Formal Specifications for Data Plane Programs

Keyu Yuan
kyuanab@connect.ust.hk
Huawei Technologies
Hong Kong SAR, China

Baber Rehman
brehman@connect.hku.hk
Huawei Technologies
Hong Kong SAR, China

Abstract

Networks are complex in nature consisting of numerous configurations. Each network configuration requires careful consideration due to the high runtime impact. Software-defined Networking (SDN) opens control plane and data plane programmability for the end user. The network programmability, on one hand, allows flexibility of defining custom headers and protocols. On the other hand, it opens new horizons of network misconfigurations. Formal methods are being applied in various domains for building robust systems. Recent studies explored the applications of formal methods in network programming. This paper explores the possibility of using Backus-Naur Form (BNF) for writing formal specifications for P4 parsers and tables. The compiler checks at compile time that the specifications are consistent with the implementation and ensures correctness.

CCS Concepts

• **Networks** → **Programmable networks**; • **Software and its engineering** → **Formal methods**.

Keywords

Data Plane Programming, Specifications, Correctness

ACM Reference Format:
Keyu Yuan and Baber Rehman. 2025. Formal Specifications for Data Plane Programs. In *2nd Workshop on Formal Methods Aided Network Operation (FMANO '25), September 8–11, 2025, Coimbra, Portugal*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3750022.3750458

1 Explanation

We live in a digitally connected world where networks play an essential role. Software-defined Networking (SDN) [7] has revolutionized modern networks by separating monolithic network architecture into control plane and data plane. P4 [1] is a widely used Domain Specific Language (DSL) for data plane programming. It has gained significant research interest in the community [3, 13–16]. However, enabling users to write data plane programs introduces potential vulnerabilities in network correctness, which is a vital property for reliable network operation. Formal methods are mature enough for specifying and ensuring correctness. Various researchers [3–5, 8, 11] studied formal correctness properties for P4 or for P4 programs. This paper proposes a lightweight and novel approach for specifying the specifications for P4 *parsers* and *tables*.

P4 Parser. A parser is a core component in P4 when defining custom packet headers. It *parses* the bits into specific *headers* or *struct* fields.

P4 Table. Table lookup and action execution are two essential functions in a P4 pipeline. *Tables* provide an abstraction to the user for setting custom lookup rules and associate actions with tables.

1.1 Specifications for Parsers

In the context of P4 programming for data plane development, ensuring accurate implementation of packet header structures is fundamental to reliable network processing. This subsection presents a verification methodology for header extraction in P4 parsers using Backus-Naur Form (BNF) grammatical specifications.

The header layout specification serves as a critical component in defining data plane programs, explicitly declaring the types of packets the program should process. We demonstrate that BNF provides an effective formalism for describing expected packet structures through context-free grammar rules. Consider the following production rule:

$$\texttt{Parsing} ::= \texttt{Ether} \wedge (\texttt{IPV4} \mid \texttt{IPV6}) \wedge (\texttt{TCP} \mid \texttt{UDP})$$

This specification requires the parser to first extract an Ethernet header, followed by either an `IPv4` or `IPv6` header, and finally terminate with either a `TCP` or a `UDP` header. The logical conjunction operator ($\wedge$) enforces strict sequential composition of header fields while maintaining branch flexibility through disjunction operators.

In addition to enhancing readability, with the help of Correct-by-Construction (CbC) [2, 6, 9], we point out that this structured representation can also serve as a reference for validating the implementation of the parser. In the BNF specification, we can have each terminal as a header-extractions, while each rule as a parser-invoke. Then we can validate user's parser implementation with respect to their BNF header specification to avoid unexpected header extraction paths. For example, in a $P4_{16}$ program ($\wedge$ and & are same):

```
// Header definition will automatically create a terminal
     with header's name.
header Ether { ... }
header IPV4  { ... }
...

//{ TopParser ::= Ether & (IPV4 | IPV6) & (TCP | UDP) }
parser TopParser(...) {
  state start {
    packet_in.extract<Ether>(...);
    // ok, TopParser handles packets beginning with Ether
    transition select(...) {
      ...: parse_ipv4
      ...: parse_tcp
    }
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FMANO '25, Coimbra, Portugal
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2103-8/25/09
https://doi.org/10.1145/3750022.3750458

```
  }
  state parse_ipv4 {
    packet_in.extract<IPV4>(...);
    // ok, IPV4 is one alternative after ethernet
  }
  state parse_tcp {
    packet_in.extract<TCP>(...);
    // not ok, there shouldn't be a packet that has tcp
     immediately after ethernet
  }
}
```

This BNF specification also perfectly suits for sub-parsers, that user will define another rule describing the sub-parser and use that rule as a non-terminal in the main rule.

```
//{ L3 ::= IPV4 | IPV6 }
parser L3(...) {
  state start { ... }
  state parse_ipv4 { ... }
  state parser_ipv6 { ... }
}

parser L2(...) { ... }
parser L4(...) { ... }

//{ TopParser ::= L2 & L3 & L4 }
parser TopParser(...) {
  state start {
    L2().apply();
    L3().apply();
    L4().apply();
  }
}
```

The verifier becomes even more powerful when combined with dataflow analysis. As a common practice in $P4_{16}$, parsed headers are typically stored in a unified metadata structure where each field corresponds to a specific header type. When headers are extracted, their respective structure fields are marked valid while others remain invalid. With this knowledge in mind, the verifier can infer the possible combinations of valid fields in the structure based on the BNF specification. For example, the expression `(IPV4 | IPV6)` indicates that hdr.ipv4 and hdr.ipv6 cannot be both valid; they are mutually exclusive. The verifier can therefore flag the program below for a potential bug.

```
struct headers {
  Ether eth;
  IPV4 ipv4;
  IPV6 ipv6;
}
//{ TopParser ::= Ether (IPV4 | IPV6) }
parser TopParser(out headers hdr, ...) { ... }
control ctr(inout headers hdr) {
  apply {
    bit<32> ipv4 = hdr.ipv4.src;
    bit<128> ipv6 = hdr.ipv6.src; // error: ipv4 and ipv6
      cannot be valid at the same time
  }
}
```

1.2 Specifications for Tables

In P4 programming, tables play a critical role in defining the behavior of packet processing within the dataplane. The language allows users to define multiple tables, where each table typically serves a distinct purpose, such as forwarding, filtering, or modifying packet headers. Ensuring that these tables are applied in the correct order and condition is essential for maintaining the intended functionality of a network device. This subsection discusses the verification of table application order using the concept of BNF as discussed in Section 1.1 for parsers.

Through inspecting public projects [10], we have found that programmer defined tables may have these relationships:

- **Consecutive Tables (`A` $\Rightarrow$ `B`)**: Table B should be applied after table A. For example, a table for modification should be applied after a table for filtering.
- **Default Tables (`A` $\rightarrow$ `B`)**: Table B should be applied only if table A missed. For example, a table for fallback actions should be applied after the main table missed.
- **Alternative Tables (`A | B`)**: Either one of Table A or Table B should be applied. For example, to process a single L3 header, only one of the `IPv4` matching tables or the `IPv6` matching tables should be applied.

By composing these annotations, users can specify their intended table application graph in their programs. For example, a specification like:

$$(A|B) \Rightarrow C \rightarrow D$$

indicates that table A or B should be matched first, then C, and if C misses then D. With these specifications in hand, the compiler can then perform control-flow analysis in the apply block of user's program to determine whether the applications of tables are consistent with the specifications. This ensures correctness in the order of table apply.

```
control MyPipe(...) {
  table A {...}
  table B {...}
  table C {...}
  table D {...}
  //{ (A|B) => C -> D }
  apply {
    if(...)
      A.apply();
      // OK, A or B are the first table to be applied
    else
      B.apply();
      // OK, A and B are in different branches
    if(C.apply().hit)
    // OK, C is applied after A/B
      D.apply();
      // Not OK, D should be applied only if C is missed
  }
}
```

2 Conclusion and Future Work

This paper proposes lightweight specifications for P4 parsers and tables, which are two core components of a P4 program. Programmer specifies the header validity and table execution order following the standard BNF format. The static analyzer ensures that the implementation is consistent with the specifications. This removes unintended behaviors from parsers and tables. We plan to implement this in P4C [12] as a future work. This paper does not raise any ethical issues.

References

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

[2] Edsger Wybe Dijkstra. 1997. *A Discipline of Programming* (1st ed.). Prentice Hall PTR, USA.

[3] Matthias Eichholz, Eric Campbell, Nate Foster, Guido Salvaneschi, and Mira Mezini. 2019. How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 12:1–12:28. doi:10.4230/LIPIcs.ECOOP.2019.12

[4] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-typed data plane programming. *Proc. ACM Program. Lang.* 6, POPL, Article 40 (Jan. 2022), 28 pages. doi:10.1145/3498701

[5] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *Proceedings of the Symposium on SDN Research* (Los Angeles, CA, USA) *(SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. doi:10.1145/3185467.3185499

[6] Derrick G. Kourie and Bruce W. Watson. 2012. *The Correctness-by-Construction Approach to Programming* (1st ed.). Springer, Berlin, Heidelberg.

[7] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (2015), 14–76. doi:10.1109/JPROC.2014.2371999

[8] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. doi:10.1145/3230543.3230582

[9] Carroll Morgan. 1994. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., GBR.

[10] Princeton-Cabernet. 2025. Open source P4 implementations. https://github.com/Princeton-Cabernet/p4-projects

[11] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 518–532. doi:10.1145/3230543.3230548

[12] The P4.org Architecture Working Group. 2016. P4 Reference Compiler (P4C). https://github.com/p4lang/p4c

[13] The P4.org Architecture Working Group. 2021. P4Runtime Specification. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[14] The P4.org Architecture Working Group. 2024. P416 Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html Accessed on 15 January 2025.

[15] The P4.org Language Consortium. 2024. P4 Portable NIC Architecture (PNA). https://p4.org/p4-spec/docs/PNA.html Accessed on 15 January 2025.

[16] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. 2020. P2GO: P4 Profile-Guided Optimizations. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks* (Virtual Event, USA) *(HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 146–152. doi:10.1145/3422604.3425941