Proceedings of the
13th International Conference on Applied Informatics
Eger, Hungary, February 19–21, 2026
DOI: 10.17048/icai.2026.169
pp. 169–181

Program Patterns in P4 Programs*

Zsófia Laczkó, Máté Tejfel

Eötvös Loránd University, ELTE, Budapest
laczkozsofi@student.elte.hu
matej@inf.elte.hu

Abstract. Programming Protocol-independent Packet Processors (P4) [2] is an open-source domain-specific programming language, which is primarily developed to describe algorithms that determine the forwarding of network data packets. In this research we defined and analysed positive (recommended) and negative (to be avoided) programming patterns in P4 codes. The defined templates were examined by static methodology. We have implemented an analyser program for the investigation of the patterns, integrated in P4Query [7], which is a static analyser tool for P4. Our analyser is able to recognise certain positive or negative templates, as well as provide feedback to developers about patterns found in their code. During the research the analyses that are already in the P4Query tool were utilized, and a new module was created built on them. Primarily, the syntax tree and the control flow graph built from the source code was used. The goal of the paper is to reduce the number of errors in P4 codes and to demonstrate which recurring solutions support or hinder the correct and effective processing of data packets.

Keywords: P4 language, P4Query, program patterns, static analysis

1. Introduction

Programming Protocol-independent Packet Processors (P4) [2] is an open source domain-specific programming language, which is primarily designed to implement algorithms that describe packet processing by different network devices (switches, network interface cards (NICs), routers, etc.). P4 is increasingly central to the de-

*This research was supported by the project no. TKP2021-NVA-29, which has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

velopment of programmable data planes, in particular software-defined networking (SDN) and high-performance network devices.

Specific programming challenges emerge when working with this language. It has pipeline-based processing model, and target-dependent compilation. P4 is a flexible language, intended for high-speed processing of numerous packets. For this reason, there is no well-developed error handling. If an error occurs, the basic solution is that the P4 program drops the packet, which can be a problem in particular cases. Lookup tables also introduce challenges, since it cannot be determined explicitly in advance which action will be executed by the application of a table. P4 only specifies the list of the available actions and establishes the schema of the lookup table. The actual table entries are defined externally to the P4 program and can be modified dynamically. For this reason, it is not visible during development which actions and parameter configurations correspond to given key values.

Due to these specific challenges, static analysis is notably important for P4 language, in addition to dynamic checks. This motivated our research: the identification of positive and negative programming patterns in P4 programs, as well as their automatic detection through static analysis. The positive patterns include recommended coding styles, best practices and solutions. As for the negative patterns, coding strategies that seem to work but potentially contain errors or unexpected behaviour were examined. This category includes examples that indicate defects only in specific edge cases. In this research, we focused on the specific scenarios for which the compiler does not alert. Our new analyzer module is able to locate certain positive or negative templates in P4 programs, as well as inform developers about the occurrences of these patterns in their code.

This research contributes to the prediction of frequent potential errors through the static analysis of P4 code, so we can increase the number of issues that can be identified without running the program. In addition, this study can support work of developers by emphasising well-functioning strategies.

2. Related work

Several articles address debugging in P4 programs, but mostly using dynamic methods such as assertions [4]. Birnfeld et al. [1] analyse P4 programs following an approach similar to the one we used. They examine problems like our negative patterns, however, they use data flow analysis specifically. This approach is useful for examining the values of variables within the program. Compared to this, the syntax tree and control flow graph analysis presented in this study is more effective for inspecting code structure and possible execution paths.

Although Gulabovska et al. [5] provides tools for analysing a different programming language (Python), the description of its static analysis methodology served as a basis for our approach. In this paper, this technique was applied for the examination of patterns. Vera [10] is a tool developed to discover errors in P4 programs, similar to our analyser module. However, it uses symbolic execution, not static

analysis. This method requires inputs. Therefore, analysing all possible execution paths sing this approach is difficult. In P4, the use of lookup tables results in a large number of potential paths.

The p4v [6] program checks properties of P4 programs using formal methods and can recognise errors. This approach translates P4 programs into an imperative language called Guarded Command Language (GCL). The verification process is then applied to this representation instead of analysing the original program code directly. The bf4 [3] tool applies both static verification and runtime checks. In addition, it is able to also modify code to prevent errors. Both this program and the previously mentioned ones are primarily designed for detecting and correcting errors. In contrast, this paper also emphasizes positive patterns, thereby providing constructive feedback to developers.

3. Background

The structure of a P4 program depends on a configuration file called architecture. This file determines the mandatory elements of a P4 program. Developers can use custom architectures for their network, however there are some widely used models. In this research we worked with the V1Model, which is one of the most used models.

3.1. Code structure and V1Model architecture

In a P4 program[1], the first section of the code describes the type definitions and the headers applied, illustrated in Listing 1.

Listing 1. Definition part.

```
#include <core.p4>
#include <v1model.p4>
...
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16> etherType;
}

header ipv4_t {
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
    ...
}
...
```

The V1Model divides P4 programs into five different blocks, as presented in Figure 1.

[1]The example code in this section is from: https://github.com/p4lang/tutorials (basic.p4)

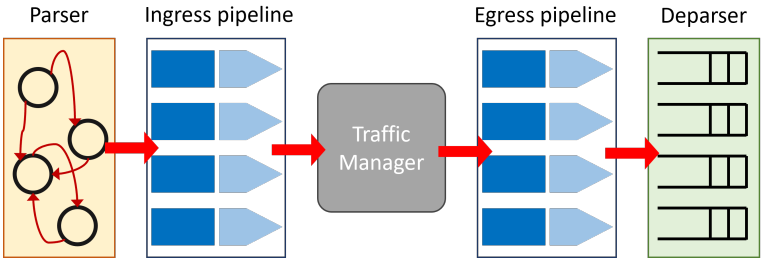

Figure 1. V1Model.

Listing 2 demonstrates the parser, which is the first block of the V1Model. This is responsible for packet processing. The program receives a bitstream at the beginning, from which the parser extracts the packet headers and separates them into fields for further handling.

Listing 2. Parser.

```
parser MyParser(packet_in packet, out headers hdr,
                inout metadata meta, ... ) {
    state start { transition parse_ethernet; }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept; }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4); transition accept; }
}
```

Listing 3 presents the second block, the ingress pipeline. The majority of the processing logic for incoming packets is executed here. Primarily lookup tables[2] are used to update the values of fields within headers or metadata. When a table is applied, a lookup is performed based on the specified key fields, and the resulting match determines an action from those defined in the table declaration. This selected action is subsequently executed. It is important to emphasize that the actual table entries are provided outside the P4 program and they can be changed during runtime.

Listing 3. Ingress pipeline.

```
control MyIngress(inout headers hdr, inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() { mark_to_drop(standard_metadata); }
    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        ... }
    table ipv4_lpm {
```

[2]Detailed description of the tables: https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.html#sec-tables

```
8           key = { hdr.ipv4.dstAddr: lpm; }
9           actions = { ipv4_forward; drop; NoAction; } ... }
10      apply {  if (hdr.ipv4.isValid()) { ipv4_lpm.apply(); } else {}  }
11  }
```

After ingress block comes the Traffic Manager, which cannot be programmed by P4 language, so it is outside the scope of this work.

The next block is the egress block, which is similar to ingress. This block may be empty; however, it is handled separately since some architectures specify which operations can appear in the ingress and which in the egress block.

Listing 4 illustrates the final block, which is the deparser. This section assembles the packet for forwarding.

Listing 4. Deparser.

```
1  control MyDeparser(packet_out packet, in headers hdr) {
2      apply { packet.emit(hdr.ethernet); packet.emit(hdr.ipv4); }
3  }
```

3.2. P4Query

Our new module is integrated in P4Query [7], which is a static analyser tool for P4. It can map a P4 program code onto a graph applying the Gremlin graph description language [9]. Gremlin is a graph traversal and graph query framework that can be used for example from the Java programming language. P4Query tool contains different analyses that result in an internal graph representation. With the graph we are able to process the code from different perspectives. Each analysis can rely on previously generated results, which are also integrated in the representation graphs (mainly by adding new edges to the graph), illustrated in Figure 2. This approach allows the execution of different analysers sequentially according to a dependency chain. The tool makes possible to examine for instance a syntax tree, a symbol table or a control flow graph.

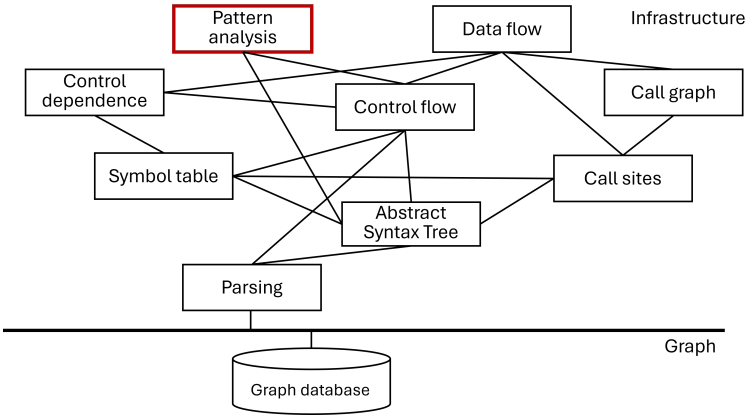

Figure 2. Analyses in P4Query.

4. Patterns

In the first part of this research recurring code patterns and programming styles were identified in P4 programs. The collected patterns were classified into two main categories: positive (recommended) and negative (to be avoided) samples. Positive patterns represent widely adopted implementation approaches that contribute to code readability, maintainability, and reliable program execution. Conversely, negative patterns introduce potential errors or ineffective code fragments.

Highlighting positive patterns provides inspection into the degree to how developers follow the conventions and best practices. The resulting feedback can support the improvement of the development team. Moreover, positive reinforcement and recognition have a motivating effect on developers. In contrast, the identification of negative patterns serves to warn developers about potential issues and risks.

During the examination of patterns, the specification of the programming language [8] was used, as well as the P4C compiler, which is the official reference compiler for the P4 language. Potentially incorrect code samples were collected and then analysed to determine in which cases problems are reported during compilation, either in the form of an error or a warning. Although this collection of patterns includes cases for which the compiler provides warnings, we mainly concentrated on those that are not detected.

The collection was not intended to cover the entire documentation. Instead, it serves as an illustrative set of examples for the analyser module. It can be further extended through the study of real-world code bases and examine other language constructions.

4.1. Positive patterns

Pattern 4.1.1. Headers have to be used with *valid* status with all their fields properly assigned values.

In P4, every header structure contains a hidden field, which has Boolean type. When its value is `true`, the header is considered *valid*. Fundamentally, the compiler issues a warning if the status of a header is set to *invalid* and then one of its fields is attempted to be used. However, if a header has been set to *invalid* status, and then to *valid* status, then values have to be assigned to its fields before being used. In this case the compiler does not issue a warning.

Pattern 4.1.2. Widely known header types should be used according to the conventional or standard practice.

In P4 programs, header types are defined by the programmer at the beginning of the source code. When a type is declared with the name of a well-known header, such as IPv4, its field names and field lengths should follow the conventional structure of that header for improving code clarity. Further examples of well-known header types are IPv6, TCP, UDP and Ethernet.

Pattern 4.1.3. Define a variable only if it did not previously exist and refer to it only if it has already been defined and initialized.

This can be verified based on the control flow analysis. In the case of local variables, the compiler typically indicates the problem, at least in the form of a warning. Regarding metadata fields, special attention should be paid to their initialization and first usage.

Pattern 4.1.4. Define variables only with permitted lengths (for instance, bit<1>, bit<8>, ...).

Different architectures and targets may support only specific sizes. As an example, some targets are able to handle only byte-wide (8-bit) variables well.

Pattern 4.1.5. In the deparse part, it is recommended to explicitly determine the order of appending data to the output packet.

There are several ways to define headers for deparsing. The simplest option is to specify the headers name explicitly. In this case, the sequence of the instructions clearly determines the order of the appending. The situation is more complex when a struct containing the headers is deparsed. In such cases, the append order is implicitly defined by the type declaration of the struct. It is also possible to deparse a header stack, in which scenario the elements are processed sequentially according to their increasing indices. In this instance the order does not appear explicitly. The implicit sequence does not automatically result in an error; however, it demands careful attention in every case.

4.2. Negative patterns

Code may contain unnecessary parts, including both syntactically observable ones and those that become superfluous during program execution.

Pattern 4.2.1. Writing unused variables, declarations (e.g. with `typedef` keyword), formal parameter, states.

Pattern 4.2.2. Unreachable code segments, like code after a return statement.

Pattern 4.2.3. Setting a header as *valid* when it has already a *valid* status.

Pattern 4.2.4. The deparser processes an *invalid* header.

In the deparser block, the appending of the data from the header is performed using the `emit` method. When it receives an *invalid* header as parameter, no action is taken. This is not necessarily incorrect code. It enables a simplified programming approach when the presence of a header is optional. However, it can easily lead to an error and is therefore reported.

Issues can occur in relation to the usage of variables, depending on the values assigned to them.

Pattern 4.2.5. Variable overflow and type conversions can cause errors.

For example, the following operations may occur an error [8]:

- int<W> $\longleftrightarrow$ bit<W>
  Issues can occur when converting between positive and negative numbers.

- bit<W> ⟶ bit<X>, int<W> ⟶ int<X>
  It truncates the value when W > X.
- int ⟶ bit<W>, int ⟶ int<W>
  Value can be truncated, overflow may occur, converting between positive and negative numbers can be incorrect. (For the last two cases, the compiler provides warnings as well.)

Pattern 4.2.6. Reading a field of a header without previously manually setting values to that field or parsing the header [1]. This results in an unspecified value.

A specific instance of this pattern is when a packet lacking an IPv4 header is handled by the `ipv4_lpm` table [1]. In this case, a header field would be used as a key which has not been parsed.

The following two patterns do not primarily cause errors. Instead, they are related to code maintainability and the appropriate use of language constructs.

Pattern 4.2.7. Repetition of larger code segments.

Pattern 4.2.8. Writing a select statement without a default case.

5. Analyser module

In order to facilitate development process, a new analyser program[3] was developed to automatically locate the collected patterns in P4 codes. Static analysis was used to achieve this, since it enables the examination of program behaviour without executing the code and the identification of the potential issues directly from the program structure. For the detection of the patterns, the syntax tree and the control flow graph (CFG) have to be examined. The new analyser module is integrated into P4Query tool, as it is able to provide an internal graph representation which contains the syntax tree together with the associated control-flow edges. This complex internal graph representation contains information derived from both the syntax tree and the control-flow analysis. This data was adequate for the currently implemented examinations. However, other analyses available in P4Query can also be utilized if further information is needed during future development.

The implementation includes several distinct analyser classes, which receive this graph representation as an input parameter. Each class is responsible for detecting one or more interconnected patterns. The Gremlin graph traversal language was primarily used for the analysis.

Table 1 contains which class is responsible for detecting which pattern. As shown in the table, the analyser is currently not able to identify all patterns. Some of the missing patterns cannot be examined using the static analysis approach applied in this work. For instance, Pattern 4.1.4 requires information about the current target architecture. Other omitted samples can be implemented in the future to extend the module. If the detection of a new pattern is required, the implementation can be extended easily by creating a new analyser class.

[3]The code is accessible from https://github.com/laczkozsofi/PatternAnalysisInP4

Table 1. Patterns related to analyser classes.

Analyser class	Connected pattern
Status of headers	4.1.1, 4.1.5, 4.2.3, 4.2.4, 4.2.6
Select statement	4.2.8
Return statement	4.2.2
Known headers	4.1.2

In the implementation every possible execution path is analysed based on the control flow graph. As a result, all occurrences of the analysed patterns are identified from the code. The purpose of highlighting potential errors is not to assert that a program will definitely behave incorrectly, but rather serves a warning about potential issues that developers should consider. As a consequence of the platform-dependent nature of the P4 language, many types of information are not available from the source code alone. In addition, frequent use of lookup tables makes analysis more difficult, since it is not visible which actions will be executed. Because the actual table entries are not defined in the P4 code, the module has to consider all possible actions. This may cause false positive warnings, for which static analysis cannot offer more detailed information.

As an outcome of the analyses of the classes, a report is generated, which delivers warnings to developers about possible issues and about the found patterns. Based on the results of the examination, feedback number is provided to developers. It is calculated by assigning negative scores to each found undesirable patterns and positive scores to recommended ones. Different templates contribute to the total score with different weights. Currently one total score is calculated. However, for more transparent feedback, the value derived from positive patterns could be reported separately from the negative score. It would emphasize better the warnings for potential issues. Nevertheless, the report already displays the individual scores for each entry.

5.1. Example: Status of headers analysis

For instance, in the context of the Status of headers pattern the module examines *valid* or *invalid* status of headers during the execution of the P4 code. Additionally, it verifies whether the *valid* status of a header really guarantees that all the fields have been assigned values. Our analysis also attends to what happens with a header during parsing and deparsing, as well as when its status is set directly by the `setValid` or `setInvalid` methods. The program issues a warning if a field has not been assigned value when it is used or a header is unnecessarily marked as *valid*. In addition, developers get a report about the deparser block. It contains the expected status of the headers to be deparsed, as well as a list of fields that may not have assigned values.

Listing 5. Ingress block with potential error.

```
1  control MyIngress(...) {
2      ...
3      action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
4          standard_metadata.egress_spec = port;
5          hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
6          hdr.ethernet.dstAddr = dstAddr;
7          hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
8      }
9
10     table ipv4_lpm {
11         key = { hdr.ipv4.dstAddr: lpm; }
12         actions = { ipv4_forward; drop; NoAction; }
13         size = 1024;
14         default_action = drop();
15     }
16
17     apply {
18         if (hdr.ipv4.totalLen == 5) {
19             meta.egress_spec = 1;
20             hdr.ethernet.setInvalid();
21         } else {
22             meta.egress_spec = 2;
23         }
24         hdr.ethernet.setValid();
25
26         if (hdr.ipv4.isValid()) {
27             ipv4_lpm.apply();
28         } else {}
29     }
30 }
```

The `ipv4_lpm` table is applied in the code section presented in Listing 5, in the yellow-boxed line. It has three actions (declared in line 12), and it cannot be determined in advance which one will be performed. During execution, the apply block may follow the true branch of the first if statement, marking the Ethernet header as *invalid* (highlighted in blue). Next, when the `ipv4_lpm` table is applied, an action is selected depending on the concrete table entries. Assume that action `ipv4_forward` is executed. The destination address field (`dstAddr`) in line 5, highlighted in red would be used without a value, because even though the status of header is reset to *valid* (in line 24, highlighted in green), its fields have not been assigned any values. Our analyser module provides a warning in this case, presented in Figure 3.

```
WARNING -> action: ipv4_forward: dstAddr has no value (line: 519)

Score: -2
```

Figure 3. Output of Status of headers analysis.

5.2. Example: Implementation of an analyser class

To demonstrate the implementation of the analysers, the class connected to the status of headers is briefly introduced. The operation of the analyser class connected to header status examination can be divided into six main steps.

1. The first step is to collect the header types used in the program based on the declarations located at the beginning of the code.
2. The next step is to construct an object for tracking the current status of the headers. This object is continuously updated during the analysis of the execution paths.
3. As the third step, the parser block is analysed to identify the headers that can be parsed during packet processing.
4. The execution paths to be analysed are constructed according to the V1Model architecture. The paths are built from the vertices of the representation graph.
5. The main phase of the analysis is the examination of the controls (the blocks present in the V1Model). For every node, the module checks whether any headers or header fields are accessed and whether the status of any header is modified.
6. The last step is the examination of the deparser block. During this step, the status of all headers to be deparsed is determined and written to the report. Furthermore, for each header, the report contains a list of fields that are not guaranteed to have assigned values.

The implementation of these phases requires numerous Gremlin queries. To demonstrate this process, the first stage is presented in the following. This step is the collection of the header types. According to the language syntax[4] this operation starts with a Gremlin traversal that selects all `HeaderTypeDeclarationContext` nodes. From these nodes, the header name can be accessed via the `NameContext` child node, and the field names are reached through the `StructFieldListContext` child node.

Listing 6. Collection of the header types.

```
List<Object> headerTypeDeclarations =
        g.V().has("class", "TypeDeclarationContext")
        .out().has("class", "DerivedTypeDeclarationContext")
        .out().has("class", "HeaderTypeDeclarationContext")
        .values("nodeId").toList();

for (Object decl : headerTypeDeclarations) {
    List<Object> headerName = g.V().has("nodeId", decl)
            .out().has("class", "NameContext")
            .repeat(__.out()).until(__.not(__.out()))
```

[4] https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.html#sec-header-types

```
11              .values("value").toList();
12
13      List<Object> headerFieldsNames = g.V().has("nodeId", decl)
14              .out().has("class", "StructFieldListContext")
15              .repeat(__.outE("syn").inV())
16              .emit().dedup().has("class", "NameContext")
17              .repeat(__.out()).until(__.not(__.out()))
18              .values("value").toList();
19      ...
20  }
```

Listing 6 illustrates the Gremlin queries needed for this traversal. At the beginning, the nodes representing header declarations are selected, which are then processed in a for loop. In the first query, the process begins by locating the current header declaration node, and the traversal continues to the node that determines the name. From there, the value attribute of the reachable leaf node contains the required information. The second query follows a similar approach, except that it moves to the node declaring the list of fields instead of the name node. It selects all nodes representing field names. The necessary data is likewise contained in the reachable leaf nodes.

6. Conclusion and Future Work

The main purpose of the study is to uncover frequently occurring programming patterns in P4 programs using static analysis, including both positive and negative aspects. On one hand, the collection of recommended patterns supports the creation of more maintainable and effective P4 programs, as well as facilitating the work of developers. On the other hand, recognizing negative patterns allows for the early detection of potential problems without program execution. This reduces the complexity of later testing activities using other approaches.

Our analysis identified 5 positive and 8 negative patterns in P4 programs. The implemented static analyser successfully detects some of these patterns, highlighting common mistakes and best practices in P4 coding, as well as providing feedback for developers about found patterns in their code.

In future work, our plan is to expand the collection of identified P4 patterns, including both new positive and negative examples. In addition, our goal is to improve the implemented analyser module so it can recognize more and more patterns automatically, further supporting code quality and correctness. Further research could explore whether there is a demonstrable relationship between the patterns and code quality.

Our research group is currently conducting research on using LLMs (Large Language Models) for refactoring P4 codes. If this approach is successful, it may also be applied to error detection similar to the analyser presented in this paper. A limiting factor in this type of research is that P4 is not a widely used language, and therefore only a limited number of examples are available.

References

[1] K. Birnfeld, D. C. da Silva, W. Cordeiro, B. B. N. de França: *P4 Switch Code Data Flow Analysis: Towards Stronger Verification of Forwarding Plane Software*, in: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1–8, doi: 10.1109/NOMS47738.2020.9110307.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker: *P4: programming protocol-independent packet processors*, SIGCOMM Comput. Commun. Rev. 44.3 (July 2014), pp. 87–95, issn: 0146-4833, doi: 10.1145/2656877.2656890.

[3] D. Dumitrescu, R. Stoenescu, L. Negreanu, C. Raiciu: *bf4: towards bug-free P4 programs*, in: Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 571–585, isbn: 9781450379557, doi: 10.1145/3387514.3405888, url: https://doi.org/10.1145/3387514.3405888.

[4] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, M. Barcellos: *Uncovering Bugs in P4 Programs with Assertion-based Verification*, in: Proceedings of the Symposium on SDN Research, SOSR '18, Los Angeles, CA, USA: Association for Computing Machinery, 2018, isbn: 9781450356640, doi: 10.1145/3185467.3185499.

[5] H. Gulabovska, Z. Porkoláb: *Survey on Static Analysis Tools of Python Programs*, in: SQAMIA, 2019, url: https://api.semanticscholar.org/CorpusID:209149918.

[6] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, N. Foster: *p4v: practical verification for programmable data planes*, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, Budapest, Hungary: Association for Computing Machinery, 2018, pp. 490–503, isbn: 9781450355674, doi: 10.1145/3230543.3230582, url: https://doi.org/10.1145/3230543.3230582.

[7] D. Lukács, G. Tóth, M. Tejfel: *P4Query: Static analyser framework for P4*, Annales Mathematicae et Informaticae 57 (2023), pp. 49–64, doi: 10.33039/ami.2023.03.002.

[8] P4 Language Consortium: *P4 Specifications*, url: https://p4.org/specifications/ (visited on 12/30/2025).

[9] M. A. Rodriguez: *The Gremlin graph traversal machine and language (invited talk)*, in: Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015, Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 1–10, isbn: 9781450339025, doi: 10.1145/2815072.2815073.

[10] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, C. Raiciu: *Debugging P4 programs with vera*, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, Budapest, Hungary: Association for Computing Machinery, 2018, pp. 518–532, isbn: 9781450355674, doi: 10.1145/3230543.3230548, url: https://doi.org/10.1145/3230543.3230548.