curve, effectively creating a barrier before operators can reap the benefits of data-plane programmability.

To alleviate that burden, in this work, we present P4I/O, an Intent-Based Networking (IBN) framework that allows the user to express their network functionality intents in a close-to-English style, which subsequently is translated into P4 code, as illustrated in Figure 3.1, with the goal of facilitating the general public to attain the benefits of data-plane programmability. To realize P4I/O, we combine the notion of Intent-Based Networking (IBN) along with the concept of pipeline configuration abstraction, resulting in a Intent-Based P4 Code Generation technique. As network services are dynamic in nature, the P4I/O is designed to accommodate changes in the intent description and realize them right away with minimum disruption to the existing services.

In order to realize P4I/O, in this work, we present the following key contributions:

- **Extensible Intent Definition Language (IDL).** In order to describe various kinds of network services as intents, we devise in §3.3 a high-level language that is close to the

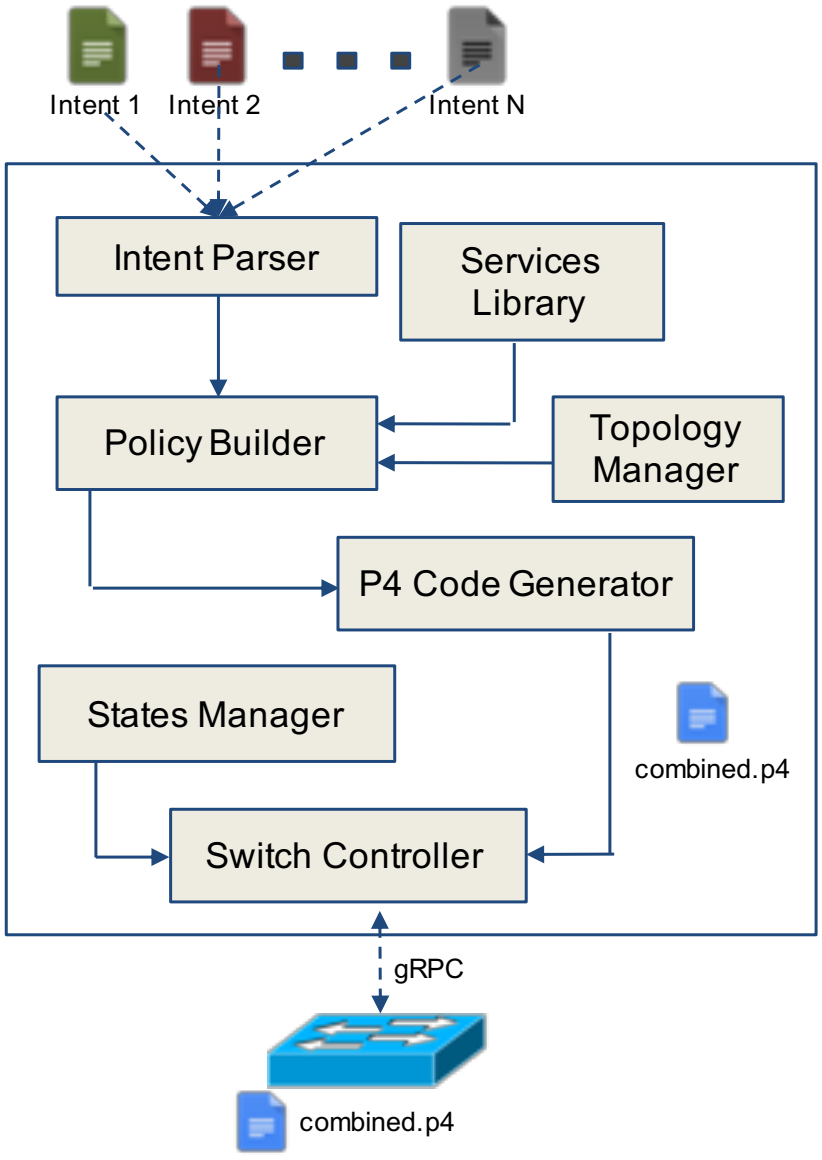


Figure 3.1: P4I/O architecture.

human language, yet precise enough to be interpreted unambiguously by the network controller. Furthermore, this language is extensible so that we can define any kind of data-plane functionality.

- **Template-Based P4 Code Generation.** We construct a repository of relevant network functions in the form of P4 code templates. These templates are then parsed and represented in a specialized data structure that facilitates combining the network functions, following the intent instructions. The code templates are then finally merged together to form a valid P4 program, as described in §3.4.
- **Dynamic Intents Realization.** We provide, in §3.5, a technique to install the resulting P4 code in a programmable switch, while permitting intent modification at any time. We realize intent modifications, with minimal disruption to the traffic forwarding process, through a state-transfer mechanism.
- **Framework Evaluation.** In §3.6, we demonstrate that P4I/O works, by building a proof-of-concept. P4I/O code has been released as open-source code [55].

3.2. Network Telemetry Use Cases

Our research has the main goal of developing a framework that has the capability to take intent descriptions as input and to generate pipeline configuration, in the form of P4 code, as output. P4I/O can be used for any network function, but to explain its building blocks, we will consider several use cases from the domain of network telemetry. Network telemetry functions provide interesting challenges to solve, because they require a state to be stored in the switch, e.g. in the form of P4 registers [61], along with complex pipeline processing logic to compute the states. We consider the following use cases: (1) Threshold-based Heavy-Hitter (HH) detection, (2) Distributed Denial of Service (DDoS) victim detection, and (3) Super-Spreader (SS) detection, which is the inverse of DDoS detection. All of these use cases are implemented using sketches, in particular *Count-Min Sketch* [16] and *Bitmap* [21].

For example, consider use case (1): HH detection is a mechanism to compute whether a packet is part of an HH flow, which we define as a flow that has more than our threshold of T packets. Consequently, the switch must track the number of packets in each flow, while it has very limited memory and processing resources. To effectively make use of the limited resources, we use the Count-Min sketch, which enables our switch to track a virtually unlimited number of flows at the expense of slightly reduced accuracy. Once we identify if a packet is a part of a heavy-hitter flow, we can take actions against it. To illustrate, some possible actions are: dropping it, marking it with a particular DSCP value, metering (rate-limiting) it, or even combination of them.

Our goal is to run a network in which network functions, such as the described telemetry functions, can be (de)activated at any time by adding/removing intents in the controller. Realizing this goal is challenging, because of the following issues:

- **Intents Composition.** As we may have several intents that run in parallel on the switch, the ordering of the execution in the pipeline processing will impact forwarding performance.
- **Hardware Constraints.** Physical switches have a limited amount of resources. We must be able to calculate how much resources will be consumed to run a set of functions even before implementing them in the switch. Failure to do so might lead to unpredictable switch behaviour.
- **State Preservation.** Intent modification translates to the changing of switch pipeline configurations. As each set of network functions has its own requirements for state containers, we must consider the problem of preserving the state from one set of functions to another. Moreover, as the new set of functions might not have the same state containers available, we need a mechanism to manage the unused state values. Finally, we also have to minimize the effect of intent modifications on the traffic forwarding process.

The following sections describe how we tackle these issues.

Listing 3.1: Drop Heavy-Hitter Action Example.

```
import drop_heavy_hitters

define intent drop_hh_any_any:
  from   endpoint('any')
  to     endpoint('any')
  for    traffic('any')
  apply  drop_heavy_hitters
  with   threshold('more or equal', 20)
```

3.3. Intent Definition Language

This section discusses the Intent Definition Language (IDL) that is employed to express network intents. We begin by identifying the requirements for such a language and subsequently present a working solution based on extending an existing IDL.

3.3.1. Requirements

While Han et al. [28] claim that there is currently no standardized definition of *network intent*, they also argue that intent is generally perceived as a business-level goal of how the network should behave, abstracting the implementation details from the operators. Reflecting on this objective, we identify the following requirements to be satisfied by our IDL:

- **Readability.** Network operators should be able to intuitively express their intended services with minimum training. We propose to cater to this need by developing a language that is close to a natural language, e.g. English, yet still concise enough to be interpreted unambiguously by the controller.
- **Abstraction.** The language should abstract out technical details of the implementation. To realize this, we have to move the details of the implementation to a lower layer. The lower-layer implementation should still be accessible by operators with advanced technical knowledge for debugging purposes.
- **Flexibility.** The language should be flexible enough to be extended with any kind of required network functions. This means that we have to design for a modular architecture, which facilitates easy extensions of new network functions.

3.3.2. Nile Language Extension

We employ Nile [31] as the base language for our IDL. Nile is a network intent language with the goal of providing an intermediate layer between a natural language and lower-level policies. In [31], user utterance is processed into a network intent expressed in the Nile language. Interested readers are referred to [31] for the grammar of Nile in *EBNF* notation. Nile satisfies our requirements of readability and abstraction, but does not facilitate the import of external module definitions. To that end, we introduce several constructs:

1. *import*: to define custom actions and import them from the actions repository,
2. *apply*: to apply the custom action by specifying the name of the action in the intent definition, and
3. *with*: to provide parameter values required by the custom action.

The import construct can be used multiple times to define more than one custom action. The specified action is then executed whenever the specified conditions in the intent definition are satisfied.

For example, consider the code snippet in Listing 3.1. We define a new action named *drop_heavy_hitters* that performs a threshold-based HH detection, with threshold $T = 20$

```
const bit<8> HH_THRESHOLD = {{ hh_threshold_val }};
```

```
const bit<8> HH_THRESHOLD = 1000;
```

```
if (meta.minRegVal > HH_THRESHOLD) {
    drop();
}
```

Figure 3.2: (Top) Template example with placeholder. (Mid) Template example with rendered placeholder value. (Bottom) Manipulation section example.

packets. The *drop_heavy_hitters* action is to drop HH flows. This intent applies to all traffic, from any source or destination.

3.4. P4 Code Templates

In this section, we describe our template-based method for forming P4 code. A knowledgeable party predefines P4 code templates that correspond to specific actions. The code templates are then imported into a special repository in the network controller. To render a template, the controller takes various attributes defined in the intent as inputs. If there is more than one intent or the intent itself is more complex, we may also need to "merge" multiple P4 code templates into one final P4 program.

The templates are written in a templating-language that has several constructs that are syntactically distinguishable from the actual text. The constructs act as placeholders that can be populated with string values. This way, the final P4 code can be rendered by populating each placeholder with precomputed string values. We employ Jinja2 [57], one of the most popular parameterized templating-languages, as our language of choice. Figure 3.2 (Top and Mid) illustrates the placeholder and its associated rendered code. The {{*variable_name*}} struct is used as a placeholder for a string named *hh_threshold_val*.

3.4.1. Network Telemetry Function Structures

The P4 code templates for the majority of network telemetry functions can be built upon the following structures:

- **Constant Definition.** Integer constants, that are needed for the function computation, are stored in this structure. The combination of constants can be done by a straightforward concatenation. Figure 3.2 (Top and Mid) depicts an example for P4 constant definition.
- **Parser Definition.** The state machine to parse the required packet headers.
- **Metadata Definition.** The metadata fields that are required by the network function to perform its computation. The metadata fields name in a P4 program should be unique. We can ensure the uniqueness of each field name by prepending a simple text that differs for each function.
- **Packet Identification.** In the packet identification section, we perform a computation to identify whether a packet belongs to a specified group of traffic (or vice versa). In our heavy hitter example, this structure is represented by the computation of the hash values and the update process of the register values. No action is enforced in the identification phase.
- **Packet Manipulation.** In this final phase, actions are taken on the identified traffic. In the P4 language, the possible actions are virtually limitless, but common ones are: count, mark, drop, meter and/or a combination of them. Considering our heavy hitter

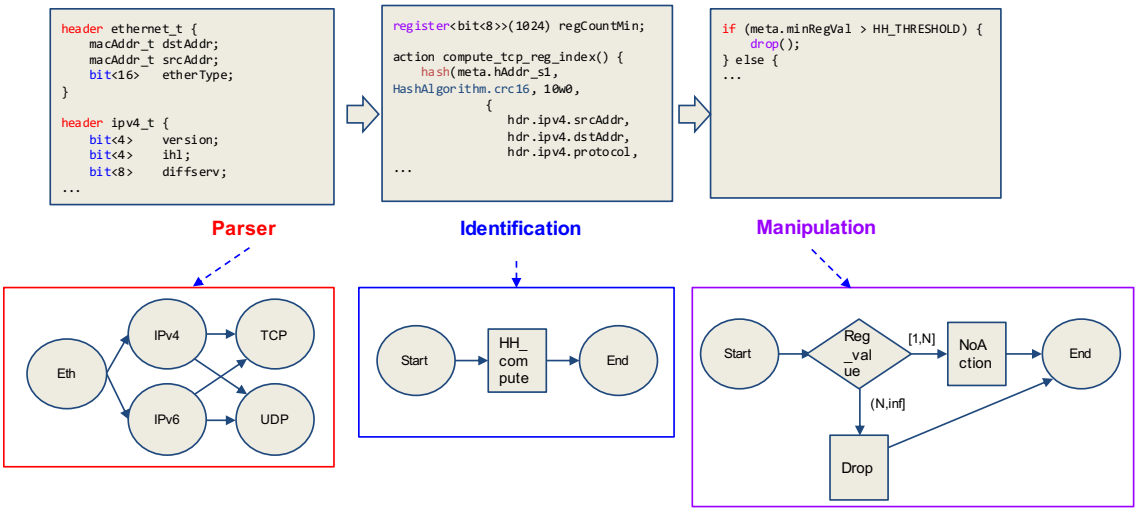


Figure 3.3: P4 code template translation to a Directed Acyclic Graph (DAG) structure.

example, an example of the action would be marking a packet with a specific DSCP value once a packet is identified to be part of a heavy hitter flow. Figure 3.2 (Bottom) depicts an example for this phase.

3.4.2. Template Representation

In this section, we present a formal data structure to facilitate combining intents and their corresponding P4 code. As the network functions that we are dealing with are logical representations of packet pipeline processing, the data structure should also be able to move from one condition to the next, which is possible via a Directed Acyclic Graph (DAG) that is based on PGA's [53] graph structure.

Policy Graph Structure

To help illustrate our graph structure, we use the HH example code in Figure 3.3. The constant and metadata definitions contain no flow information and therefore are not processed further. The parser definition code can be represented as a directed graph with the vertices representing the packet header names and the edges representing a possible state transfer from one header to another.

Combining Templates

We proceed to define how several policy graphs can be combined. Each of the phases in §3.4.1 has different characteristics and should be treated differently. The simplest cases are the constant and metadata definitions. To combine constant and metadata definitions from several intents, we need to make the names unique by prepending each name with a unique text – preferably generated from the intent id number – and then do a string concatenation to combine all of the constant and metadata definitions from various intents together.

For the parser graphs, we expect many overlapping nodes coming from several policies, which can be resolved via computing the union of the vertices and the union of the edges from all of the graph policies. The resulting edges and vertices comprise the final graph for the parser.

Finally, the identification and manipulation actions are stitched together one after another. Formally, let's consider two intents $I_1 := (i_i, m_1)$ and $I_2 := (i_2, m_2)$, with (i_n, m_n) defined as a sequence of identification and manipulation actions. After the combination, we get the action sequence of $Combined := (i_1, m_1, i_2, m_2)$.

P4 Code Generation

For the final P4 program, we need another *base template* that contains placeholders for the aforementioned P4 code structures in §3.4.1. This *base template* represents a valid P4 program with several empty sections, ready to be filled in with the rendered policies.

3.5. Intent Realization

This section describes the realization of the final P4 program that we generated in §3.4.2 into the network. We also aim to design a solution that is able to handle intents modification, i.e. added or removed, at any time. We start by defining the software components used to realize the controller and finally discusses how the system handles intent modification. For ease of explanation, we limit the scope of the intent realization to one programmable switch connected to several hosts, but our framework can be extended to intent realization over multiple switches.

3.5.1. Software Components

To realize the solution that we have described in §3.4, we design a software that is composed as depicted in Figure 3.1, P4I/O consists of the following components:

1. **Intent Parser.** The intent parser parses the intents and stores them in a hierarchical key-value storage intended to be used by the policy builder.

2. **Actions Library.** The actions library acts as a repository for defined actions represented as P4 code templates. It parses the P4 code templates into a policy graph as explained in §3.4.2.

3. **Topology Manager.** The topology manager keeps track of each host connected to the switch ports. The information of which host is connected to which port is used to build a correct P4 match-action table entry required for the packet forwarding process.

4. **Policy Builder.** The policy builder executes the policy graphs join computation as explained in §3.4.2. It outputs the combined policy graph required to generate the final P4 program.

5. **P4 Code Generator.** The P4 code generator has as main objective to generate a working P4 program with the input of a policy graph, as described in §3.4.2.

6. **States Manager.** The states manager keeps track of all match-action table entries that are implemented on the switch. Each time a new P4 program is installed in the switch, the switch will lose all of its old entries and therefore needs to be repopulated by the entries stored in this component.

7. **Switch Controller.** The switch controller is responsible for pushing the generated pipeline configuration file into the switch and maintaining the communication channel to the switch, e.g. via gRPC. This process is done on-the-fly, while the switch is forwarding traffic. The push process is done by calling the *setPipelineConfig* procedure, as defined in P4Runtime [63].

3.5.2. Intent Modification

Switch pipeline configuration modification is realized via the interface defined in the P4Runtime specification [63], via the procedure of *SetForwardingPipelineConfig*. By default, each time a BMV2 [6] *simple_switch_grpc* is instructed to load a new pipeline configuration, it will lose all of the match-table entries and all of the other states, e.g. register, counter, meter, etc. We refer to this problem as the *state preservation problem*.

We handle the problem of state preservation by providing two mechanisms:

1. **Match-Table Entries Preservation.** Preservation is done by rewriting the match-action table entries every time a switch pipeline configuration is reloaded. The entries are stored at the *State Manager* component. However it should be noted that the speed of rewriting table entries could be a potential bottleneck for the whole configuration reload process.

2. **Switch States Preservation.** The states in the switch can be preserved via two approaches: *external* and *internal backup*.

(a) ***External Backup.*** In this approach, the value in the states is first read by the network controller. After the states are completely read, the controller reloads the switch with the new configuration and writes back the state arrays into the switch. The writing process can be done in two ways: (1) multiple single values are written into the switch memory, or (2) single multiple values are written by passing the whole array in one write procedure. The default *simple_switch_grpc* only supports method (1).

(b) ***Internal Backup.*** This method does the backup within the internal memory of the switch. As it does not require any external communication, there will be no external communication overhead. However, the switch should have enough memory for the backup and dedicate some computation resources for the states duplication procedure.

For our proof-of-concept, we adopt the *internal backup* mechanism by developing a custom P4 software switch based on BMV2's *simple_switch_grpc* that has the functionality to preserve state from one configuration to another. It will first store all of its state to its internal memory. After the new pipeline configuration is enforced, the switch will try to restore the backed-up state, provided the previous state container still exists in the new one. The algorithm for the switch state preservation is depicted in Algorithm 1. This process happens while incoming packets are temporarily stored on an input buffer in the switch.

foreach *registers* ∪ *counters* ∪ *meters* ∪ ... **do**
  $backupState[stateName] \leftarrow oldValue$;
end
enforceNewPipelineConfig;
foreach *element in backupState* **do**
  if *element exists in new pipeline* **then**
    $newValue \leftarrow backupState[stateName]$;
  end
end

Algorithm 1: State Transfer Algorithm.

3.6. Evaluation

We evaluate the feasibility of P4I/O through a proof-of-concept implementation. We begin with describing the setup to realize P4I/O before closing it with a discussion of our result. To note, our prototype implementation is written in Python and contains approximately 4.530 lines of code.

3.6.1. Proof of Concept Setup

We define 6 epochs, $t'_0, t'_1, .., t'_5$, to aid us in evaluating intent modifications on P4I/O. In each epoch, different intents are defined. In t'_0, we run DDoS and SS detection functions. In t'_1, we add one Heavy-Hitter (HH) detection intent, which drops traffic exceeding a threshold of $T = 1800$ packets. In t'_2, we remove the SS detection. Further, in t'_3 we also remove the HH detection, leaving us with only DDoS detection. We then add back HH detection in t'_4, this time with a threshold of $T = 3600$ packets. Finally, we enter the last epoch of t'_5 by removing the HH detection function, effectively leaving DDoS detection as the only running function.

Formally, we can express the network functions in each epoch as the following sets: $t'_0 = \{DDoS, SS\}$, $t'_1 = \{DDoS, SS, HH(T = 1800)\}$, $t'_2 = \{DDoS, HH(T = 1800)\}$, $t'_3 = \{DDoS\}$, $t'_4 = \{DDoS, HH(T = 3600)\}$, and $t'_5\{DDoS\}$.

We use a simple topology with our modified *simple_switch_grpc* software switch connected to a traffic generator and a traffic receiver as illustrated in Figure 3.4. The generator then injects random UDP traffic containing random payload with the traffic rate of 1 megabyte per second. We use a single IP address per sending/receiving side. The topology is run on top of

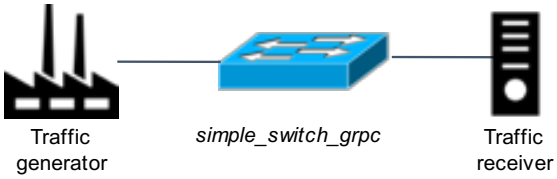


Figure 3.4: Proof-of-Concept topology.

Mininet [41], which in turn runs on top of a Ubuntu Linux 16.04 desktop with a dual-core 1.6 GHz Intel i5 processor and 2GB of RAM. The resources are shared between the host, Mininet, *simple_switch_grpc*, and P4I/O.

3.6.2. Result and Discussion

Figure 3.5 depicts the throughput graph as observed from the receiving side. At epoch t'_0, the traffic rate is relatively stable until we reach t'_1, which temporarily causes a fluctuating rate, due to the actualization of the new P4 code. We can note that at approximately $t = 27s$, the HH threshold $T = 1800$ is reached causing the traffic to be dropped. At epoch t'_2, the HH state from the previous epoch is successfully preserved as proven by the absence of traffic. The removal of HH detection network function at the beginning of epoch t'_3 gives us back the inbound traffic. Likewise, t'_4 also demonstrates the same phenomenon as t'_1, but with a twice as long delay before we reach the new threshold of $T = 3600$.

Reflecting on this result, we believe that our approach is applicable to real-world use cases. We can conclude that the existing hardware is fast enough to do pipeline configuration modifications with minimum interruption to the traffic forwarding process. While we see some fluctuating throughput rate in the result, it can be explained by the condition of our testbed that performs all computation, e.g. P4 code generation and traffic forwarding, in a single machine.

3.7. Related Work

Programmable Network. Pyretic [46] allows building network services by means of network programmability. However, their approach to the network programming language is of imperative nature. PGA [53] addresses the problem of network policies reconciliation, i.e. aligning overlapping/conflicting policies, by devising a graph abstraction that inspires our DAG abstraction. PGA is implemented as an extension of Pyretic. Janus [3] extends the work of PGA by adding the notion of dynamic policies and incorporating QoS constraints. Janus' implementation is also based on Pyretic.

Intent-Based Networking. Nile [31] provides a human-readable IDL for implementing network services and is focused on translating user utterance into an IDL and incorporating user feedback to improve the translation model. Marple [48] and Sonata [27] generate pipeline configurations – also in P4 – from dynamic queries, but do not focus on the technique for generating the code. Moreover, they do not consider the problem of intent modification on-the-fly. Donovan & Feamster [20] propose the notion of intention-based monitoring, which offloads the task of matching traffic to the data-plane. Their Pyretic-based implementation matches traffic based on attributes with static mapping like domain name and AS number.

Sketch-Based Network Telemetry. OpenSketch [70] utilizes sketches for various flow measurement tasks. However, their implementation is on NetFPGA, which provides no mechanism to reload the switch pipeline configuration on-the-fly. UnivMon [42] contests OpenSketch's approach by proposing a universal sketch algorithm adopted from universal stream theory. UnivMon focuses on evaluating the performance and memory utilization of the universal sketch.

3.8. Conclusion

In this work, we have presented P4I/O, an intent-based networking framework that facilitates a simple adoption of P4 data-plane programmability. Through P4I/O, programmable

switches can be quickly and easily programmed, without having to learn the corresponding data-plane programming languages, such as P4. In order to build P4I/O, we have described an extensible Intent Definition Language, used a P4 code template approach, and enabled intent modifications on-the-fly. We have made an open-source Proof-of-Concept implementation of P4I/O, with which we have demonstrated that intents can indeed be installed/removed in the field.

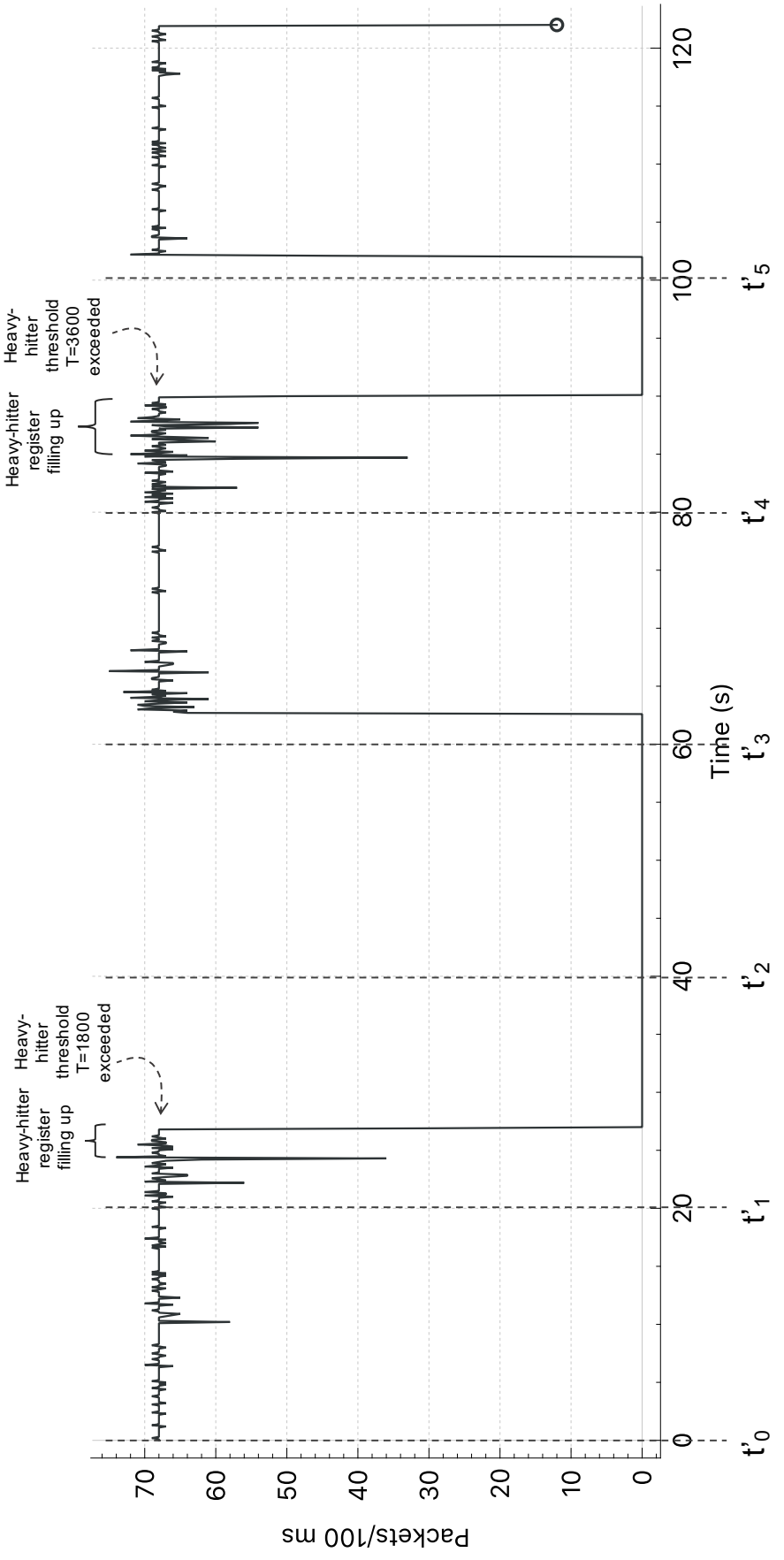


Figure 3.5: Throughput as observed from the receiving side in various epochs.

4

GP4P4: Enabling Self-Programming Networks

Abstract. *Recent advances in programmable switches have enabled network operators to build high-speed customized network functions. Although this is an important step towards self-* networks, operators are now faced with the burden of learning a new language and maintaining a repository of network function code. Inspired by the Intent-Based Networking paradigm, we propose a new framework, GP4P4: a genetic programming approach able to autonomously generate programs for P4-programmable switches directly from network intents. We demonstrate that GP4P4 is able to generate various network functions in up to a few minutes; an important first step towards realizing the vision of 'Self-Driving' networks.*

4.1. Introduction

The concept of Self-Driving Networks, analogous to the concept of Self-Driving Cars, has been a Utopian dream in the field of computer networks. That ultimate goal of running a network that behaves solely based on our intent is rapidly coming in reach through fast advances in the domains of network programmability and artificial intelligence [10, 35].

The introduction of the network programming language P4 [9], which allows for data-plane programmability, has enabled network operators to construct high-speed network functions customized to their own needs. However, this does require them to create and maintain a large library of network function code, which is prone to human error: a single mistake can render the system unusable. Moreover, data-plane programming languages keep evolving and new languages are constantly introduced. For example, the move from $P4_{14}$ [62] to $P4_{16}$ [61] introduced major non-backward-compatible changes to the language. As languages keep evolving, to remain up to date, a network operator would need to adjust his entire catalog of code templates.

To avoid these problems entirely, we propose to leave the programming of the network to the network itself by enabling it to automatically generate data-plane code based on sets of less complex, human-readable rules or intents. As a proof of concept, we present GP4P4, a framework that automatically generates data-plane programs satisfying sets of behavioral rules based on first-order logic. GP4P4 enables operators to modify their network functionality near instantaneously without modifying any code themselves. Figure 4.1 depicts a high-level rendition of GP4P4.

To the best of our knowledge, GP4P4 is the first framework for automatically generating data-plane code. We believe that this framework is an important first step towards a future where *self-programming* networks can fully program and adapt themselves to their current goals and circumstances with minimum intervention by network operators.

Machine-learning has recently been applied within the control-plane (see [67]) and to boost the performance of network functions (e.g. [23, 24, 68]), but the utilization of machine learning techniques to generate network functions themselves has yet to be considered. Also, a

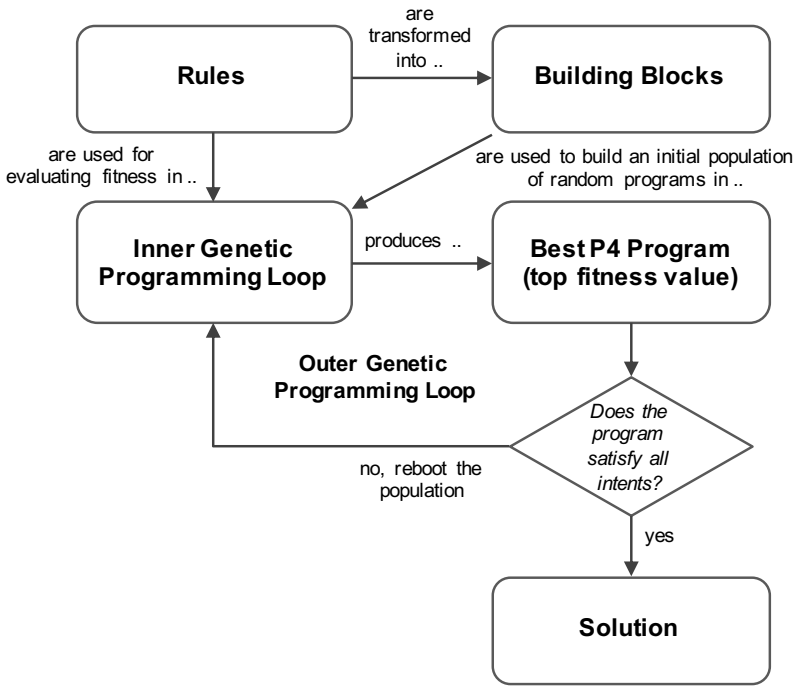


Figure 4.1: LGP4P4 overview.

few position papers on self-driving networks have appeared [22, 23, 32, 33, 69], but again a concrete framework that enables the network to program itself is missing.

Our main contributions are:

1. GP4P4 itself, a framework for automatically generating P4 programs using techniques adapted from Linear Genetic Programming (LGP). LGP is a machine-learning technique to "evolve" an initially randomized population of programs towards satisfying an objective function [11],

2. an evaluation module required to make LGP suitable for data-plane programmability. Our proposed evaluation module evaluates programs by creating synthetic network traces and simulating the output of P4 programs on these traces. In this regard, GP4P4 is fully self-sufficient and does not depend on any external network traces or physical switches, and

3. a Proof-of-Concept experiments demonstrating the efficacy of GP4P4.

4.2. GP4P4

Figure 4.2 gives a high-level overview of GP4P4. Behavioral rules – the intents of the network operator – lie at the base of our framework. They are analyzed to obtain the P4 building blocks which the framework uses to create P4 programs, as well as for evaluating the programs during and after their generation. In the inner loop, GP4P4 evolves programs using *Linear Genetic Programming (LGP)*, a machine-learning technique to "evolve" an initially randomized population of programs towards satisfying an objective function [11]. If the best of these programs satisfies all behavioral rules, GP4P4 presents this program as the solution. If not, it reboots the population of P4 programs and restarts the inner loop. This process continues until a solution has been found.

4.2.1. Behavioral Rules

We describe combinations of network functions as a set of *behavioral rules* on packet attributes (headers and metadata). If these rules are followed for each packet, we deem the

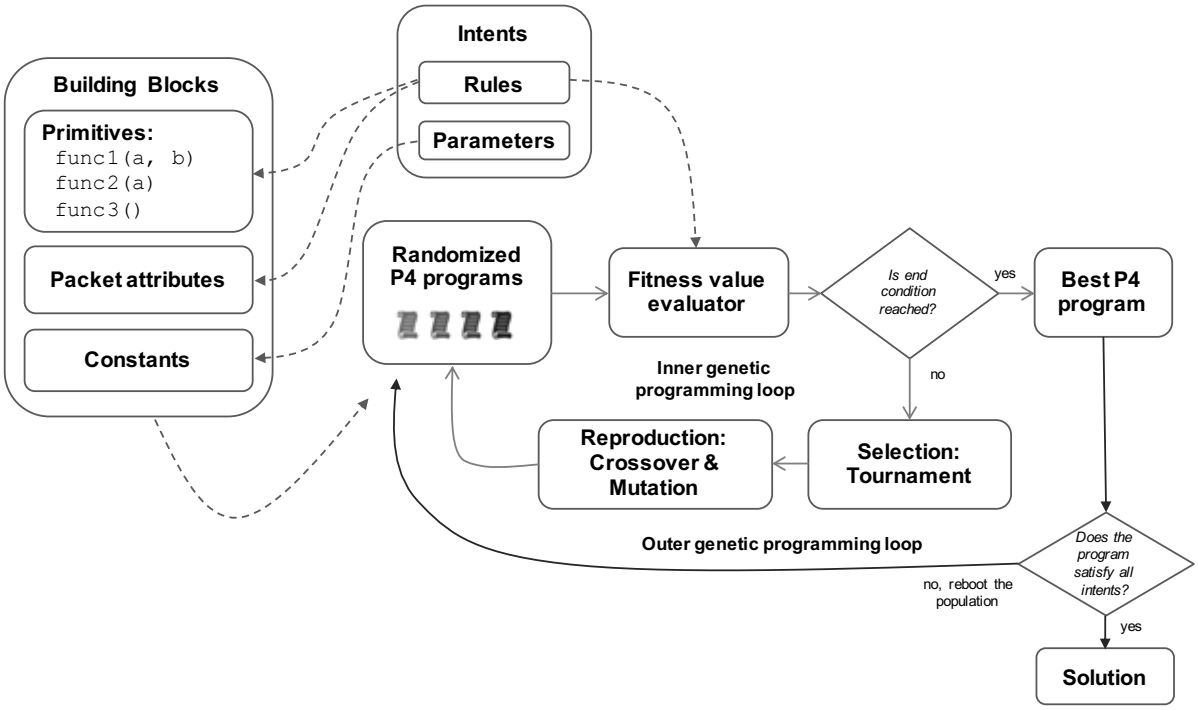


Figure 4.2: Program generation overview.

P4 program *valid*. In contrast to P4 programs themselves, these rules describe the intended outcome of a program, and not its methodology. In other words, these rules can be seen as a low-level description of the intents of a network operator. The goal of GP4P4 is to automatically generate a valid P4 program for any provided set of behavioral rules. Consequently, to change the functionality of their switches, network operators only need to state their intents in the form of behavioral rules, after which a P4 program can automatically be generated and installed in the network.

In GP4P4, behaviour rules are expressed in the form of IF-THEN predicates connecting packet input conditions to output conditions. For any packet for which the input conditions (IF) are true, we require the output conditions (THEN) to be also true. Both the input and output conditions are expressed as one or more equal (EQ) or not equal (NEQ) Boolean expressions on packet attributes combined with AND. Packet attributes are referenced using dot notation (e.g. `pkt_in.src_ip`). For example, the rule

```
IF (pkt_in.src_ip EQ 192.168.1.1)
THEN (pkt_out.out_port EQ 2)
```

means that if the source IP address of an incoming packet is 192.168.1.1, the packet should be outputted from port 2. We require all input conditions (IF statements) to be mutually exclusive.

NAT Example

Consider the example situation in Figure 4.3. A P4 switch connects two networks, *inside* and *outside*. To apply NAT, we want the switch to replace the source destination IP address of host H1 in *inside*, 192.168.1.1, with 10.0.0.10 in any outgoing packet, and the destination IP address 10.0.0.10 with 192.168.1.1 in any incoming packet. The IP addresses of all other packets should be left unchanged. To determine if packets are from inside or outside, we can match on their input port (`pkt_in.port_num`); packets arriving at port 0 are moving from inside to outside, while packets arriving at port 1 are moving from outside to inside. Figure 4.5 shows the resulting 4 behavioral rules for this network function, while Figure 4.4 illustrates a flow-chart that covers all possible input cases in the NAT example.

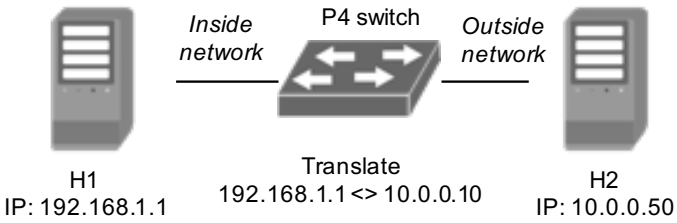


Figure 4.3: NAT example topology.

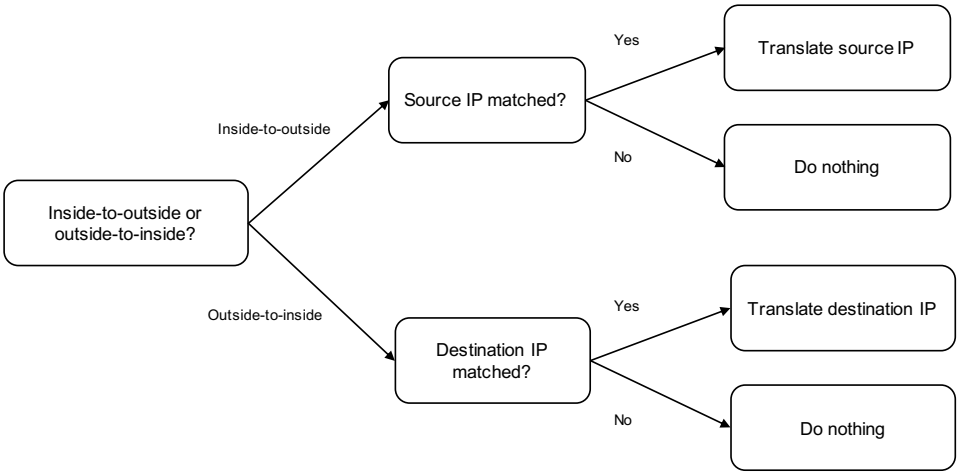


Figure 4.4: NAT example behavior Flow-chart.

```
RULE1:
IF   (pkt_in.port_num EQ 0
      AND pkt_in.src_ip EQ 192.168.1.1)
THEN (pkt_out.src_ip EQ 10.0.0.10
      AND pkt_out.dst_ip EQ pkt_in.dst_ip)
RULE2:
IF   (pkt_in.port_num EQ 0
      AND pkt_in.src_ip NEQ 192.168.1.1)
THEN (pkt_out.src_ip EQ pkt_in.src_ip
      AND pkt_out.dst_ip EQ pkt_in.dst_ip)
RULE3:
IF   (pkt_in.port_num EQ 1
      AND pkt_in.dst_ip EQ 10.0.0.10)
THEN (pkt_out.src_ip EQ pkt_in.src_ip
      AND pkt_out.dst_ip EQ 192.168.1.1)
RULE4:
IF   (pkt_in.port_num EQ 1
      AND pkt_in.dst_ip NEQ 10.0.0.10)
THEN (pkt_out.src_ip EQ pkt_in.src_ip
      AND pkt_out.dst_ip EQ pkt_in.dst_ip)
```

Figure 4.5: NAT example behavioral rules.

4.2.2. Program Generation

In Genetic Programming (GP), an initially randomized population of programs is gradually evolved to satisfy an objective function by selection and reproduction, similar to the biological concept of natural selection. To move through the search space, reproduced programs are

randomly modified by *mutation* and/or *crossover* operations.

An important concept in GP is that of the phenotype versus the genotype. The phenotype is the program itself, while the genotype is an internal lower-level representation of the program that is more suitable for GP. Given a genotype, we can directly construct the phenotype by de-encoding this representation. In practice, there are three major genotype representations in GP: (1) linear, (2) tree-based, and (3) graph-based. In a tree-based approach, programs permanently branch of after every IF-statement. Thus, this representation is more likely to evolve nested IF-statements than successive IF-statements. The graph-based approach, such as the one implemented by Cartesian Genetic Programming (CGP) [45], does not suffer from this "problem," but limits our ability to perform crossover operation[1]. Crossover is vital for creating programs that satisfy our behavioral rules, as it allows a program that satisfies one rule to "merge" with a program that satisfies another rule to, hopefully, create an offspring that satisfies both rules. As we want to be able to evolve programs with both nested and successive IF-statements, as well as make use of crossover, we choose to use Linear Genetic Programming (LGP) in GP4P4. LGP evolves sequences of *primitive*s of an imperative programming language. Each of these primitives represents a snippet of code in the phenotype program. In GP4P4 each primitive corresponds to a basic one-line declaration in P4, such as `src_ip = 10.0.0.10;`. We assume a set of primitives is provided to GP4P4 every time a new program has to be generated.

Figure 4.2 gives an overview of the program generation module of GP4P4. Program generation in GP4P4 runs in two loops: the outer and inner loop. In the inner loop, LGP is applied to evolve an initial random population of programs, i.e. sequences of primitives, into a program that satisfies all behavioral rules. The inner loop finishes either when a solution has been found or when this process takes too long. In the later case, the outer loop restarts the inner loop with a new initial population of random programs. This process helps prevent the program generation module from getting stuck in a sub-optimal local minimum.

Building Blocks

A primitive may read (write) from (to) any switch register, metadata value, or packet header. Additionally, a primitive may also access one or more constants (e.g. port *0*). In GP4P4 we treat all these input/output locations and values as *register*s. We store these registers in a single array, and allow a primitive to operate on any combination of registers in this array. Note that this means that the same primitive may correspond to different P4 declarations, depending on which registers it accesses. The registers of constants are read-only and are not allowed to be written to. Figure 4.6 shows an example of the combination of GP4P4 primitives, registers, and an individual program. Note that we do not have to store the explicit values of each register, but only to which part of the memory or packet they refer. We refer to the combination of registers and primitives as the *building block*s of a GP4P4 program.

To construct GP4P4 primitives, P4 declarations are simplified and written in prefix notation. For example, the P4 declaration `var3 = var5;` is transformed to the GP4P4 primitive `ASSIGN(var3, var5)`. Although P4 declarations are normally written in infix notation, it is easier to encode genes in prefix format. The if-then statement (`if() { }`) is cut into two primitives, corresponding to `if() {` and `}`. In addition, we restrict these primitives to two input registers, and create a separate primitive for each possible comparison operator: `IF_EQ(a,b)` for `if (a == b) {`, `IF_NEQ(a,b)` for `if (a != b) {`, and `ENDIF` for `}`. We do not allow any other control-flow statements, such as the else statement. Although this choice of primitives is rather limited, it still supports a wide range of possible declarations, albeit in the form of multiple primitives. For example, `if(a == b && b == c) {` is represented as `IF_EQ(a,b), IF_EQ(b,c).`

P4 allows for a wide array of possible memory locations, metadata values and packet headers. Including all these possibilities as registers would severely hamper the ability of GP4P4 to evolve programs into the right direction. Thus, GP4P4 automatically extracts all registers from the behavioral rules themselves. A packet attribute or constant is included as a register if and only if it is used in at least one of the behavioral rules.

[1] Combining information from two parent programs to create new offspring.

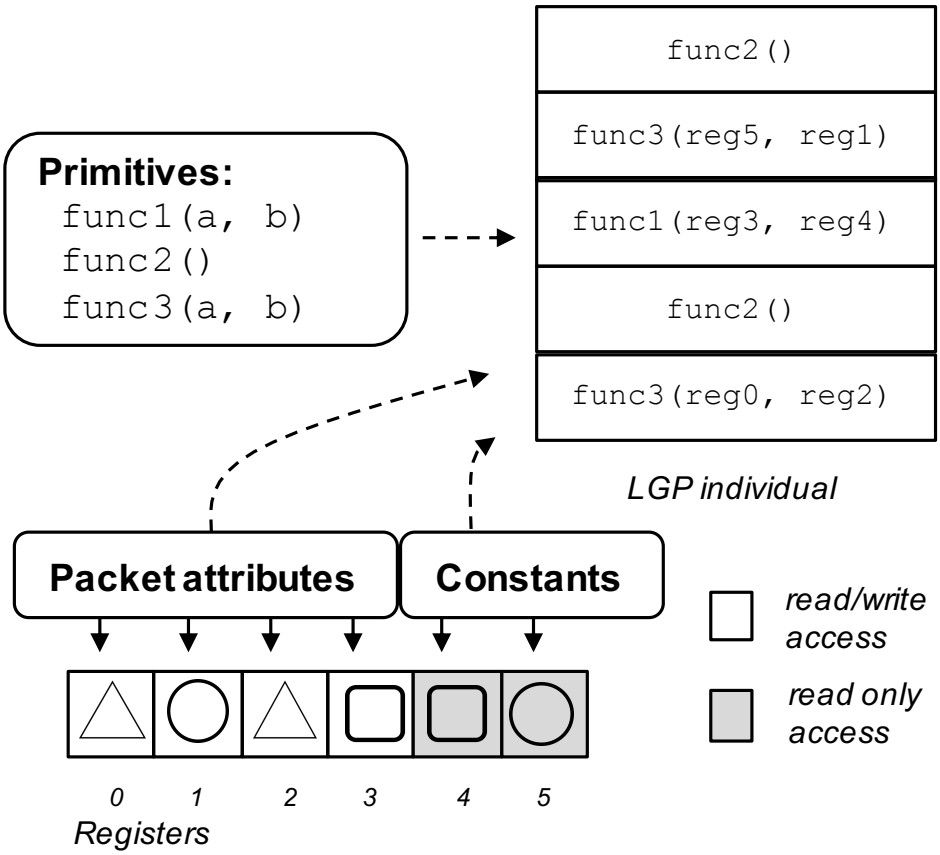


Figure 4.6: Example primitives, registers, and GP4P4 program genotype.

```
if port_num == 0 {
 if src_ip == 192.168.1.1 {
  src_ip = 10.0.0.10
 }
}
if port_num == 1 {
 if dst_ip == 10.0.0.10 {
  dst_ip = 192.168.1.1
 }
}
```

(a)

```
IF_EQ(port_num, 0)
 IF_EQ(src_ip, 192.168.1.1)
  ASSIGN(src_ip, 10.0.0.10)
 ENDIF()
ENDIF()
IF_EQ(port_num, 1)
 IF_EQ(dst_ip, 10.0.0.10)
  ASSIGN(dst_ip, 192.168.1.1)
 ENDIF()
ENDIF()
```

(b)

Primitives:
IF_EQ(a, b)
ENDIF()
ASSIGN(a, b)

Inputs:
port_num
src_ip
dst_ip

Constants:
0
1
10.0.0.10
192.168.1.1

(c)

Figure 4.7: NAT function code: (a) in P4 language, (b) in GP4P4 notation, (c) building blocks

As an optimization step, we categorize each attribute and constant by its data type, such as integer, string, Boolean, or IP address. We then limit the registers each primitive is allowed to access by type. This reduces the search space and ensures each primitive + register combination translates to correct P4 code.

Figure 4.7 depicts an example of a P4 code snippet and its translation in the prefix notation that is used throughout this work. Note that these two formats represent the exact same P4 code.

Initial Population

To initialize the inner loop, we generate a population of N syntactically correct programs of primitives with a length between min_len and max_len. To construct each program, GP4P4 first randomly picks a program length between min_len and max_len. Then it randomly selects this number of primitives (with replacement), randomly selects valid registers for each primitive, and puts the primitives in sequence. This process is repeated every time the inner loop is restarted.

Selection and Reproduction

Within each iteration of the inner loop, GP4P4 holds two tournaments between $t_r \times N$ randomly selected programs, where t_r is the tournament size ratio. The program with the highest fitness value of each tournament (or winner) is chosen for reproduction, while the bottom n_r programs (or losers) of each tournament are chosen to be replaced by the offsprings of the two winners. In LGP, the new set of programs created by replacing the losers by the offspring of the winners is called a new *generation*. The inner loop continues the process of generating new generations until it either finds a valid program or reaches a predefined generation limit.

Each of the $2 \times n_r$ offspring is created in pairs of two:

1. Duplicate both winners
2. Perform a crossover between both offspring with probability P_c
3. Mutate offspring 1 with probability P_m
4. Mutate offspring 2 with probability P_m

To reduce the computation time, we compute and store the fitness value of each program as soon at it is created. This way, we reduce the number of fitness values that need to be computed every iteration from $t_r \times N$ to $2 \times n_r$.

Mutation

To mutate a program, GP4P4 first selects a random index i in the program. Then, with equal probability, it either adds a new random primitive to the program at $i+1$, removes the current primitive at i, or replaces the current primitive at i with a new random primitive. Random primitives are generated in the same way as described previously in Section 4.2.2, with a few notable exceptions: To help evolve the program towards satisfying new rules, we generate new random if-then primitives with a higher probability than other primitives. GP4P4 selects a new, random if-then primitive with probability P_{if} and a non-if-then primitive with probability $1 - P_{\text{if}}$. In addition, to prevent new if-then primitives from dropping the fitness level of the program, GP4P4 adds an `ENDIF()` primitive directly after every new if-then primitive it adds to a program. Similarly, when removing an if-then or `ENDIF()` primitive, GP4P4 also removes the corresponding `ENDIF()` or if-then primitive.

Crossover

To perform a crossover between two programs, GP4P4 randomly selects a *unit* of code of both programs and swaps these units with each other. In GP4P4, a unit is either a single non-if-then primitive or a sequence of primitives starting with an if-then primitive and ending with its corresponding `ENDIF()` primitive. By only swapping valid blocks of code, we ensure that the resulting two programs remain syntactically valid. The crossover process is demonstrated in Figure 4.8.

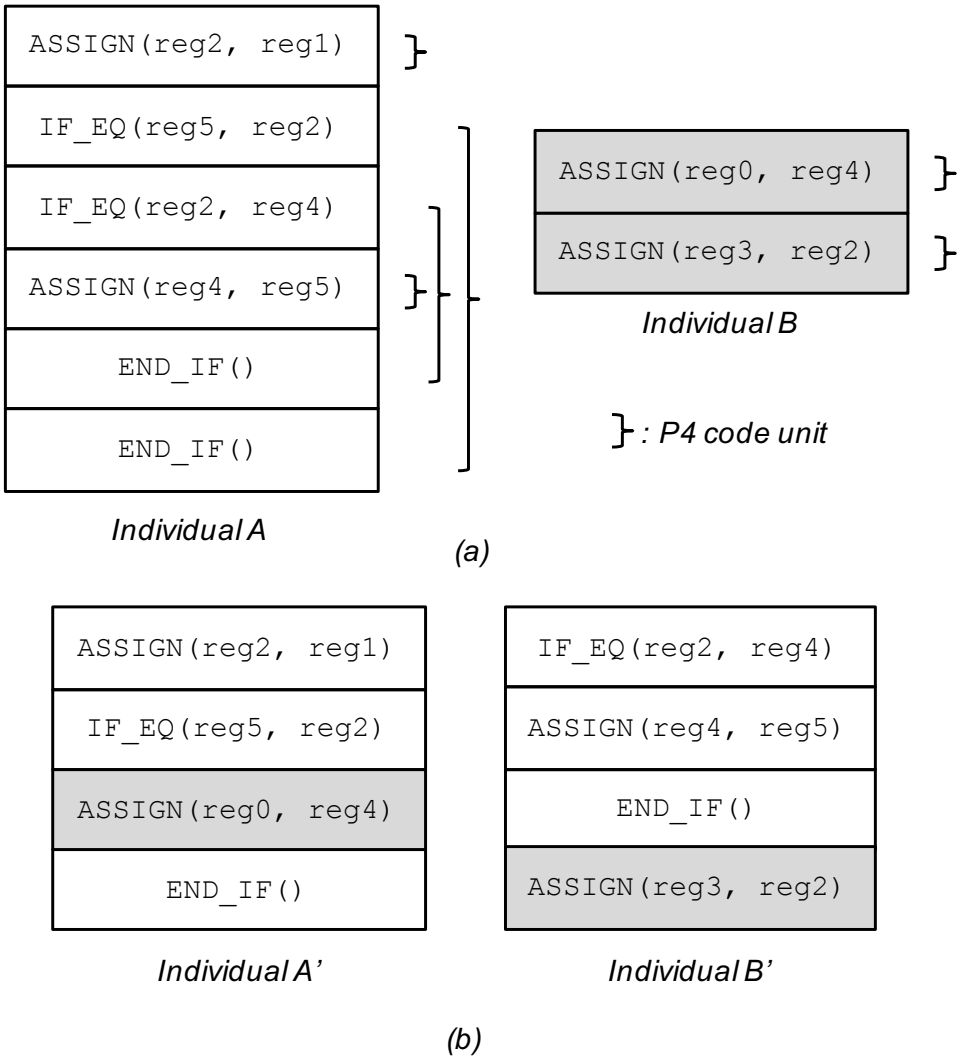


Figure 4.8: Unit-based crossover in GP4P4: (a) parent individuals, (b) offspring individuals. The '}' symbol denotes a swappable unit.

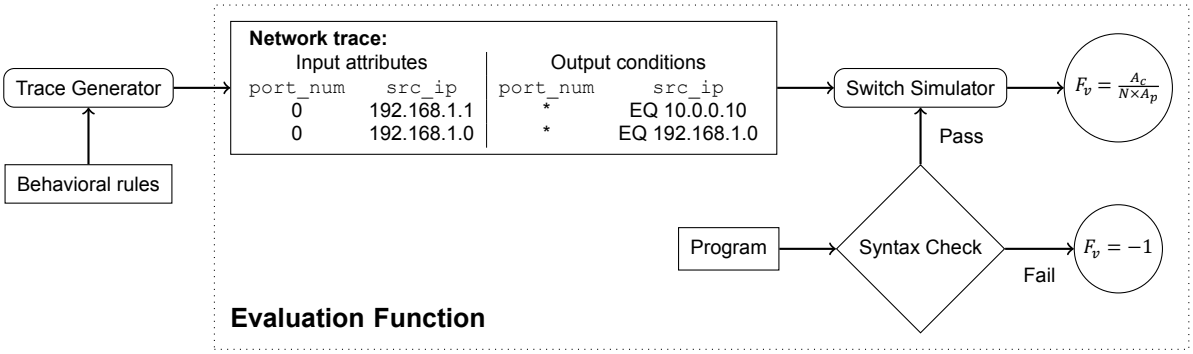


Figure 4.9: Evaluation module overview.

4.2.3. Program Evaluation

The evaluation module plays a critical role in GP4P4, as it guides the evolution of programs in the right direction, as well as checks if a program satisfies all behavioral rules. A good evaluation function should evaluate, in fine granularity, how close a program is to satisfying all rules and express this in a numerical value. In the case of P4 programs, this is not

Data: Behavioral rules R_1 to R_M, packet multiplier k
Result: Network trace of input packet attributes $P_1, \ldots, P_{M \times k}$ and output conditions $C_1, \ldots, C_{M \times k}$
foreach *Rule* R_i **do**
  for $1 \leq j \leq k$ **do**
    index $:= (i-1) \times k + j$;
    Construct semi-randomized packet P_{index} that satisfies the IF conditions of R_i;
    Set the output conditions C_{index} to the THEN conditions of R_i;
  end
end

Algorithm 2: Trace Generator.

a straightforward process, as programs may seemingly satisfy a rule for one packet, while breaking it for another. Figure 4.9 gives an overview of the evaluation module.

First, we check the syntax correctness of the program by inspecting it in *Syntax Checker* module. A syntactically incorrect program is assigned a fitness value of -1 and will not go through further checks. Further, the *Trace Generator* generates a synthetic network trace of packets and output conditions based on the behavior rules supplied to the framework. Next, the *Switch Simulator* simulates the program and processes the network trace. For each packet in the trace, the simulator counts the number of packet output attributes that satisfy the behavioral rules.

Next, the fitness value F_v is calculated using the following equation:

$$F_v := \frac{A_c}{N \times A_p}, \tag{4.1}$$

where:

- Fv: the fitness value
- A_c: the total number of valid output attributes over all packets in the trace
- N: the total number of packets in the network trace
- A_p: the number of output attributes per packet

As the fitness value is normalized, a fitness value of 1 denotes a program that satisfies *all* output conditions of the behavioral rules.

Trace Generator

The Trace Generator is responsible for generating network traces to evaluate generated programs on. In addition to generating the input packet attributes themselves, the Trace Generator also generates the corresponding conditions on the output packet attributes, so the Switch Simulator can quickly evaluate each program. To reduce computation time, the same network trace is re-used throughout the inner and outer genetic programming loops. Thus, the Trace Generator is only run once, just before starting the outer genetic programming loop.

Algorithm 2 describes the Trace Generator. For each rule, the Trace Generator creates k packets. Packet input attributes are created in a semi-randomized fashion to match the IF conditions of the rule, while the output attribute conditions are directly taken from the THEN conditions of the rule. By creating packets for each rule, we ensure that the fitness evaluation function evaluates programs on each rule as well. As the IF conditions of the behavioral rules are required to be mutually exclusive, the Trace Generator only needs to consider one rule at a time. To reduce computation time, the same network trace is re-used throughout the inner and outer genetic programming loops. Thus, the Trace Generator is only run once, just before starting the outer genetic programming loop.

Switch Simulator

Compiling a program to P4, and then running the program on a real or emulated switch can take up a significant amount of time. Thus, this approach would not be practical for GP4P4, which constantly needs to evaluate new programs. We propose running and evaluating each program on a simulated switch instead, while guaranteeing the same output/fitness as a real switch. This way, we can still assign accurate fitness scores to programs and test if they satisfy all behavioral rules, without wasting time on P4 code compilation and installation. The GP4P4 Switch Simulator assigns an evaluation score to programs by simulating their output for each packet of the network trace. As the simulator and an actual switch would both give exactly the same output, the fitness value as determined by the simulator is equal to that determined by an actual switch as well.

To save time, the simulator (written in Python) runs directly on the sequence of primitives (the genotype) described in Section 4.2.2 instead of on P4 code (phenotype). When simulating a program, the Switch simulator first initializes a new list of registers, as described in Section 4.2.2. It then "runs" the program on each packet of the network trace by

1. Copying the packet attributes to the corresponding registers.
2. Interpreting the GP4P4 primitives line by line, reading and modifying the register values whenever required.
3. Copying the output packet attributes from the corresponding registers.

The fitness value of the program is then determined by counting the total number of satisfied output conditions, A_c, and dividing this value by the total number of output conditions, $N \times A_p$.

In the Switch Simulator, all primitives are assigned their own Python function. Consequently, to interpret a GP4P4 primitive, the simulator simply executes the corresponding Python function. If-then primitives form their own special case: when the simulator encounters an if-then primitive, it checks if its condition is true. If it is, the simulator continues to the next line. If not, the simulator searches for and skips forward to the corresponding `ENDIF()` primitive. To prevent the simulator from jumping to the end of a nested if-then block instead, it keeps track of its current depth while searching for the correct `ENDIF()` primitive.

4.2.4. End Condition

In this subsection, we describe several possible end conditions to terminate the inner and outer loops.

Inner Loop End Condition

The inner GP loop can be terminated on several conditions:

- **A correct program has been found.** This is the most straightforward condition. The loop stops whenever a program that satisfies all rules has been found.
- **Generation limit has been reached.** To abruptly terminate an inner loop that has been running for a long time, we can set a predefined limit of generations. If the limit has been reached, the loop exits without a correct program.
- **No significant improvement after certain generations.** Another possible end condition is to stop whenever we do not see any significant improvement in the maximum fitness value of the population for a certain number of generations.

The second and third end conditions produce an incorrect program that satisfies only some part of the rules. However they are needed in practice to terminate loops that run for an indefinite amount of time.

Outer Loop End Condition

Similar to the inner loop, the outer GP loop can be terminated on two conditions: when a correct program has been found *or* the *attempt limit* has been reached. An attempt is defined as one run through the outer loop.

Table 4.1: LGP Parameters.

Parameters	Values
Population size, N	3200
Inner loop iteration limit	3000
min_len	1
max_len	10
Crossover rate, P_c	1
Mutation rate, P_m	0.4
P_{if}	0.5
Tournament size ratio, t_r	0.05
Tournament losers, n_r	3
Trace generation multiplier, k	1

Table 4.2: Network Function Properties.

Function Name	rules	prims	ins	outs	cons	blen	bifd
NAT	4	IFEQ, ENDIF, ASSIGN	3	2	4	6	1
Firewall	4	IFNEQ, ENDIF, DROP	5	5	2	5	2
Server Balancer	2	IFEQ, ENDIF, ASSIGN	2	2	6	4	1
Link Balancer	2	IFEQ, ENDIF, ASSIGN	2	2	5	4	1
DSCP Marker	2	IFEQ, ENDIF, ASSIGN	5	5	4	4	1
PAT	4	IFEQ, ENDIF, ASSIGN	4	4	4	5	2
Router	2	IFEQ, ENDIF, ASSIGN, SUB	4	4	7	5	1

Header definitions. *rules*: number of rules, *prims*: primitives used as building blocks, *ins*: number of inputs, *outs*: number of outputs, *cons*: number of constants, *blen*: baseline (manually written solution) code length, *bifd*: baseline code maximum nested-IF-statement depth.

4.3. Experiments

We demonstrate GP4P4 on 7 small network functions: Network Address Translation (NAT), Firewall, Server Balancer, Link Balancer, DSCP Marker, Router, and Port Address Translation (PAT). Table 4.2 shows the properties of these network functions. The prototype of GP4P4 was implemented in Python language, while the experiments were run on an Intel Xeon CPU E5-2690 running Ubuntu 14.04.6 LTS (kernel version 3.13.0-151).

Unless explicitly mentioned, by default the experiments are run with the parameters indicated in Table 4.1. Further the inner loop is terminated when a generation limit is reached *or* whenever a correct program is found. As for the outer loop, it is terminated when the attempt limit reaches a threshold of 2000 attempts *or* a correct program is found.

4.3.1. Generation Time and Program Length in Various Network Functions

As can be seen in Figure 4.10, GP4P4 can generate each of the 7 network functions in a matter of minutes. Even for the most difficult function (Router), a valid solution is usually found within 100 seconds. The worst-case generation time we encountered was around 329 seconds. As network functions do not constantly need to be regenerated, this is well within acceptable limits. In fact, GP4P4 enables networks to almost immediately react to changing requirements from users or network operators, as the network can generate and install a completely new P4 program within minutes.

From Figure 4.11, we can observe that GP4P4 in general generates program with small variance between program lengths. Only the PAT network function has a relatively larger variance. One possible explanation is because the solution to PAT has a larger *nested-if* depths requirement to all other network function, except for Firewall. Interestingly, this phenomenon does not occur in Firewall, which has the same nested-if depth requirement as PAT. However, if we observe the generation time for Firewall, it becomes clear that Firewall usually finds a solution in a really short time (near zero), most likely because the solution is often found in the initial population. Thus, the Firewall does not even have to evolve any

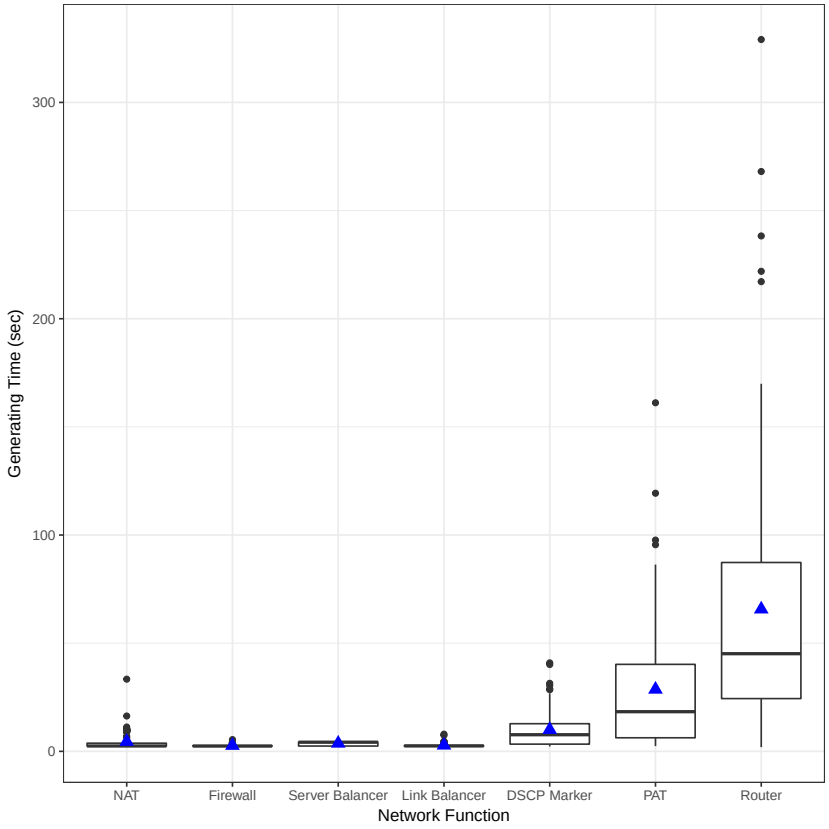


Figure 4.10: Tukey boxplot of the generation times of 7 network functions. The blue triangle shows the average generation time. Each network function was generated 100 times.

nested-if blocks like PAT. This can be further explained because the number of constants in the Firewall function is relatively low (2).

4.3.2. Various Parameter Effects on the Generation Time

Next, we consider the effect of changing different parameters on the program generation time. In general, there does not seem to be a clear-cut rule for the optimal setting for *all* network functions. However, in all our experiments, a program could still be generated within 15 minutes at worst, suggesting that it is still possible to achieve reasonable generation times even with non-optimal parameters.

Figure 4.12 and Figure 4.13 show the generation time of DSCP Marker and PAT functions versus the population size, generation limit, tournament size ratio, tournament losers, minimum initial program length, maximum initial program length, crossover rate, mutation rate and mutation IF-block rate. We chose to illustrate these 2 network functions because they have relatively high generation times, and thus presumably are more difficult to generate. We can observe that in both network function, the effect that each parameter does to the generation time is fairly similar.

For all network functions except NAT, a population size of around 1000 seems to be near-optimal. Going below 1000 increased generation times. Given that NAT is a relatively easy function to generate[2] and we want to prioritize the generation time of more difficult functions, this seems to be an optimal choice for the population size.

For the more difficult programs, a low tournament size ratio of at most 0.1 results in both lower generation times and generation time variance. A lower tournament size ratio allows more sub-optimal programs to evolve. Presumably, this helps increase the number of possibilities GP4P4 considers, which allows it to find valid programs more quickly.

[2]In fact, even with a population size of 1000 all NAT experiments finished within 3 minutes.

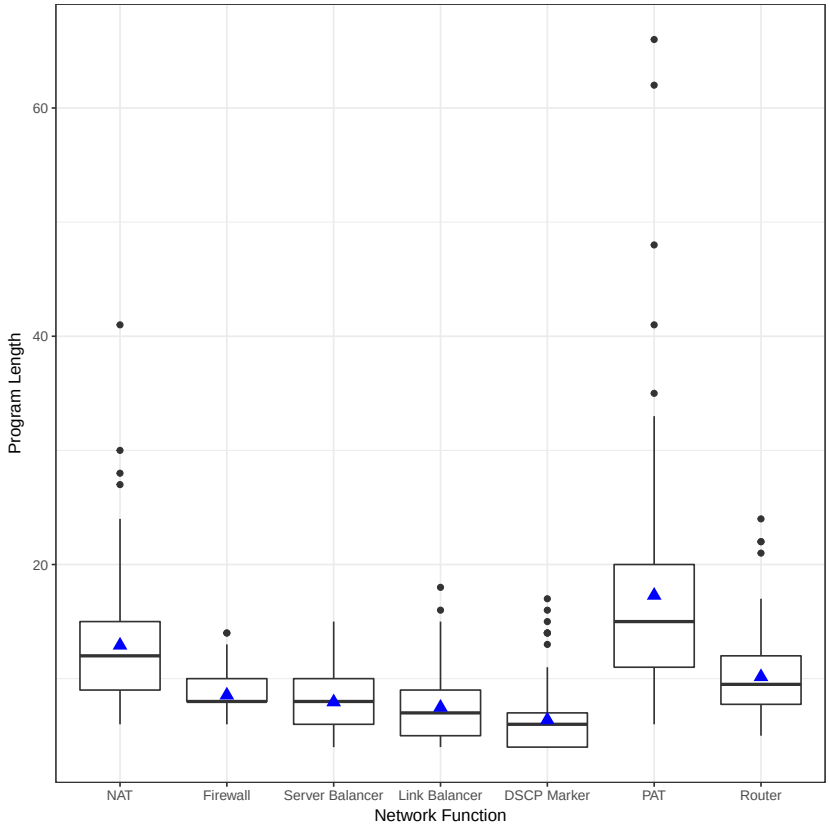


Figure 4.11: Tukey boxplot of the resulting program length of 7 network functions. The blue triangle shows the average program length. Each network function was generated 100 times.

Introducing crossover decreases the generation times of all network functions. In addition, increasing the crossover rate also seems to reduce variance. Increasing the crossover rate had a large impact on all generation times except those for PAT, which it reduced slightly. The optimal crossover rate seems to be around 0.9.

Changing the mutation rate only had a noticeable effect on the generation times of PAT and DSCP Marker. However, as it decreased both the average generation time and the variance between generation times of these functions, mutation is clearly worthwhile to include in GP4P4.

On the other hand, the size of generation limit, tournament losers, minimum initial program length, and the probability of introducing new if-blocks do not seem to give any significant impact on the generation time for both function, except for a slight variation in the variance between the generation times.

4.3.3. Various Parameter Effects on the Program Length

We now inspect the impact of changing different parameters on the resulting program length. In general, the size of generated programs are relatively more stable than the time needed to generate those program. Even while varying the maximum initial program length – the parameter that has most significant effect on the program lengths – the average program lengths do not seem to change much, while the variance between program lengths do change quite significantly.

Figure 4.14 and Figure 4.15 show the resulting program lengths of DSCP Marker and PAT functions versus the population size, generation limit, tournament size ratio, tournament losers, minimum initial program length, maximum initial program length, crossover rate, mutation rate and mutation IF-block rate. We chose to illustrate these 2 network functions for the same reason as in the previous subsection, with an additional reason to inspect the interesting behavior of the PAT, which was the only function with high variance between the

program lengths as depicted in Figure 4.11.

We can observe that most of the parameters do not seem to impact the program length significantly, thus we do not analyse the results we get for these parameters. The parameters with little to no effect to the program lengths are: population size, generation limit, tournament size ratio, size of tournament losers, mutation rate, and probability of introducing new if-blocks.

The minimum and maximum initial program lengths do seem to have the most significant effect on the program length. As can we logically expect, longer initial program length – either minimum or maximum – generates a longer program. As we aim for shorter, more effective, solutions, the most optimal value for these parameters is as low as possible, translated to 1 for the minimum initial length and 10 for maximum initial length.

Further, the crossover rate also seems to give some impact on the variance between the resulting program lengths. This is quite noticeable for the PAT function, but not so much in the DSCP marker function. From the PAT function, we can see that lower crossover rate gives programs with shorter length. Although the lowest value for the crossover rate seems to be best in this case, we must also remember that the highest crossover value gave us the shortest generation time. As minimizing the length of the resulting program is not first priority, the conclusion that we have for crossover rate from the generation section still holds true in this case.

4.4. Conclusion

The size and complexity of networks has grown formidably, making managing and programming them a daunting task. In this work we provide a first step towards automating this process by enabling self-programming networks. While the introduction of P4 has enabled network operators to construct high-speed customized network functions, this has come at a significant cost: Network operators now have to create and maintain a large repository of network functions, which introduces a completely new vector for network failures. However, this does not need to be the case. We have proposed GP4P4, a framework for automatically generating data-plane code satisfying sets of simple behavioral rules. Our proposed framework, called GP4P4, uses Linear Genetic Programming techniques to automatically evolve a population of P4 network programs towards satisfying a given rule-set. GP4P4 evaluates these programs by simulating a P4 switch and generating a synthetic trace of network packets tailored towards effectively evaluating a specific rule-set. This not only reduces the computation time significantly, but also allows GP4P4 to generate P4 programs without relying on any external switches or network traces.

Our experiments show that GP4P4 can generate P4 programs within minutes. This enables networks to quickly react to changing requirements, as networks can now generate and install completely new P4 programs quickly.

Although GP4P4 is currently tested with simple behavioral rules, we believe it is an important first step towards a future of *self-programming* networks: networks that can fully program and adapt themselves to their current goals and circumstances with minimal intervention by network operators.

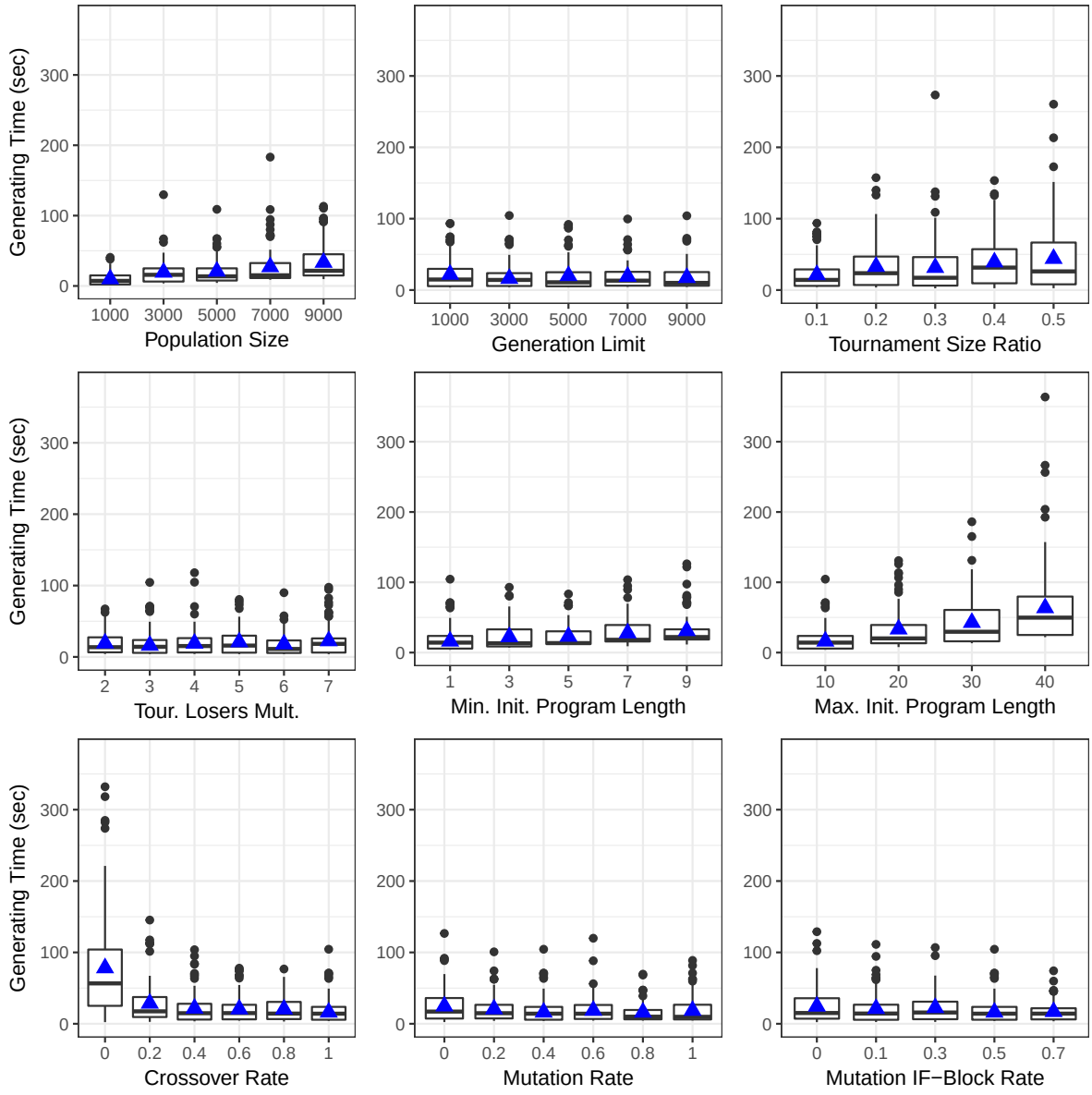


Figure 4.12: Tukey boxplots of the generation times of DSCP Marker function versus the population size, generation limit, tournament size ratio, tournament losers, minimum initial program length, maximum initial program length, crossover rate, mutation rate and mutation if-block rate. The blue triangle shows the average generation time. All experiments were repeated 100 times.

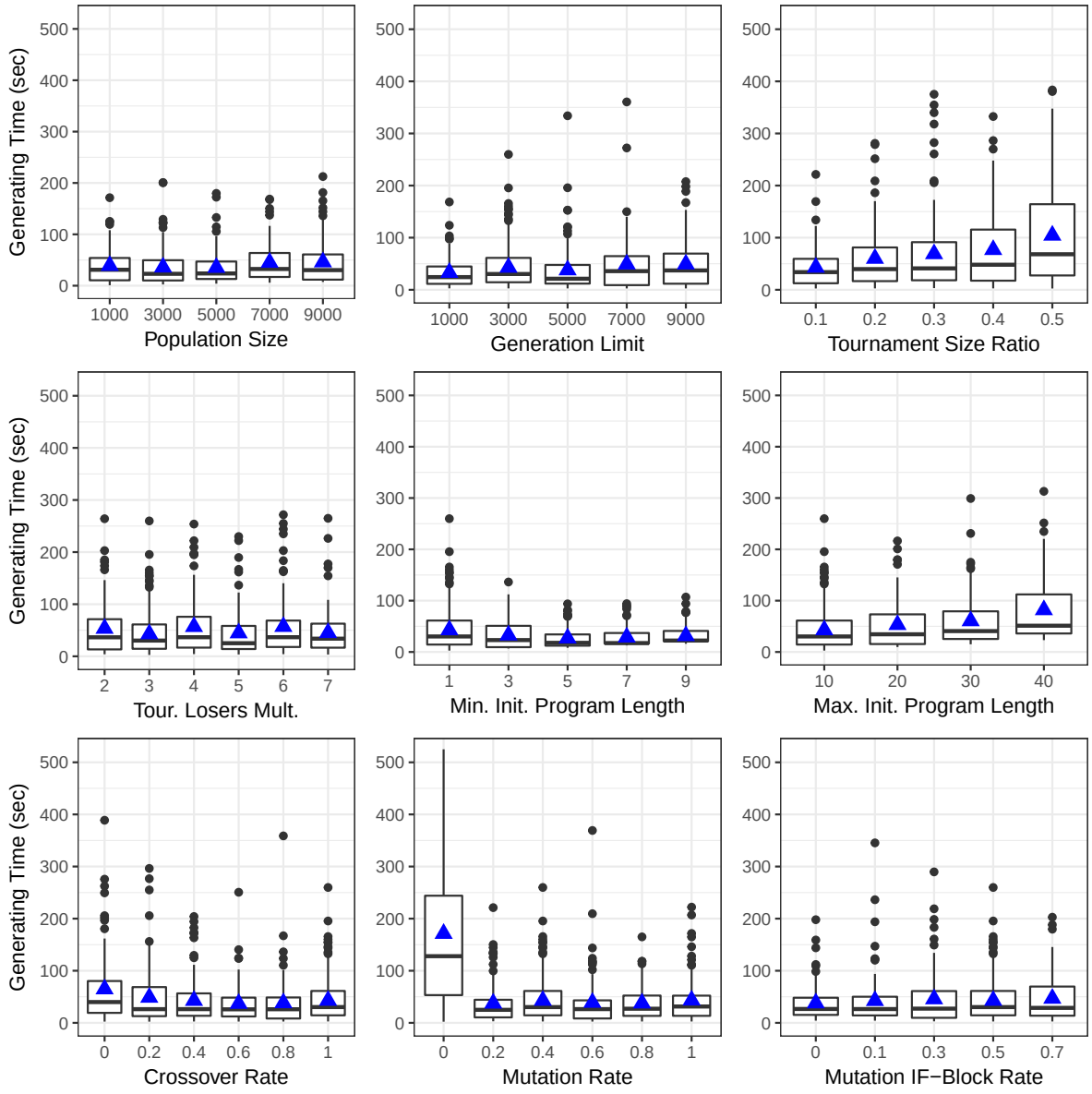


Figure 4.13: Tukey boxplots of the generation times of PAT function versus the population size, generation limit, tournament size ratio, tournament losers, minimum initial program length, maximum initial program length, crossover rate, mutation rate and mutation if-block rate. The blue triangle shows the average generation time. All experiments were repeated 100 times.

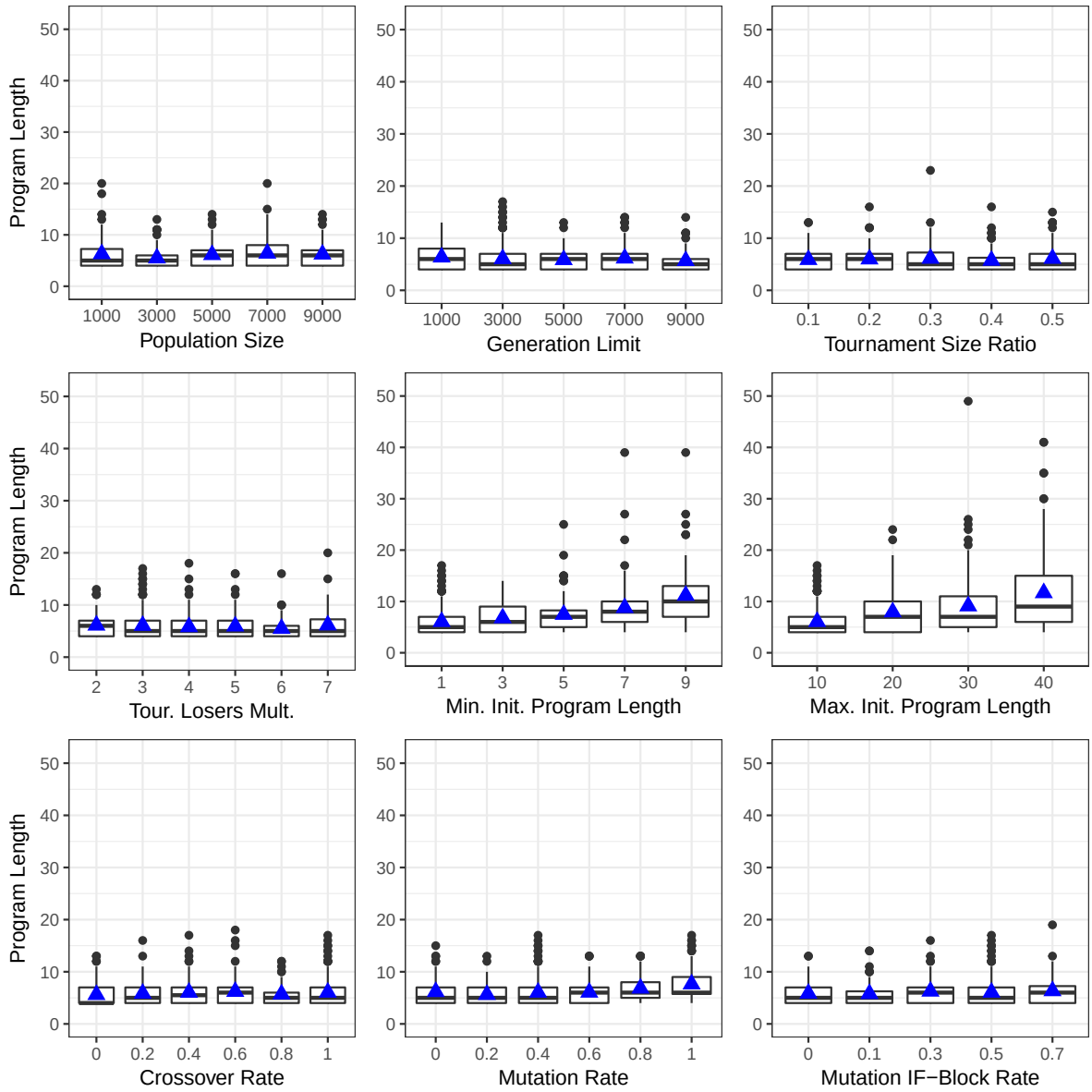


Figure 4.14: Tukey boxplots of the resulting program length of DSCP Marker function versus the population size, generation limit, tournament size ratio, tournament losers, minimum initial program length, maximum initial program length, crossover rate, mutation rate and mutation if-block rate. The blue triangle shows the average program length. All experiments were repeated 100 times.

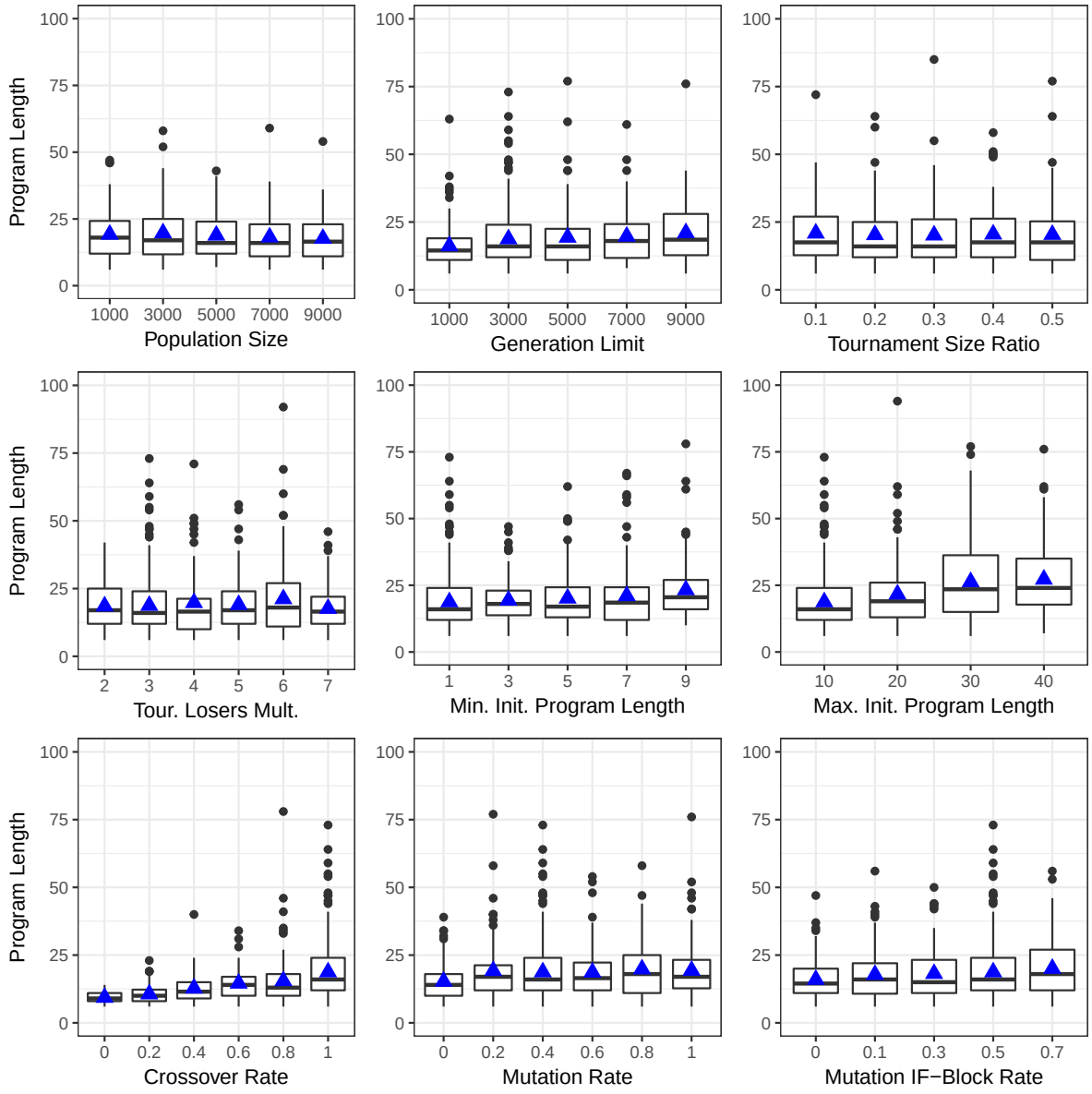


Figure 4.15: Tukey boxplots of the resulting program length of PAT function versus the population size, generation limit, tournament size ratio, tournament losers, minimum initial program length, maximum initial program length, crossover rate, mutation rate and mutation if-block rate. The blue triangle shows the average program length. All experiments were repeated 100 times.

5

Conclusion

This chapter concludes the work that has been done within this report. We begin by answering the research questions from Chapter 1, stating the contributions that we have made in P4I/O and GP4P4 frameworks, before finally closing it with mentioning some possible future research directions.

5.1. Answer to the Research Question

The research questions posed in Chapter 1 can be answered as follows:

- **RQ1. In what language can we express our network intent?** The answer to this question is twofold: we can express our intent in a high-level close-to-English style as demonstrated in the P4I/O platform or the lower-level behavioral rules as demonstrated in the GP4P4 platform. We based the former on the Nile language [31], which already satisfies all our objectives for an easy-to-learn and intuitive intent language, while the latter is specifically developed to guide our adaptation of Genetic Programming technique for "evolving" P4 programs.
- **RQ2. How to design and implement a system that can generate a P4 program based on user intent?** Our first framework, P4I/O searches for a program using a straightforward graph union operation and then generates it using parameterized templates. Our second framework, GP4P4 aims to fix the static characteristic intrinsic to the code templates by using a more intelligent technique of Linear Genetic Programming (LGP). The LGP searches for a solution from a hypothesis domain of P4 language primitives, packet attributes and constants derived from the user-supplied *network behavioral rules*.
- **RQ3. How is the efficacy of such a system?** We have shown by a Proof-of-Concept demonstration that the generation of a working P4 program in our first framework of P4I/O takes a really short time, even on non-server-grade hardware. P4I/O was also able to push the code into the BMV2 software switch in a short time, enabling us to change intent even on-the-fly. The second framework of GP4P4 was able to generate P4 code in a relatively short time with a maximum generation time of around 5 minutes even for the most complex network function in our test cases. We consider this generation time acceptable as the behavioral rules are not often revised by the users so that the framework can even generate the code proactively – before the user requests for it.

5.2. Contribution

We have presented the following key contributions in the P4I/O framework:

1. **Extensible Intent Definition Language (IDL).** To describe various kinds of network services as intents, we devise a high-level language that is close to the human language, yet precise enough to be interpreted unambiguously by the network controller.

Furthermore, this language is extensible so that we can define any kind of data-plane functionality.

2. **Template-Based P4 Code Generation.** We constructed a repository of relevant network functions in the form of P4 code templates. These templates are then parsed and represented in a specialized data structure that facilitates combining the network functions, following the intent instructions. The code templates are then finally merged to form a valid P4 program.

3. **Dynamic Intents Realization.** We provided a technique to install the resulting P4 code in a programmable switch while permitting intent modification at any time. We realize intent modifications, with minimal disruption to the traffic forwarding process, through a state-transfer mechanism.

4. **Framework Evaluation.** We demonstrated that P4I/O works, by building a proof-of-concept. P4I/O code has been released as open-source code [55].

Besides, the following contributions were presented in the GP4P4 framework:

1. **GP4P4**. We presented GP4P4, a framework for automatically generating P4 programs using techniques adapted from Linear Genetic Programming (LGP). LGP is a machine-learning technique to "evolve" an initially randomized population of programs towards satisfying an objective function [11].

2. **An evaluation module to make LGP suitable for data-plane programmability.** We proposed an evaluation module that evaluates programs by creating synthetic network traces and simulating the output of P4 programs on these traces. In this regard, GP4P4 is fully self-sufficient and does not depend on any external network traces or physical switches.

3. **Proof-of-Concept experiments demonstrating the efficacy of GP4P4.** The experiments demonstrated that GP4P4 was able to generate P4 codes based on rules within a few minutes.

5.3. Future Work

For future work, we have identified the following potential research directions:

1. **Enhancing the behavioral rules for GP4P4 with more advanced operators.** The current behavioral rule is quite limited in the sense it only supports the `IF-THEN` construct and also only supports conjunctive (`AND`) logical operator between the conditions. It would be interesting to add more logical constructs, like `ALL` and disjunctive (`OR`) logical operator, to build more complex network functions.

2. **"Mining" network behavioral rules from external sources.** Currently, the behavioral rules are supplied by the network operators. This can limit the usefulness of the GP4P4 framework as the network function is only going to be as good as how the rules are defined. Further, writing behavioral rules for network functions requires deep knowledge of the behavior of the intended function. To this end, we propose to search for a way to obtain these rules automatically from an external source. An example approach for this problem would be mining behavioral rules from the corpus of network device documentation on the World Wide Web.

Bibliography

[1] Ieee standard for local and metropolitan area networks: Media access control (mac) bridges. *IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998)*, pages 1–281, June 2004. doi: 10.1109/IEEESTD.2004.94569.

[2] Donald E. Eastlake 3rd and Anoop Ghanwani. Transparent Interconnection of Lots of Links (TRILL) Support of the Link Layer Discover Protocol (LLDP). Internet-Draft draft-eastlake-trill-lldp-01, Internet Engineering Task Force, September 2012. URL `https://datatracker.ietf.org/doc/html/draft-eastlake-trill-lldp-01`. Work in Progress.

[3] Anubhavnidhi Abhashkumar, Joon-Myung Kang, Sujata Banerjee, Aditya Akella, Ying Zhang, and Wenfei Wu. Supporting Diverse Dynamic Intent-based Policies using Janus. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies - CoNEXT '17*, number 1, pages 296–309, New York, New York, USA, 2017. ACM Press. ISBN 9781450354226. doi: 10.1145/3143361.3143380. URL `http://dl.acm.org/citation.cfm?doid=3143361.3143380`.

[4] A. Adams, J. Nicholas, and W. Siadak. Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised). RFC 3973 (Experimental), January 2005. URL `http://www.ietf.org/rfc/rfc3973.txt`.

[5] Saeed Arezoumand, Kristina Dzeparoska, Hadi Bannazadeh, and Alberto Leon-garcia. MD-IDN : Multi-Domain Intent-Driven Networking in Software-Defined Infrastructures. 2017.

[6] Barefoot Network. Behavioral model repository. `https://github.com/p4lang/behavioral-model`, 2019. [Online; accessed 23-January-2019].

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486011. URL `http://doi.acm.org/10.1145/2486001.2486011`.

[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. ISSN 0146-4833. doi: 10.1145/2656877.2656890. URL `http://doi.acm.org/10.1145/2656877.2656890`.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[10] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):16, 2018.

[11] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 1441940480, 9781441940483.

[12] Brad Cain, Steve Deering, Isidor Kouvelas, Bill Fenner, and Ajit S. Thyagarajan. Internet group management protocol, version 3. *RFC*, 3376:1–53, October 2002. URL http://www.ietf.org/rfc/rfc3376.txt.

[13] Walter Cerroni, Chiara Buratti, Simone Cerboni, Gianluca Davoli, Chiara Contoli, Francesco Foresta, Franco Callegati, and Roberto Verdone. Intent-based management and orchestration of heterogeneous openflow/IoT SDN domains. *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*, 2017. doi: 10.1109/NETSOFT.2017.8004109.

[14] Cisco. Intent-based networking: Building the bridge between business and it, 2018. URL https://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/digital-network-architecture/nb-09-intent-networking-wp-cte-en.pdf. Online; accessed 1 November 2018.

[15] Rami Cohen, Katherine Barabash, Benny Rochwerger, Robert Birke, and Renato Recio. An Intent-based Approach for Network Virtualization. pages 42–50, 2013.

[16] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005. ISSN 0196-6774. doi: 10.1016/j.jalgor.2003.12.001. URL http://dx.doi.org/10.1016/j.jalgor.2003.12.001.

[17] Robert Craven, Jorge Lobo, and Emil Lupu. Policy refinement: decomposition and operationalization for dynamic domains. *International Conference on Network and Service Management, CNSM*, (October 2015):1–9, 2011. doi: 10.1109/CNSM.2010.5691331.

[18] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3451-8. doi: 10.1145/2774993.2774999. URL http://doi.acm.org/10.1145/2774993.2774999.

[19] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629578. URL http://doi.acm.org/10.1145/1629575.1629578.

[20] Sean Donovan and Nick Feamster. Intentional network monitoring: Finding the needle without capturing the haystack. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 5:1–5:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3256-9. doi: 10.1145/2670518.2673872. URL http://doi.acm.org/10.1145/2670518.2673872.

[21] Cristian Estan, George Varghese, and Michael Fisk. Bitmap algorithms for counting active flows on high-speed links. *IEEE/ACM Trans. Netw.*, 14(5):925–937, October 2006. ISSN 1063-6692. doi: 10.1109/TNET.2006.882836. URL http://dx.doi.org/10.1109/TNET.2006.882836.

[22] Nick Feamster and Jennifer Rexford. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583*, 2017.

[23] Y. Geng, S. Liu, F. Wang, Z. Yin, B. Prabhakar, and M. Rosenblum. Self-programming networks: Architecture and algorithms. In *2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 745–752, Oct 2017. doi: 10.1109/ALLERTON.2017.8262813.

[24] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Simon: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019.

[25] Brighten Godfrey. Verification and intent-based networking: Closing the control loop, 2017. URL https://www.veriflow.net/verification-intent-based-networking-closing-control-loop/. Online; accessed 1 November 2018.

[26] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010.

[27] Arpit Gupta, Rob Harrison, Ankita Pawar, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Network Telemetry. 2017. doi: 10.1109/TBME.2002.804593. URL http://arxiv.org/abs/1705.01049.

[28] Yoonseon Han, Jian Li, Doan Hoang, Jae-hyoung Yoo, and James Won-ki Hong. An Intent-based Network Virtualization Platform for SDN. 2016.

[29] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

[30] Google Inc. Dialogflow, 2018. URL https://dialogflow.com/. Online; accessed 1 November 2018.

[31] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks - SelfDN 2018*, pages 15–21, New York, New York, USA, 2018. ACM Press. ISBN 9781450359146. doi: 10.1145/3229584.3229590. URL http://dl.acm.org/citation.cfm?doid=3229584.3229590.

[32] Juniper Networks. The self-driving network, March 2017. white paper.

[33] Patrick Kalmbach, Johannes Zerwas, Peter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. Empowering self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, pages 8–14. ACM, 2018.

[34] Dave Katz and Dave Ward. Bidirectional forwarding detection (bfd). *RFC*, pages 1–49, June 2010. URL https://tools.ietf.org/html/rfc5880.

[35] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. Adaptable and data-driven softwarized networks: Review, opportunities, and challenges. *Proceedings of the IEEE*, 107(4):711–731, 2019.

[36] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000. ISSN 0734-2071. doi: 10.1145/354871.354874. URL http://doi.acm.org/10.1145/354871.354874.

[37] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, pages 768–774, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1623755.1623877.

[38] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, Jun 1994. ISSN 1573-1375. doi: 10.1007/BF00175355. URL https://doi.org/10.1007/BF00175355.

[39] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 830–838, Piscataway, NJ, USA, 2010. IEEE Press. ISBN 978-1-4244-5836-3. URL http://dl.acm.org/citation.cfm?id=1833515.1833654.

[40] Diego Kreutz, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):63, 2015. URL http://arxiv.org/abs/1406.0440.

[41] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0409-2. doi: 10.1145/1868447.1868466. URL http://doi.acm.org/10.1145/1868447.1868466.

[42] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, Vladimir Braverman Johns, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. *Sigcomm*, (Question 24):101–114, 2016. doi: 10.1145/2934872.2934906. URL http://doi.acm.org/10.1145/2934872.2934906.

[43] Antonio Marsico, Michele Santuari, Marco Savi, Domenico Siracusa, Abdul Ghafoor, Stephane Junique, and Pontus Skoldstrom. An interactive intent-based negotiation scheme for application-centric networks. *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*, 2017. ISSN 20490801. doi: 10.1109/NETSOFT.2017.8004251.

[44] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355746. URL http://doi.acm.org/10.1145/1355734.1355746.

[45] Julian F. Miller. *Cartesian Genetic Programming*, pages 17–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-17310-3. doi: 10.1007/978-3-642-17310-3_2. URL https://doi.org/10.1007/978-3-642-17310-3_2.

[46] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482629.

[47] J. Moy. OSPF Version 2. RFC 2328 (Standard), April 1998. URL http://www.ietf.org/rfc/rfc2328.txt.

[48] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*, pages 85–98, 2017. ISBN 9781450346535. doi: 10.1145/3098822.3098829. URL http://web.mit.edu/marple/marple-sigcomm17.pdfhttp://dl.acm.org/citation.cfm?doid=3098822.3098829.

[49] NeMo. Nemo: An application's interface to intent based networks, 2016. URL http://www.nemo-project.net/. Online; accessed 1 November 2018.

[50] D. Oran. OSI IS-IS Intra-domain Routing Protocol. RFC 1142 (Informational), February 1990. URL http://www.ietf.org/rfc/rfc1142.txt.

[51] P4.org. P4 language tutorial, 2018. URL http://bit.ly/p4d2-2018-spring.

[52] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218. URL http://dl.acm.org/citation.cfm?id=2789770.2789779.

[53] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787506. URL http://doi.acm.org/10.1145/2785956.2787506.

[54] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. URL http://www.ietf.org/rfc/rfc4271.txt.

[55] Mohammad Riftadi and Fernando Kuipers. riftadi/p4io: Intent-based p4 code generation framework. https://github.com/riftadi/p4io, 2019. [Online].

[56] Mohammad Riftadi and Fernando A. Kuipers. P4I/O: Intent-Based networking with P4. In *2019 2nd International Workshop on Emerging Trends in Softwarized Networks (ETSN 2019 at NetSoft)*, Paris, France, June 2019.

[57] Armin Ronacher. Jinja2. http://jinja.pocoo.org/, 2019. [Online; accessed 11-January-2019].

[58] Davide Sanvito, Daniele Moro, Ilario Filippini, Antonio Capone, and Andrea Campanella. ONOS Intent Monitor and Reroute service: enabling plug&play routing logic. *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, (NetSoft):272–276, jun 2018. doi: 10.1109/NETSOFT.2018.8460064. URL https://ieeexplore.ieee.org/document/8460042/.

[59] Eder J. Scheid, Cristian C. MacHado, Muriel F. Franco, Ricardo L. Dos Santos, Ricardo P. Pfitscher, Alberto E. Schaeffer-Filho, and Lisandro Z. Granville. INSpIRE: Integrated NFV-based Intent Refinement Environment. *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management*, pages 186–194, 2017. doi: 10.23919/INM.2017.7987279.

[60] Thomas Szyrkowiec, Michele Santuari, Mohit Chamania, Domenico Siracusa, Achim Autenrieth, Victor Lopez, Joo Cho, and Wolfgang Kellerer. Automatic Intent-Based Secure Service Creation Through a Multilayer SDN Network Orchestration. *Journal of Optical Communications and Networking*, 10(4):289, 2018. ISSN 0888-8892, 1523-1739. doi: 10.1111/j.1523-1739.2011.01777.x. URL https://www.osapublishing.org/abstract.cfm?URI=jocn-10-4-289.

[61] The P4 Language Consortium. $P4_{16}$ language specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, 2017. [Online; accessed 13-January-2019].

[62] The P4 Language Consortium. $P4_{14}$ language specification. https://p4lang.github.io/p4-spec/p4-14/v1.0.5/tex/p4.pdf, 2018. [Online; accessed 16-May-2019].

[63] The P4.org API Working Group. P4runtime specification. https://s3-us-west-2.amazonaws.com/p4runtime/docs/v1.0.0-rc4/P4Runtime-Spec.html, 2018. [Online; accessed 13-December-2018].

[64] Yoshiharu Tsuzaki. Reactive Configuration Updating for Intent-Based Networking. pages 97–102, 2017.

[65] Belma Turkovic, Fernando Kuipers, Niels van Adrichem, and Koen Langendoen. Fast network congestion detection and avoidance using p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, NEAT '18, pages 45–51, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5907-8. doi: 10.1145/3229574.3229581. URL http://doi.acm.org/10.1145/3229574.3229581.

[66] D. C. Verma. Simplifying network administration using policy-based management. *IEEE Network*, 16(2):20–26, March 2002. ISSN 0890-8044. doi: 10.1109/65.993219.

[67] Junfeng Xie, F Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, Chenmeng Wang, and Yunjie Liu. A survey of machine learning techniques applied to software defined networking (sdn): Research issues and challenges. *IEEE Communications Surveys & Tutorials*, 21(1):393–430, 2018.

[68] Tong Yang, Lun Wang, Yulong Shen, Muhammad Shahzad, Qun Huang, Xiaohong Jiang, Kun Tan, and Xiaoming Li. Empowering sketches with machine learning for network measurements. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 15–20. ACM, 2018.

[69] Touseef Yaqoob, Muhammad Usama, Junaid Qadir, and Gareth Tyson. On analyzing self-driving networks: A systems thinking approach. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, pages 1–7. ACM, 2018.

[70] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. *Proceedings 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 13:29–42, 2013. ISSN 1470-0328.