Safe and Secure Software-Defined Networks in P4

Jens Kanstrup Larsen and Alceste Scalas

Technical University of Denmark — {jekla,alcsc}@dtu.dk

Abstract

Software-Defined Networking (SDN) is a modern approach to network management that uses *programmable* controllers to direct and reconfigure the flow of network traffic. The P4 programming language is an open source, vendor-neutral standard adopted by an ever-increasing set of programmable network control devices. SDN makes network management more flexible — however, with increased programmability comes an increased possibility of introducing bugs, resulting in network failures. Motivated by this, we aim to design and develop a strongly-typed, statically-verified DSL for writing P4 *control plane* programs — i.e. programs that update the configuration of programmable network controllers. In this abstract we illustrate the problem, and outline our preliminary design and formalisation; we also outline our longer-term vision of a fully-verified P4 programming pipeline, ensuring that desired network properties will never be broken by configuration updates.

1 An overview of P4

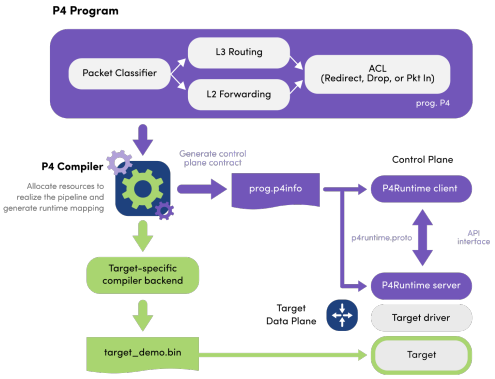
The P4 SDN standard has two main components: a *data plane* programming language (called P4), and a *control plane* programmer interface (called P4Runtime API).

A P4 data plane program specifies how a device is expected process network packets, via *tables* that match incoming packets (e.g. by inspecting their IP protocol header) and perform actions (e.g. drop a packet, of forward it to another network device). P4 data plane programs are compiled, deployed and executed on P4-compatible network routers; the hardware vendors typically provide the necessary compiler. When compiling a P4 data plane program, an associated *P4info* file is generated, which describes the routing tables and their actions.

The P4 *control plane* API (P4Runtime) specifies how an external program (a P4Runtime client) can interact with a P4-enabled device (a P4Runtime server), e.g. to query its routing tables and actions or update them, thus changing the network configuration. When interacting with a P4Runtime server, the client API uses a P4Info file (see above) that is expected to match the server configuration. The P4 workflow is outlined in figure 1 (left).

Some potential issues in P4Runtime programs. The P4Runtime API is defined by a protobuf[1] specification. Such specifications are language-independent, and provide limited type information. From a Protobuf specification, various tools can automatically generate the glue code which allows to perform P4Runtime API calls in various programming languages. P4Runtime officially supports Python (by providing a ready-to-use APIs), resulting in applications similar to figure 1 (right): that code inserts a new table entry into a table called `"IPv4_table"`, which compares a packet's IPv4 address field (called `"dst_addr"`) with the `dst_ip_addr` value, running the action `"Tunnel_Ingress"` if it matches. Observe that the table, field and action names in figure 1 are all provided as strings. The P4Runtime API performs some runtime checks (based on the aforementioned P4Info file) to verify whether e.g. a table called `"IPv4_table"` exists on the server, and whether it has a match field called `"dst_addr"`, and a possible action called `"Tunnel_Ingress"`; if not, it raises an exception — and only then

[1] https://developers.google.com/protocol-buffers




```
table_entry = p4info_helper.buildTableEntry(
    table_name="IPv4_Table",
    match_fields={
        "dst_addr": (dst_ip_addr, 32)
    },
    action_name="Tunnel_Ingress",
    action_params={
        "dst_id": tunnel_id,
    })
ingress_sw.WriteTableEntry(table_entry)
```

Figure 1: The P4 workflow (image from https://p4.org/) and a P4Runtime client program (from https://github.com/p4lang/tutorials/blob/master/exercises/p4runtime/mycontroller.py).

would the client know that their P4Runtime API calls are incorrect. This run-time exception may crash or halt the P4Runtime client program, thus leaving a network unconfigured, or partially configured; we want to spot this mistake (and others) *statically*.

2 Towards a DSL for safe P4 control plane programs

To address the issues outlined in Section 1, we aim at designing and developing a strongly-typed DSL for P4 control plane programs, able to statically ensure that P4Runtime client-server interactions never go wrong. The key idea is that we want to capture information available in the P4Info file *at the type level*, and provide a generic, strongly-typed P4Runtime client API that adds stricter type checks on top of the existing (weakly-typed) Protobuf-based API.

```
type TableActions[T] =
  T match
    "IPv4_Table" =>
        "Drop"
      | "IPv4_Forward"
      | "Tunnel_Ingress"
    "Tunnel_Table" =>
        "Drop"
      | "Tunnel_Forward"
      | "Tunnel_Egress"
```

Our plan is to design an *embedded* DSL that reuses (as much as possible) the typing system of an existing host language, and only adds a thin layer on top of the Protobuf-generated P4Runtime API. In particular, we plan to leverage three powerful features of the Scala 3 programming language: *singleton types*, *dependent function types*, and *match types*. For example, consider the match type on the left: it says that the type `TableActions[T]` is determined by the type argument `T`; and in particular, an instance of type `TableActions["IPv4_Table"]` (where `"IPv4_Table"` is the singleton type only inhabited by the homonymous string) is the union between the singleton types `"Drop"`, `"IPv4_Forward"` and `"Tunnel_Ingress"`. By combining these match type constraints with Scala 3's dependent function types, we plan to design a strongly-typed P4Runtime API such that, if a table called `"IPv4_Table"` is being updated (as in fig. 1), then the programmer can only provide an action between `"Drop"`, `"IPv4_Forward"`, or `"Tunnel_Ingress"`; other actions (like `"Tunnel_Forward"` or `"Tunnel_Egress"`) are only available when updating other tables (in this example, `"Tunnel_Table"`).

Preliminary formalisation. Our formalisation models simple networks with one P4Runtime server and one client; we will later address networks with multiple clients and servers.

To allow for a smooth transition between formalisation and implementation as embedded Scala 3 DSL, we formalise a version of $F_{<:}$ (System F with subtyping) extended with communication and concurrency primitives (inspired by the π-calculus) and elements of Scala 3 singleton

types, dependent function types, and match types taken from [2]. This design is inspired by $\lambda^{\pi}_{\leqslant}$ [10]. The resulting syntax of a P4Runtime client (t) and its types (T) is summarised below. Due to space limits, we omit the syntax of P4Runtime servers and networks, and their semantics.

$$
\begin{array}{rcl}
t & ::= & v \mid x \mid y \mid z \mid \ldots \\
& \mid & \texttt{end}\ t \mid t\ \texttt{>>}\ t \\
& \mid & \{f_1 = t_1, \ldots, f_n = t_n\} \\
& \mid & t.f \mid \lambda x : T.t \mid \lambda X <: T.t \\
& \mid & t\ t \mid t\ T \mid op(\overline{t}) \\
& \mid & t\ \texttt{match}\ \{\overline{x : T \Rightarrow t}\}
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & 0|1|2|\ldots|\texttt{tt}|\texttt{ff}|\texttt{"string"}|\ldots \\
& \mid & \{f_1 = v_1, \ldots, f_n = v_n\} \\
& \mid & \lambda x : T.t \quad (\text{with } fv(t) \subseteq \{x\}) \\
\\
op & ::= & \texttt{read} \mid \texttt{insert} \\
& \mid & \texttt{modify} \mid \texttt{delete}
\end{array}
\qquad
\begin{array}{rcl}
T & ::= & Int \mid Bool \mid Str \mid \ldots \\
& \mid & \{f_1 : T_1, \ldots, f_n : T_n\} \\
& \mid & X \mid \Pi x : T.T \\
& \mid & \forall X <: T.T \\
& \mid & T\ \texttt{match}\ \{\overline{T \Rightarrow T}\} \\
& \mid & \underline{v}
\end{array}
$$

The syntax includes ground types and values (integers, booleans, ...), records, function abstraction and application, and several operations (op) which are abstractions of the actual P4Runtime API calls. Furthermore, it has dedicated constructs for monadic operations, namely `end` (*unit*) and `>>` (*bind*), which are meant to encapsulate the effects of the P4Runtime API.

The key elements of the type system are the `match` construct, dependent function types ($\Pi x : T.T$), and the singleton type constructor $\underline{v}$. A `match` on types is conceptually similar to a `match` on terms. The singleton type constructor $\underline{v}$ represents the unique type only inhabited by the value v. For example, $\underline{42}$ is the type of the integer 42, and $\underline{42}$ is also a subtype of the Int type, i.e. $\underline{42} <: Int$. Our plan is to use a given P4Info file to synthesise a set of match type constraints similar to `TableActions` (shown in the opening of this section), in order to validate the usage of the P4Runtime operations (`read`, `insert`, `modify`, `delete`).

3 Conclusion

We outlined our preliminary work towards an embedded DSL for safe and secure P4 control plane programs. Our next goal is to complete the DSL formalisation, prove its properties, and implement it, supporting the essential P4Runtime API calls. The DSL type safety will ensure that critical P4Runtime errors (e.g. updating a nonexistent table) are caught at compile time.

From there, our longer-term plan is to achieve a *fully verified* P4 programming pipeline where strongly-typed P4Runtime control plane programs only deploy *verified* data plane configurations, ensuring that desired network properties (such as node reachability) are preserved across updates. This is a much more ambitious goal, that will require the verification of whole networks of P4-enabled devices, and will combine both data plane and control plane verification.

Related Work. Several works address the verification of the control plane in various models of software-defined networks — but (to the best of our knowledge) no previous work addresses the P4 control plane. [9][4] are graph-based in their analysis, and they focus on the BGP routing protocol. FSR [11] introduces a new type of analysis based on routing algebra, still focused on BGP. Batfish [5] is a protocol-agnostic tool that also considers data plane snapshots when performing analysis. Notably, Batfish supports compiling many vendor-specific configuration languages to an intermediary language. For this reason, many subsequent tools base their verification on this intermediary language. The recent work [3] proposes a formal language for analysing the control/data plane interactions in SDNs based on NetKAT [1].

Several tools specifically analyse P4 *data plane* programs. Their methodologies include translation to the verifiable language Datalog [8], static analysis based on formal semantics [6], and program annotations [7]. However, no tool currently exists for analyzing the P4 *control* plane, and in particular, there is no existing tool for verifying P4Runtime API calls.

Acknowledgements. We thank the anonymous reviewers of NWPT 2022, and our collaborators Philipp Haller and Roberto Guanciale (KTH) for their suggestions and feedback.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *SIGPLAN Not.*, 49(1):113–126, jan 2014.

[2] Olivier Blanvillain, Jonathan Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. Technical report, 2021.

[3] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi, and Hünkar Can Tunç. DyNetKAT: An algebra of dynamic networks. In Patricia Bouyer and Lutz Schröder, editors, *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13242 of *Lecture Notes in Computer Science*, pages 184–204. Springer, 2022.

[4] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 43–56, 2005.

[5] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, 2015.

[6] Ali Kheradmand. A formal semantics of P4 and applications. Master's thesis, 2018.

[7] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

[8] Nuno Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. Automatically verifying reachability and well-formedness in p4 networks. *Technical Report, Tech. Rep*, 2016.

[9] Bruno Quoitin and Steve Uhlig. Modeling the routing of an autonomous system with c-bgp. *IEEE network*, 19(6):12–19, 2005.

[10] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 502–516, 2019.

[11] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking*, 20(6):1814–1827, 2012.