UniCorn-P4: A Universal Control Plane and GUI for P4

Fabian Ihle*, Moritz Flüchter*, Steffen Lindner*, Michael Menth*
*University of Tübingen, Chair of Communication Networks
{fabian.ihle, moritz.fluechter, steffen.lindner, menth}@uni-tuebingen.de

***Abstract*—Traditional networking equipment is limited to the features and management tools supplied by the manufacturer. Software-Defined Networking (SDN) is an approach to loosen this dependency on manufacturers by allowing developers to implement custom control planes. Data plane programming further evolves this concept, enabling developers to fully customize the forwarding logic on network devices. Currently, Programming Protocol-independent Packet Processors (P4) is the prominent technology in academia and industry for data plane programming. However, the development process of new P4 prototypes is difficult and slow. Both the data and control plane must be developed simultaneously, require a complex tech stack, and leverage new programming languages.**

In this work, we introduce UniCorn-P4, a universal control plane and GUI for P4. UniCorn-P4 supports the development of P4 prototypes by providing a control plane that is compatible with any P4 program. Developers can focus on implementing P4 data planes and use UniCorn-P4 to validate their implementation. UniCorn-P4 connects to P4 switches, automatically detects the available data plane entities, and enables user configuration over the graphical user interface. Further, UniCorn-P4 can emulate virtual testbeds of network topologies using the Mininet framework for rapid prototyping.

***Index Terms*—Data Plane Programming, P4, Rapid Prototyping**

I. Introduction

Traditional network devices are delivered as a closed system with hardware and software bundled together. Operators can configure features through a provided interface, such as a web GUI or the SNMP protocol. However, the configuration is limited to the functionality exposed by the vendor. With SDN, a custom control plane can be implemented. The resulting flexibility is the reason why SDN is widely used in practice, such as in data centers [1] and in 6G networks [2]. However, a SDN network device is still limited by the capabilities of the data plane. With data plane programming, the forwarding algorithms and data structures can be implemented directly in the data plane. These algorithms can be tailored to the needs of the network and allow for full customization of packet forwarding logic, limited only by the hardware and the expressiveness of the programming language.

Currently, Programming Protocol-independent Packet Processors (P4) [3] is widely used in industry and academia for data plane programming [4]. It comprises a programming language, architecture abstraction, and a data plane API. P4 facilitates rapid prototyping of new protocols and is therefore used in research on next-generation technologies such as 6G [5]–[7]. However, developing and especially testing new prototypes with P4 is a time-consuming and complex process. Developers have to implement a data plane program, then a control plane application, and finally a testing environment before they can validate their program. Further, developers must be familiar with the P4 data plane programming language and the API to the data plane.

We developed UniCorn-P4 [8], a universal control plane and GUI for faster prototyping. Developers can connect UniCorn-P4 to multiple hardware or virtual switches and then load P4 programs onto them. UniCorn-P4 creates a controller instance for each switch and discovers the available data plane entities by parsing the compiled P4 program. Then, users can use the GUI to manipulate table entries in the data plane or load entries from a file. This allows them to test and validate their P4 prototypes without writing a custom control plane. Developers can either provide their own testbed environment in hardware or software or use the Mininet [9] extension for UniCorn-P4. The Mininet extension enables rapid prototyping with auto-generated virtual network devices. As a result, UniCorn-P4 streamlines the development process of P4 programs and can be leveraged as a prototyping environment.

II. Background

In this section, we provide background information on the programming language P4 including the basic concept of architectures in P4 and developing data plane programs for P4 switches. Further, we explain how data plane APIs such as the P4 runtime manage the runtime control of the data plane.

A. The Programming Language P4

P4 [3] is a domain-specific programming language to describe the data plane of P4-programmable switches, the so-called targets. A target can either be software-based, like the BMv2 [10] switch or hardware-based, like the Intel Tofino switching ASIC [11]. Each target implements a specific architecture, such as the Portable Switch Architecture (PSA) [12], the Tofino Native Architecture (TNA) [11], or the v1model architecture. The BMv2 implements the `simple_switch_grpc` architecture derived from the v1model architecture.

P4 uses a two-layer compiler model. First, a front-end compiler performs syntactic and target-independent semantic analysis. It then compiles the P4 program into an intermediate representation, e.g., into a JSON object. Second, a target-specific compiler performs transformations and maps the intermediate representation onto the target. This allows data plane algorithms to be developed in a common language while supporting different hardware- and software-based targets.

The P4 language can be used to implement custom algorithms that manipulate and forward packets. A P4 program provides a programmable packet parser, multiple Match-Action Units (MAUs), and a programmable packet deparser. After parsing the packet, the program logic is applied in one or more MAUs. MAUs consist of one or more Match-Action Tables (MATs) that can use logical expressions, simple arithmetic operations, and branching constructs. The concept of a MAT is illustrated in Figure 1.

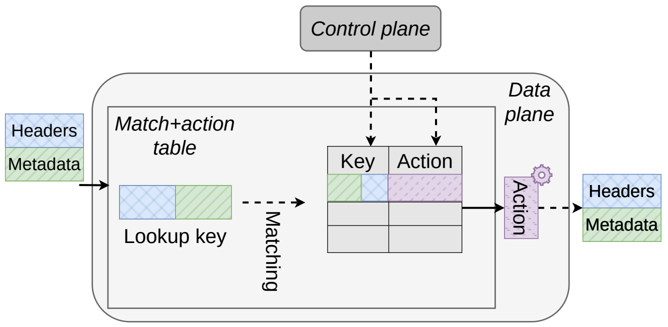

Fig. 1. A packet is matched according to a composite key of header fields and metadata to an entry in the table. The associated action on a table hit is executed. The entries in each MAT are populated by the control plane.

A MAT consists of predefined key fields that are matched for a packet. The key of a MAT consists of packet header fields or metadata. An associated action is performed when a key is matched against an entry in the MAT. An action can manipulate packet data or make a packet forwarding decision and is defined by the programmer. While the MATs in the data plane define the table structure, i.e., the matching key, and the available actions, the content of those tables is populated by the control plane. More information on P4 can be found in an extensive survey by Hauser et al. [4].

B. Runtime Control of the Data Plane

The control plane implements the runtime control of a switch, such as adding, deleting, or modifying MAT entries. To that end, the data plane must expose endpoints that allow runtime control of data plane entities from the control plane. This interface is called the data plane API.

The P4 Runtime is a data plane API that is independent of the P4 program and the used target. The control plane needs to know about the existing MATs and its structure. For this purpose, the control plane communicates with the gRPC[1] protocol with the data plane API. The control plane can query the information from the target if the target has a program loaded. Otherwise, the control plane can load the p4info file and push a P4 program to the target. The p4info file is a file generated by the P4 compiler that contains all accessible P4 entities, such as MATs. Multiple libraries implement the P4 Runtime in common programming languages such as Python [14] or Go [15]. Other data plane APIs exist, such as the Barefoot Runtime [16], [17].

[1]gRPC is a modern, high performance, open source Remote Procedure Call (RPC) framework that can run in any environment [13].

III. UniCorn-P4: A Universal Control Plane and GUI for P4

In this section, we introduce UniCorn-P4, a generic graphical user interface (GUI) control plane for P4 programs. We first explain the features of UniCorn-P4 and the resulting development workflow. Then, we illustrate its architecture and APIs. Last, we introduce the UniCorn-P4 rapid prototyping extension for Mininet.

A. Development Workflow with UniCorn-P4

We first elaborate on the standard development process of P4 prototypes illustrated in Figure 2 and then explain how UniCorn-P4 can streamline this workflow.

The developer starts by writing the P4 program ❶ and compiles it for the target switch ❷. Then, the developer uses the knowledge of the existing MATs to design the corresponding controller ❸. For the validation, the developer has to set up a virtual or physical testbed that contains the target switch ❹. In the testbed, the P4 program is loaded onto the switches ❺ that are connected to the controller. The controller then writes the MAT entries to the switches ❻. After these steps, packets can be sent through the switch to validate the P4 program.

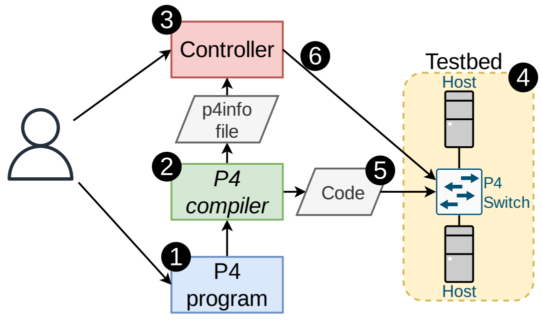

Fig. 2. The development and validation workflow of a P4 prototype. First, the P4 program is developed and compiled. Then, the controller can be developed using knowledge about the existing MATs. Last, the compiled P4 program is loaded onto switches in a virtual or physical testbed which are connected to the controller.

UniCorn-P4 simplifies this development process by providing a universal controller with a GUI for prototyping and validation (❸, ❺ and ❻). Developers can connect switches by supplying the network address of the switch, the gRPC port, and its device ID. Then, UniCorn-P4 can be used to load a P4 program onto these switches. UniCorn-P4 creates a controller instance for each switch and parses the P4 Runtime p4info file. This way, the controller instance automatically identifies the available MATs, actions, and data fields. Once a switch is initialized, developers can view and modify the MATs entries, or load them from a configuration file. Further, switch configurations and MAT entries are stored in a database, enabling users to restore snapshots of configured switches, programs, and table entries across multiple sessions from their history.

B. Architecture

The architecture of UniCorn-P4 is split into two main components: frontend and backend. Both components come in a dockerized environment for ease of use. Figure 3 illustrates this architecture and its communication interfaces. The frontend provides the user interface for configuring P4 switches and visualizes the configured switches including the content of their MATs. Configuration actions by the user are passed to the backend over an HTTP REST API. The backend contains the main controller which manages the connected switches and configures them over the P4 Runtime interface.

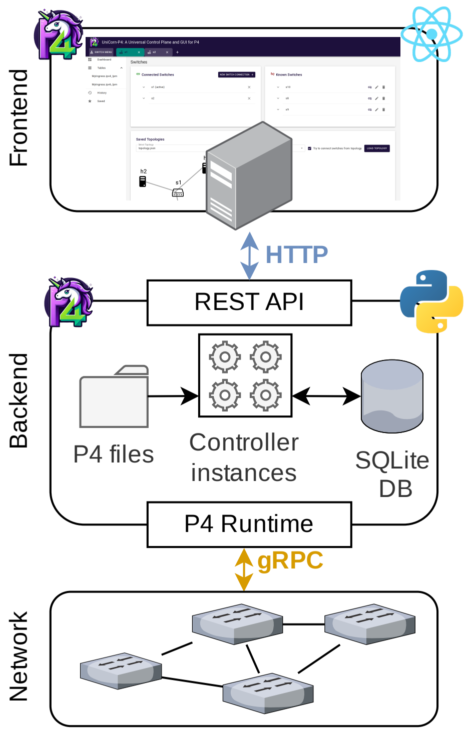

Fig. 3. The architecture of UniCorn-P4. The backend parses the P4 files to discover the MAT structure. This information is then used to connect to the P4 switches over the P4 Runtime interface. Further, the configuration for known switches is stored in an SQLite database. Changes to the MATs performed by the user in the frontend are sent to the backend over a REST API.

1) Frontend: The frontend of UniCorn-P4 is developed using React [18], ensuring a responsive interface. Information about switches and their MATs is dynamically retrieved from the backend in a generalized format. This allows the frontend to visualize the information without needing additional knowledge about specific P4 programs. Further, users can interact with the MAT entries without having to manually define the table format. The frontend also uses this information to provide a template for adding or changing table entries. It generates a form for the user to fill with all available match fields and possible actions. The information from this form is then passed back to the backend to modify the selected MAT.

2) Backend: The core of the backend consists of multiple controller instances that each implement the control plane for a target switch. Each instance manages an active switch connection over the P4 Runtime interface. For each switch connection, the controller parses the corresponding p4info file to discover the available data plane entities. The REST API of the backend exposes this information to the frontend and can also receive calls from the frontend, e.g., to write a new MAT entry to a switch. Data that needs to be stored over multiple sessions is written to and read from an SQLite database.

C. Mininet Extension

To simplify the prototyping of networks, UniCorn-P4 provides an extension for the SDN emulator Mininet [9] that enables testbed emulation (step ❹ in Figure 2). Mininet allows for rapid prototyping of networks with software-based targets emulated on constrained resources, such as a local workstation. The emulation framework leverages OS-level virtualization to build emulated networks in a specified topology. The topology can specify switches, hosts, and links between these nodes. Mininet then creates processes and network namespaces to emulate the network. In combination with the BMv2 [10] software-based P4 target, the emulated nodes can be loaded with a P4 program to emulate a SDN network environment. Each node can be individually programmed and configured to facilitate a SDN-based network without the need for expensive hardware.

With UniCorn-P4, developers can define network topologies consisting of hosts, switches, and links in a JSON format and load them via the web interface of UniCorn-P4. UniCorn-P4 creates the emulated network in a docker container using Mininet, BMv2 [10] switches, and Linux hosts. Then, developers can load P4 programs onto the switches and configure them over UniCorn-P4.

IV. Discussion

UniCorn-P4 leverages the P4 Runtime to communicate with and configure P4 switches, making it compatible with any target that implements the P4 Runtime API. Examples are the commonly used hardware targets Intel Tofino 1 and 2 [11], or the Mellanox Spectrum switch [19]. However, manufacturers also provide their own data plane APIs for advanced features such as architecture-specific externs. For example, the Intel Tofino uses the so-called Barefoot Runtime [16] which extends the P4 Runtime. Therefore, the current version of UniCorn-P4 cannot configure these advanced features. Future work should focus on adding support for other well-known data plane APIs to support a larger scope of targets.

Another shortcoming of UniCorn-P4 is that it currently only supports the configuration of MATs defined in a P4 program. However, there are other P4 entities that UniCorn-P4 cannot interact with, such as registers or counters. Info on these so-called externs is not exposed in the p4info file as they are part of the architecture and not the P4 program. Therefore, they are currently not configurable by UniCorn-P4. Configuration support for these externs may be added to UniCorn-P4 in future work.

V. Conclusion

In this work, we presented UniCorn-P4 a universal P4 controller for rapid prototyping of P4 programs in data plane programming. UniCorn-P4 simplifies the development process of

P4 programs by providing a web-based GUI for configuration and visualization of P4 switches. The backend of UniCorn-P4 implements a control plane that communicates with connected switches via the data plane API and derives the available MATs and their structure from P4 programs. Developers can manipulate the MAT entries manually or load them from configuration files. Switch configurations and MAT entries are stored in a database and can be restored over multiple sessions. With the Mininet extension for UniCorn-P4, network topologies can be emulated for testing P4 programs. However, UniCorn-P4 has some limitations as the user manually triggers changes to the MATs. It cannot automatically react to changes, e.g., like a routing protocol would react to changes in the topology.

Future work should extend the feature set of UniCorn-P4, e.g., to allow the definition of network topologies in the frontend or to support more architectures and data plane APIs. Further, it may also be used to teach students the first steps of P4 without programming a controller. UniCorn-P4 is open source and can be found on GitHub [8].

References

[1] A. Shirmarz and A. Ghaffari, "Performance Issues and Solutions in SDN-based Data Center: a Survey," *The Journal of Supercomputing*, vol. 76, 10 2020.

[2] Q. Long, Y. Chen, H. Zhang, and X. Lei, "Software Defined 5G and 6G Networks: a Survey," *Mobile Networks and Applications*, vol. 27, no. 5, pp. 1792–1812, 2022.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 87—95, July 2014.

[4] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research," *Journal of Network and Computer Applications (JNCA)*, vol. 212, Mar. 2023.

[5] S. Qi, K. Ramakrishnan, and J.-C. Chen, "$L^2$6GC: Evolving the Low Latency Core for Future Cellular Networks," *IEEE Internet Computing*, 2024.

[6] F. Paolucci, D. Scano, F. Cugini, A. Sgambelluri, L. Valcarenghi, C. Cavazzoni, G. Ferraris, and P. Castoldi, "User Plane Function Offloading in P4 Switches for Enhanced 5G Mobile Edge Computing," in *IEEE International Conference on the Design of Reliable Communication Networks (DRCN)*, pp. 1–3, 2021.

[7] V. Jain, S. Panda, S. Qi, and K. Ramakrishnan, "Evolving to 6G: Improving the Cellular Core to Lower Control and Data Plane Latency," in *IEEE International Conference on 6G Networking (6GNet)*, pp. 1–8, 2022.

[8] F. Ihle, M. Flüchter, S. Lindner, and M. Menth, "UniCorn-P4." https://github.com/uni-tue-kn/UniCorn-P4.

[9] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *ACM SIGCOMM Workshop on Hot Topics in Networks*, no. 19, 2010.

[10] P4 Language Consortium, "GitHub: Behavioural Model Version 2 (BMv2)." https://github.com/p4lang/behavioral-model. *Last accessed on 17.07.2024*.

[11] Intel®, "$P4_{16}$ Intel® Tofino™ Native Architecture – Public Version." https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, Apr. 2021. *Last accessed on 17.07.2024*.

[12] The P4.org Architecture Working Group, "$P4_{16}$ Portable Switch Architecture (PSA)." https://p4.org/p4-spec/docs/PSA.html, Apr. 2021. *Last accessed on 17.07.2024* (working draft).

[13] The gRPC Authors, "gRPC - A High Performance, Open Source Universal RPC Framework." https://grpc.io/. *Last accessed on 15.07.2024*.

[14] The P4.org Architecture Working Group, "P4Runtime Specification." https://github.com/p4lang/p4runtime, Apr. 2021. *Last accessed on 18.07.2024*.

[15] A. Bas, "p4runtime-go-client." https://github.com/antoninbas/p4runtime-go-client, July 2020. *Last accessed on 18.07.2024*.

[16] APS Networks, "Barefoot Runtime Helper." https://github.com/APS-Networks/bfrt-helper. *Last accessed on 15.07.2024*.

[17] S. Lindner and F. Ihle, "Rust BF Runtime Interface (rbfrt)." https://github.com/uni-tue-kn/rbfrt. *Last accessed on 15.07.2024*.

[18] Meta Open Source, "React Library." https://react.dev/. *Last accessed on 15.07.2024*.

[19] A. Lo, "Controlling P4Runtime-enabled Mellanox Spectrum Switch with ONOS." https://wiki.onosproject.org/display/ONOS/Controlling+P4Runtime-enabled+Mellanox+Spectrum+switch+with+ONOS. *Last accessed on 22.08.2024*.