IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 June 2025, accepted 3 July 2025, date of publication 15 July 2025, date of current version 21 July 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3589353

RESEARCH ARTICLE

P4Muse: Enabling Modular P4 Programming via Compiler-Managed Code Merging Without Syntax Modifications

MOHSEN RAHMATI[1], FRANÇOIS-RAYMOND BOYER[1], BILL PONTIKAKIS[1], JEAN PIERRE DAVID[2], (Member, IEEE), AND YVON SAVARIA[2], (Life Fellow, IEEE)

[1]Department of Computer and Software Engineering, Polytechnique Montréal, Montreal, QC H3T 1J4, Canada
[2]Department of Electrical Engineering, Polytechnique Montréal, Montreal, QC H3T 1J4, Canada

Corresponding author: Mohsen Rahmati (mohsen.rahmati@polymtl.ca)

This work was supported in part by Intel, Noviflow, Kaloom, Prompt Quebec; and in part by the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT Domain-specific programming languages such as P4 enable flexible and high-performance packet processing for programming network data planes. However, many P4 programs remain monolithic, limiting the development of modular and reusable protocols and libraries. Introducing modularity to P4 has proven challenging, as existing approaches—such as trans-compilers and virtualization—often sidestep direct integration with the P4 language and compiler, constraining backward compatibility and extensibility. This paper introduces P4Muse (P4 Modularity and Unification for Seamless Extensibility), an open-source P4C compiler extension that enhances the modularity of P4 without requiring new syntax or annotations. P4Muse is developed by integrating new compiler passes for automatic code merging, fostering modular design and reuse. We demonstrate its benefits through three classes of use cases that support P4 modularity, enabling code reusability, data plane pipeline composition, and vendor-customer compatibility. using the V1Model architecture and the BMv2 software switch. Our results show that P4Muse effectively supports modular P4 program development without altering existing P4 syntax, providing a robust solution that significantly improves code reusability, flexibility, and extensibility while maintaining backward compatibility.

INDEX TERMS Programmable data planes, P4, compiler, intermediate representation, modularity, reusability, extensibility, incremental programming, composability, BMv2, vendor-customer framework.

I. INTRODUCTION

Traditional network devices, such as routers and switches, typically use vendor-controlled algorithms for processing data in both the control and data planes [1]. Although control plane settings can be modified through user interfaces, the core algorithms remain proprietary. However, recent advances in Software-Defined Networking (SDN) and data plane programmability have shifted this paradigm.

SDN enables users to override standard control plane algorithms via APIs such as OpenFlow [2], [3], centralizing control in an SDN controller. This centralization simplifies previously complex distributed network functions and is particularly beneficial in settings that demand adaptability, such as data centers and 5G networks. Programmable data planes further extend user control by allowing customization of packet forwarding, protocol headers, and packet behaviors, capabilities that were once limited to equipment vendors.

This evolution has led to new programming languages for data plane programmability, including P4 [4], [5] and NPL [6]. In particular, the Protocol-Independent Packet Processor (P4) enhances network device flexibility and introduces challenges, such as a lack of support for modularity and advanced programming constructs. This absence of support for modularity limits the potential for code reusability and flexibility in large-scale network programs. To address

The associate editor coordinating the review of this manuscript and approving it for publication was Maurizio Casoni.

these limitations, modularity is essential, allowing the reuse of protocol libraries, composition of data plane pipelines, and secure vendor-customer collaboration without extensive custom syntax.

Despite its necessity, achieving modularity in P4 directly within its compiler has been a challenge. Previous solutions, such as daPIPE [9], P4 Weaver [10], P4-Ansible [11], μP4 [12], and Lyra [13], indirectly introduced modularity and lacked essential features such as backward compatibility, simple syntax, or compatibility with important classes of use cases. Backward compatibility allows current P4 codes to be directly used with that tool. A direct approach to modularity within the P4 language can address these gaps by supporting the following:

- **Code reusability**: Enables developers to modularly reuse existing P4 code, reducing redundancy and simplifying program development.
- **Data plane pipeline composability**: Allows network managers, who may not have software development skills, to incorporate various functionalities into the data plane by selecting different modules within a data plane pipeline.
- **Vendor-customer collaboration framework**: Enables customers to integrate their code with vendor-provided code to enhance functionality. Since vendors may be reluctant to share proprietary code, this setup allows customers to send their code to the vendor for integration without direct access to the vendor's code.

Although modularity offers numerous benefits and is essential for network programmers, achieving it directly in the P4 language and its compiler, especially for large programs such as those for switches [7] or SRv6 [8], is challenging. Previous attempts to introduce modularity in the P4 language have relied on indirect approaches due to these challenges. However, despite presenting a learning curve, indirect methods do not offer the full benefits of modularity, such as backward compatibility with existing P4 code and straightforward extensibility. Additionally, these approaches do not address all classes of use cases that we consider here.

Previous work on modularity in the P4 language does not address specific key scenarios. In contrast, P4Muse provides backward compatibility with existing P4 code without introducing new syntax or annotation mechanisms to support code reusability, data plane pipeline composability, and a vendor-customer framework.

To address these challenges, P4Muse is introduced as an extension of an open-source P4 compiler [14] that supports P4 modularity. It enhances modularity across key components, such as the parser, header, struct, ingress, egress, and deparser. It does so without modifying the P4 syntax. It merges the base code, which provides foundational functionality, such as IPv4 processing, with additional code, which we call extension code, to introduce new capabilities, such as IPv6 support. The contributions of this paper are summarized as follows:

- P4Muse introduces a novel extension to the P4 compiler that enhances modularity through objects. This extension enables developers to write modular P4 programs that take advantage of reusable protocol and application libraries, addressing a long-standing challenge in the P4 community.
- A key feature of P4Muse is the automatic merging of P4 modules, which facilitates the integration of various functionalities in network programming. We define and evaluate a three-level modularity framework: construct-level modularity for extensible and incremental programming, protocol-level modularity for seamless protocol integration, and application-level modularity for composing complex network functions. These modularity levels improve the flexibility and scalability of P4-based systems.
- The proposed solutions are validated by implementing three classes of use cases that demonstrate code reusability, data plane pipeline composability, and vendor-customer compatibility.
- Six case studies are implemented to explore the impact of modularity in merging multiple P4 programs. These studies focus on protocol and application-level integration, security improvements, performance optimization, and extensibility. The effectiveness and versatility of P4Muse are evaluated using the V1Model architecture and the BMv2 software switch.

The remainder of this paper is organized as follows. We summarize the related work on P4 modularity in subsection I-A and the related work using p4 modularity in subsection I-B. The goals and challenges of implementing P4 modularity are described in section II. The framework of P4Muse, including its compiler and interface, is reviewed in section III. The methodology and levels of modularity are presented in section IV and section V, respectively. We present classes of use cases in section VI, followed by case studies and evaluation in section VII and section VIII. The challenges of modularity and the current limitations of P4 are discussed in section IX, while the limitations of our work and directions for future research are covered in section X. Finally, the paper concludes in section XI.

A. RELATED WORK ON P4 MODULARITY

This section reviews various approaches to enhancing modularity in P4 programming, focusing on SDN automation, data plane composition, compiler design techniques, and incremental programming. We discuss the limitations of existing solutions to achieve true modularity and code reuse in P4.

1) SDN AUTOMATION

Software-defined networking (SDN) automation facilitates the orchestration of modular components to create integrated P4 programs for both control and data planes, addressing the shortage of experts skilled in both areas.

ClickP4 [18] is an approach that enables developers to create P4 modules following a specified methodology. However, these modules must be manually integrated into ClickP4's source code to orchestrate various components and construct a more extensive P4 program. In the context of automating modular and programmable control and data planes [19], [20], research has explored orchestration tools that employ P4-based modules in conjunction with control applications within the Open Network Operating System (ONOS) controller. For modular switches in programmable forwarding planes [21], the μP4 language has been used to develop the One Big Switch (OBS) abstraction, which optimizes and reduces resource utilization.

Although these approaches represent a step forward in modularity for P4, they have notable limitations. Unlike modern programming languages, they require manual integration of new modules, and developers cannot reuse preexisting P4 code, limiting flexibility and code reusability.

2) DATA PLANE COMPOSITION AND VIRTUALIZATION

Data plane composition and virtualization research efforts have explored modularity through virtualization approaches [22].

HyPer4 [23] was the first study to virtualize P4 programs using a P4-hypervisor. HyPer4 also introduced a versatile table capable of supporting various types of matching, such as exact and ternary matches, and implemented primitive actions. Building on this, HyperV [24] and later HyperVDP [25] developed a unique parsing structure that incorporates a description header (DH) containing additional information, including the program ID and header length. HyperVDP further optimized the match-action stage, reducing the number of steps required in action processing.

P4Bricks [26] introduced a method for creating a single data plane configuration file by merging multiple compiled data plane files. This approach combines parallel and sequential composition, distinguishing it from the method used in P4Visor [27]. P4VBox [28], unlike HyPer4 and HyperVDP, reengineers parallel lookup modules using Hardware Description Language (HDL) to support multiple programs simultaneously on the data plane.

P4Visor [27], [29] focuses on compiler-level virtualization for testing network applications, merging high-level intermediate representations (HLIR) of P4 programs using a frontend compiler. However, P4Visor's modular development capabilities are limited, and it only supports P4 version $P4_{14}$, not $P4_{16}$. PRIME [31], [32] expands this line of research by introducing parsing and combining techniques for various P4 programs, ensuring consistent traffic steering through packet recirculation. However, data plane composition and virtualization methods require additional tables to handle virtualization or recirculation, which consumes additional hardware resources and increases latency.

3) COMPILER DESIGN

Efforts in compiler design have introduced intermediary compilers that precede the P4 compiler, translating outputs into P4, NPL, or low-level code to address P4's limitations.

μP4 introduces a modular framework ahead of the P4 compiler, allowing programmers to write target-agnostic, modular data plane code that can be converted to P4. However, μP4 lacks backward compatibility with existing P4 code and requires learning new syntax, which presents a learning curve. Lyra shares μP4's objective but takes a different approach, creating a unified pipeline abstraction to address the challenges of portability, extensibility, and compatibility in data plane programming. In P4All [33], an elastic data structure is introduced that dynamically adjusts to hardware resources, supporting code reuse across diverse environments. While innovative, it does not extend to reusing elements like parser states.

However, these approaches have notable drawbacks. They often require programmers to learn a new syntax, and most lack backward compatibility with existing P4 code. By not directly utilizing the P4 language, these compilers also limit the community's ability to build upon these works within P4 itself.

4) INCREMENTAL PROGRAMMING

Incremental programming enables the integration of customer-specific code into a vendor's stable codebase or code modification through runtime programmability. Due to the vendor's deeper understanding of the hardware, vendor-produced code is typically more reliable and less error-prone than customer-generated code.

In daPIPE, incremental programming facilitates the merging of vendor and customer code by establishing guidelines such as prioritizing vendor code, maintaining control plane integrity, and protecting the intellectual property of vendor contributions. P4 Weaver builds on daPIPE's methods, enhancing incremental programming by providing controlled code isolation and sequencing. While daPIPE uses a GUI to display editable sections of the vendor's code, P4 Weaver employs an annotation system in both customer and vendor code. Both approaches use a client/server architecture to safeguard the vendor's intellectual property.

P4Ansible introduces incremental programming features by adding 'override', 'super', and 'default' keywords to the Bison parser of the P4 compiler. This allows reuse of base code, parser states, controls, structs, and headers. Although P4Ansible is intended for incremental programming, it lacks a clear method for customers to identify reused sections of the base code, and its incremental programming capabilities remain incomplete.

B. RELATED WORKS USING P4 MODULARITY

1) SERVICE FUNCTION CHAIN (SFC) FRAMEWORKS

Service function chain (SFC) frameworks [34], [35], [36], [37] facilitate the flexible composition of network

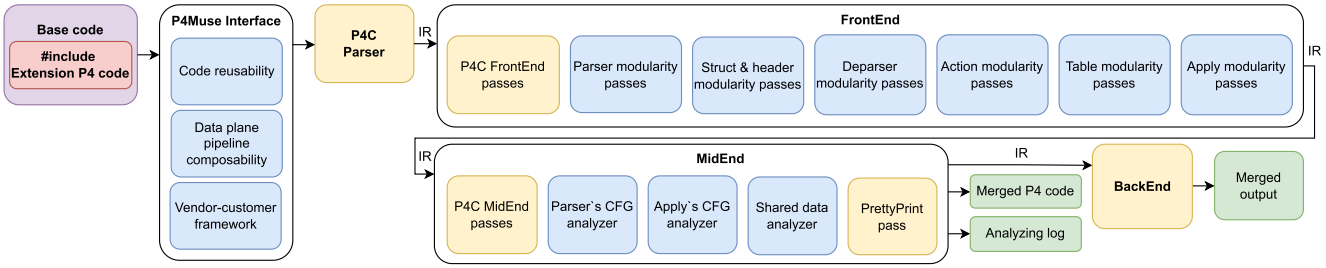


FIGURE 1. Overview of the P4Muse compiler showing its modular interface options and compiler stages.

functions, allowing dynamic chaining of services while maintaining performance. These frameworks underscore the importance of modularity in the P4 language for enabling SFCs in a standardized way without introducing additional latency.

2) RUNTIME PROGRAMMABILITY

Research on runtime programmability [38], [39], [40], [41], [42] addresses the limitations of current chip architectures and programming languages in supporting in-service updates, such as dynamically adding or removing protocols and functions. These works emphasize the need for modularity in the P4 language to simplify updates and support efficient runtime modifications.

II. GOALS AND CHALLENGES FOR IMPLEMENTING P4 MODULARITY

A. DESIGN GOALS OF P4Muse

1) OBJECT SYSTEM IN THE P4 LANGUAGE

The goal of P4Muse is to bring modularity in P4 programming by introducing objects for each protocol/parser state, collecting relevant elements such as the parser state, header and struct data types, and deparser state. Similarly, objects are defined for tables, including their keys, values, and actions, as well as for each apply sub-block. Modularity is achieved by comparing these objects across the base and extension codes, facilitating the merging of P4 programs. In section X, we will discuss potential future enhancements for referencing these objects.

2) MODULARITY WITHOUT NEW SYNTAX OR ANNOTATIONS

Since P4 is a relatively new language, learning additional syntax or annotations to support modularity can add significant complexity for developers. Therefore, achieving modularity with minimal new syntax, or ideally without any additional syntax, is a key goal.

3) BACKWARD COMPATIBILITY

Backward compatibility is essential, as many organizations and academic institutions have invested significant time and resources in developing P4 code. A lack of backward compatibility would limit the ability to introduce modularity into existing P4 codebases.

4) ADDRESSING P4 LANGUAGE LIMITATIONS FOR MODULARITY

To our knowledge, no prior work directly introduces modularity across all aspects of the P4 language using its compiler. In doing so, P4Muse identifies and addresses the limitations within P4 that hinder modularity, paving the way for concepts similar to object-oriented programming in future developments.

B. CHALLENGES IN ADAPTING P4 FOR MODULARITY

1) MONOLITHIC STRUCTURE OF P4 PROGRAMS

The P4 programming language employs a monolithic structure, lacking the modularity found in object-oriented programming languages. Furthermore, P4 lacks mechanisms to maintain connections between different code components. For example, there is no standardized way to link an IP header to its corresponding state in the parser. Developing a modular framework for P4 is essential to ensure these connections are maintained.

2) NAMING INCONSISTENCIES

A significant challenge in achieving modularity is the inconsistency in naming conventions across extension and base codes. Since different programmers may use varying names for similar components, reliably comparing and integrating code elements becomes difficult. This inconsistency complicates establishing a consistent and accurate comparison for modularity.

III. THE P4Muse CLASSES OF USE CASES

This section provides an overview of the P4Muse classes of use cases, focusing on its modular interface options and the detailed architecture of the P4Muse compiler. Figure 1 illustrates the structure and modular functionality of the P4Muse compiler, serving as a visual guide to the different stages and passes within the P4Muse framework.

A. P4Muse INTERFACE

The P4Muse interface provides users with three modular options for network programming: code reusability, data plane pipeline composability, and vendor-customer compatibility. These options enable flexibility in developing and reusing modular P4 components.

B. P4Muse COMPILER

Figure 1 presents the structure of the P4Muse compiler, highlighting new custom passes (shown in blue) that integrate with the previously existing stages of the P4C compiler (shown in yellow). The P4Muse framework enables modularity by including extension code—such as additional protocols or processing modules—directly into the base P4 code via the #include directive. This directive incorporates modular P4 files, functioning similarly to modular file inclusion in other languages, allowing programmers to add specific components without modifying the core codebase. During the MidEnd phase, the PrettyPrint pass produces the final merged P4 code, generating both compiler and P4 outputs. P4Muse receives the regular P4C intermediate representation (IR) and transforms it into a merged IR while maintaining the structure of the standard P4C IR. Unlike a transcompiler, which translates one language into another, or a preprocessor, which performs text substitution before compilation, P4Muse directly extends the P4C compiler with custom passes that operate on the IR of P4 code. If the backend were removed and P4Muse were used solely to generate the merged P4 code without producing the switch configuration output, it would function as a transcompiler. Additionally, P4Muse provides analysis logs to assist programmers in debugging and ensuring the accurate merging of modular data and control plane components.

IV. METHODOLOGY FOR IMPLEMENTING THE PROPOSED P4 MODULARITY

A. METHODOLOGY OF PARSER

1) THE STRUCTURE OF PARSER IN P4 LANGUAGE

In the P4 language, a parser operates as a state machine, beginning with an initial state called 'start' and terminating in one of two possible states: 'accept' or 'reject'. The 'start' state initiates the parsing process, while the terminal states indicate the result—'accept' signifies successful parsing, and 'reject' denotes a parsing failure. Although the 'start' state is integral to the parsing sequence, the 'accept' and 'reject' states are logically external and serve as endpoints rather than user-defined states within the parser's primary logic.

2) NAME STANDARDIZATION FOR PARSER MERGING

The first step in merging the two parsers involves matching and standardizing the names and types of the inputs and outputs, found in orange code lines as shown in Figure 2, as different programmers may have used inconsistent naming conventions. Specifically, we modify the names of variables and types in the second parser tree to match those in the first parser, ensuring consistency for ''packet,'' ''headers,'' ''metadata,'' and ''standard metadata''. For example, if the second parameter in the extension code was ''out myHeaders mh'', it would rename type ''myHeaders'' and variable ''mh'' to match those in the second parameter of base code. This renaming process extends throughout the entire codebase, including program component types, as well as control blocks like 'VerifyChecksum,' 'Ingress,' 'Egress,' 'ComputeChecksum,' and 'Deparser.'

In Figure 2, which shows the codes for 'ExtensionParserFirewall', 'BaseParserAdvancedTunnel', and the merged parser, any discrepancies in variable names and types such as ''packet,'' ''headers hdr,'' ''metadata meta,'' and ''standard_metadata_t standard_metadata'' are resolved. This standardization ensures that the merged parser operates smoothly by maintaining consistent names across all program components.

3) THE DATA COLLECTING ALGORITHM

To facilitate merging, we first create an object for each parser state (or protocol), collecting details such as the parser state node, transition value, extracted header, child transition values, header data type, and struct data type it uses. During additional passes, we finalize the collection of corresponding header nodes, struct nodes, and deparser state nodes. Data from each finite-state machine (FSM) in the two parser trees is collected state-by-state to enable comparison between objects in the extension and base codes.

Unlike prior approaches such as PRIME and its improved version [31], [32], which relied on state names, we faced the challenge that state names alone could not be used for comparison, as different programmers may use different names for each state. Instead, we used transition values, extracted headers, and child transitions to compare FSM states. For protocols like IPv4 and IPv6, these transition values correspond to protocol numbers published by the Internet Assigned Numbers Authority (IANA) [43]. Therefore, states with different names but the same transition value (i.e., protocol field value), extracted header, and child transitions are merged.

Another challenge is that a state's transition value is always located in its parent state in the P4 language. To address this, in each code we examined each state's name in its parent state and compared them to identify the transition value associated with each state. This process allowed us to accurately compare the two parser trees based on transition values, with exceptions for the start, reject, verify, and accept states defined by the P4 specification. For these states, we relied on state names rather than transition values. The top state without a transition value in each tree (typically Ethernet) was considered equivalent in both trees.

Figure 3 presents the parser graphs corresponding to Figure 2. Key nodes include 'start,' 'parse_ethernet,' 'parse_myTunnel' with transition value ''16w0 × 1212,'' 'parse_ipv4' with transition value ''16w0 × 800,'' and terminal states 'accept' and 'reject' for the base code, as well as 'start,' 'parse_ethernet,' 'parse_ipv4' with transition value ''16w0 × 800,'' 'tcp' with transition value ''8w6,'' and terminal states 'accept' and 'reject' for the extension code. In P4 notation, 'w' and 's' indicate an unsigned numeric and a signed numeric value, respectively, with a specified bit width on the left and value on the right of that letter.

```
#include ExtensionCode.p4

parser BaseParserAdvancedTunnel (packet_in packet,
    out headers hdr,
    inout metadata meta,
    inout standard_metadata_t standard_metadata) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_MYTUNNEL: parse_myTunnel;
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_myTunnel {
        packet.extract(hdr.myTunnel);
        transition select(hdr.myTunnel.proto_id) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

```
parser ExtensionParserFirewall (packet_in packet,
    out headers hdr,
    inout metadata meta,
    inout standard_metadata_t standard_metadata {
    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 select(hdr.ipv4.protocol) {
            TYPE_TCP: tcp;
            default: accept;
        }
    }
    state tcp {
        packet.extract(hdr.tcp);
        transition accept;
    }
}
```

```
parser MergedParser (packet_in packet,
    out headers hdr,
    inout metadata meta,
    inout standard_metadata_t standard_metadata) {
    state start {
        transition parse_ethernet;
    }
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_MYTUNNEL: parse_myTunnel;
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_myTunnel {
        packet.extract(hdr.myTunnel);
        transition select(hdr.myTunnel.proto_id) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol){
            TYPE_TCP: tcp;
            default: accept;
        }
    }
    state tcp {
        packet.extract(hdr.tcp);
        transition accept;
    }
}
```

FIGURE 2. Parser blocks of advanced tunnel, firewall from the P4 tutorials [15], along with the merged P4 code developed in this work.

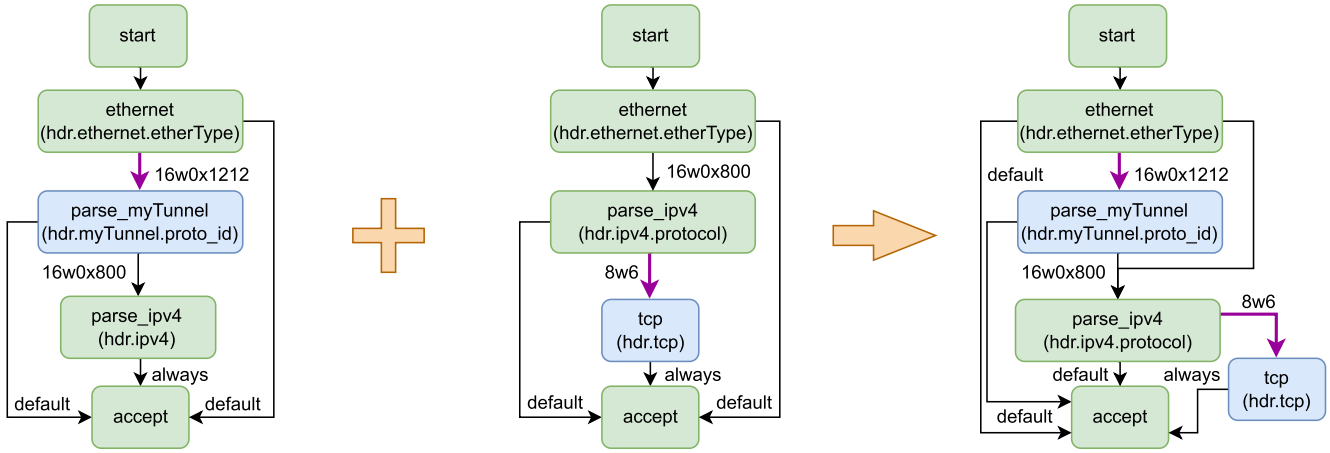


FIGURE 3. Parser graphs of an advanced tunnel, a firewall, and the corresponding merged P4 codes. Note that the firewall's acceptance or rejection of packets occurs in the match-action unit, not in the parser stage.

4) THE COMPARISON AND MERGING ALGORITHM

In the P4 source, we consider the base code as the main code when merging two codes, ensuring minimal merging while prioritizing the base code over the extension code. We identify elements present in the extension code but missing from the base code and integrate them step by step into the base code to produce the merged code. We use header structures and transition select values to guide this merging process within the parser. The transition values usually come from standard protocol numbers and will thus match between codes, but codes written independently using non-standard numbers could exist; if both codes use the same number for different protocols, the algorithm will report an error indicating that the parsers are incompatible unless both protocols also happen to have matching header structures. The merging algorithm relies entirely on the input code because we achieve modularity without introducing new syntax or annotations. If two programmers use different transition values corresponding to two similar headers, the algorithm treats them as distinct headers. Given that the transition values were explicitly assigned by the programmers, it is more plausible that they intended to use two similar header

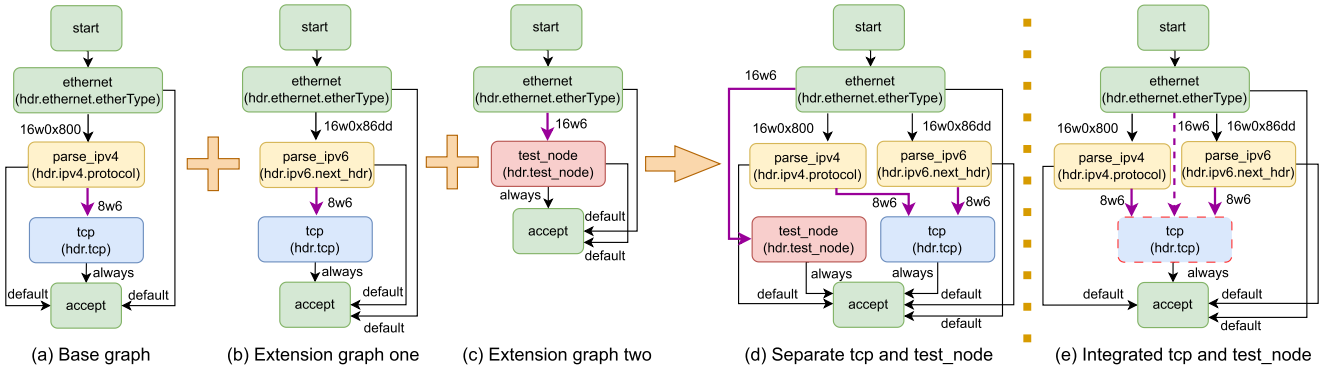


FIGURE 4. Comparison between integrated and separate parsing strategies with different graphs.

structures in separate contexts rather than being unaware of each other's implementations. We define three types of nodes that determine how elements from the extension parser should be incorporated:

- **Equivalent Node:** A node is considered equivalent if it satisfies the following conditions: (1) it has the same transition value, corresponding to a part of the header extracted by its parent state; (2) it extracts the same header as the corresponding state; and (3) it possesses the same child transitions as the corresponding state.
- **Mergeable Node:** A node is considered mergeable if it satisfies the same conditions as an equivalent node for the transition value and header extraction, but differs in its child transitions.
- **Different Node:** A node is considered different if it cannot satisfy the first and second conditions as an equivalent node for the transition value and header extraction.

If Equivalent or Mergeable nodes have a Different node as child for the same transition value, they are different because they extract a different header. In this case, the parsers are said to be **incompatible** and cannot be directly merged by this algorithm.

We will illustrate with an example why we do not use more restrictive merging rules comparing parents or full path to the root node. In Figure 4, the base graph (a) has a `tcp` node as child of `parse_ipv4`, while extension code one (b) has the `tcp` node as child of `parse_ipv6`. The TCP protocol is the same in IPv4 and IPv6, having the same header fields to extract, thus it makes sense to merge them even though they have different parents and transitions from the root.

On the other hand, the `test_node` in extension code two (c) and the `tcp` node have similar transition values of `"8w6"` and `"16w6"` (the value 6 but on different number of bits 8 or 16). Still, because they extract different headers, they should be kept separate, as illustrated in graph (d) of Figure 4. If the header extracted by `test_node` is identical to the header extracted by `tcp`, and both share the same transition value, the merged graph will integrate the `tcp` and `test_node` states.

When comparing transition values, we consider both the transition value itself and its length to distinguish between `"8w6"` and `"16w6"`. However, we do not compare the bit position of the transition value. Otherwise, TCP would not have been merged as in Figure 4(d), because for different types of headers, the transition value may be located in different parts of the header.

When comparing extracted headers, we check the total size of the header, as well as the size and type of each subpart. If a mismatch occurs in any subpart, we attempt to merge consecutive subparts to achieve a match. In algorithm 1, three main situations are considered when comparing the base and extension parsers using three defined nodes.

- **Comparison of Equivalent Nodes:** In this case, both the base and extension parsers have a state with the same transition value, extracted header, and identical child states in "transition select". The algorithm retains only one instance of this state and its children's transitions in the merged parser.
- **Comparison of Different Nodes:** If a state exists in the base parser but not in the extension parser, the algorithm does nothing because it considers the base code as the main code. For example, in Figure 2, ttparse_myTunnel is in the base parser but not the extension parser.
  If a state exists in the extension parser but not in the base parser, it is added to the final parser. The algorithm identifies this state and inserts it into the base parser. This ensures that unique states from the extension parser are appropriately incorporated into the merged parser structure. For example, in Figure 2, the `tcp` state exists in the extension parser but not in the base parser. The algorithm identifies `tcp` as a state missing from the base parser. Finally, it merges the `tcp` state as a child in the base parser.
  Regarding the children of a merged node from the extension parser, the merging process also incorporates their children's transitions in the "transition select". If a child node does not exist in the base parser, it is treated as a new state and will be merged in subsequent comparisons. If the child node already exists in the base

parser, the merged extension parser reuses it without introducing a new state.

- **Comparison of Mergeable Nodes:** When a state in both parsers has the same transition value and extracted header but different child transitions in "transition select", the algorithm merges the children's transitions from the extension parser that do not exist in the base parser. For instance, in Figure 2, the `TYPE_TCP: tcp` transition is a child transition of `parse_ipv4` in the extension parser but not in the base parser, so `TYPE_TCP: tcp` is added as a child transition of `parse_ipv4` in the merged parser.
  If a node from a merged transition in "transition select" already exists in the base parser, the merged parser reuse it without introducing a duplicate state. Otherwise, it is treated as a new state and merged in subsequent comparisons. For example, in the case of the `TYPE_TCP: tcp` transition, the algorithm merges the `tcp` state.

In algorithm 1, 'compareParser' compares two parser trees, `baseObjTree` and `extObjTree`, and flags nodes in the extension tree or the child's transition that need to be merged. It consists of two main functions, `compChilds` and `compParser`.

- **Function `compareParser` (lines 1-9)**: This is the main function that iterates over all states in `extObjTree` and compares them with `baseObjTree` to identify which states in `extObjTree` require merging. The function first creates `baseStates`, a mapping that associates each state's transition value with its corresponding state (line 2). This allows for efficient lookup and reduces computational complexity. Then, it iterates through each state in `extObjTree`, storing its transition value in `extTransition` to improve comparison efficiency.
  - **Situation 3**: If `extState` does not exist in `baseObjTree`, it means the state is unique to `extObjTree`. The algorithm flags `extState` for merging (line 9), ensuring that `extState` and its children's transitions will be added to the final merged parser tree.
  - **Situation 2**: If `extState` exists in `baseObjTree` but has different child transitions, the `compareChildTransitions` function is called to identify specific child transitions missing in `baseState` (line 7). The function iterates through the child transitions of `extState` and marks those not present in `baseState` for merging (lines 10-14).
  - **Situation 1**: If an `extState` in `extObjTree` has an equivalent transition value and extracted header in `baseObjTree` and their children's transition sets are identical, no merging is required. This situation is identified in the else condition of the `if` statement in line 11, where the algorithm confirms that the states and their child's transitions match exactly.
- **Function `CompChildTrans` (lines 10-14)**: This function compares the child transitions of two states, `baseState` and `extState`, to determine which child transitions from `extState` do not exist in `baseState` and should be marked for merging. It first checks if the child transition sets of `baseState` and `extState` are different (line 10). If they differ, it iterates over the transitions in `extState` that are not present in `baseState` and marks the states and the transitions for merging (lines 11-14).
- **Function `childTransitionVals` (lines 15-16)**: This function returns the set of transition values for all child states of a given `state`. It is used to determine whether the child transition sets of two states are equivalent.
- **Function `compareHeader`**: This function ensures that the headers being compared have the same total size, as well as identical sizes and types for each subpart. The function examines each subpart one by one. Suppose the sizes of individual subparts do not match. In that case, the algorithm attempts to merge consecutive subparts, provided that their combined size exactly equals the corresponding subpart in the other header (e.g., merging subparts of sizes 2, 2, 2, and 2 to match a subpart of size 8). Merging is allowed only if all merged subparts share the same type. The algorithm accumulates the sizes of consecutive subparts until one of the following conditions is met. The comparison is valid if the accumulated size matches the corresponding subpart's size in the other header. Partial mismatches are not allowed, meaning that merging subparts of sizes 4 and 5 to match a subpart of size 8 is considered invalid. The headers are considered incompatible if neither exact matching nor merging results in a successful comparison. All subparts must be fully matched by the end of the comparison for the headers to be considered equivalent.
  This algorithm enables an efficient merging process by using transition values, extracted headers, and children's transitions to compare states. This is necessary because state names may differ between the base and extension parsers. Transition values and children's transitions allow accurate identification and merging of equivalent states, making it well-suited for modular parser integration.

In algorithm 2, the function constructs a merged parser by integrating states and child transitions from the extension object tree `extObjTree`, which were flagged by algorithm 1. We also use the `preorder` function from the P4 compiler, which is part of the visitor pattern and allows actions to be performed on a node before recursively visiting its children during IR tree traversal. The merging process is divided into two main parts:

Algorithm 1 Parser Comparison Algorithm

```
Input: baseObjTree, extObjTree
Output: Updated extObjTree
1  Function compareParser(baseObjTree, extObjTree):
2      baseStates ← map of (state.transitionVal → state) foreach state in baseObjTree;
3      foreach extState in extObjTree do
4          extTransition ← extState.transitionVal;
5          if extTransition in baseStates then
6              if compareHeader (baseStates[extTransition], extState) then
7                  compareChildTransitions (baseStates[extTransition], extState);
8          else
9              // Situation 3: Different Nodes
               extState.requiresMerging ← true;
10 Function compareChildTransitions (baseState, extState):
11     if childTransitionVals(baseState) ≠ childTransitionVals(extState) then
           // Situation 2: Mergeable Nodes
12         extState.childTransitionsRequireMerging ← true;
13         foreach transition in extState.transitions not in baseState.transitions do
14             transition.requiresMerging ← true;
       // Else: Situation 1: Equivalent Nodes
15 Function childTransitionVals(state):
16     return set of child.transitionVal for each child in state.childs;
```

Algorithm 2 Parser Merging Algorithm

```
Input: IR::parserState node, extObjTree, baseObjTree
Output: Updated base parser
1  extSetAddState ← set of extObj for each extObj in extObjTree if (extObj.requiresMerging);
2  extMapMergeTransition map of (extObj.transitionVal → vector of extObj.transition) (for each extObj in extObjTree if (extObj.childTransitionsRequireMerging)) and (for each extObj.transition in extObj.transitions if (extObj.transition.requiresMerging)) ;
3  processedAddState ← false;
4  processedMergeTransition map of (baseObj.transitionVal → false) (for each baseObj in baseObjTree);
5  Function preorder(IR::parserState node by reference):
       // Adding whole states
6      if (not processedAddState) and (getTransitionVal not in extSetMergeTransition) then
7          parserStateVector ← IR::vector of node;
8          if extObj in extSetAddState then
9              parserStateVector.append(extObj);
10         processedAddState ← true;
11         return parserStateVector;
       // Merging child transitions
12     if not processedMergeTransition[getTransitionVal] then
13         if getTransitionVal in extSetMergeTransition then
14             selectCaseVector ← IR::vector of existingCase for each existingCase in node.selectExpression.selectCases;
15             selectCaseVector.append(extObj.transition) for each extObj.transition in extObj.transitions;
16             node.selectExpression ← new IR::selectExpression( node.selectExpression.srcInfo, node.selectExpression.select, selectCaseVector);
17         processedMergeTransition[getTransitionVal] ← true;
18     return node;
```

- **Adding Whole States (lines 5-11)**: If the current node has not yet been processed for whole-state addition, it is checked against `extSetAddState`. If it belongs to this set, both the base state and its corresponding extension state (`extObj`) are added to `parserStateVector`. The function then marks `processedAddState` as `true` and returns `parserStateVector`. The algorithm ensures that each preorder execution handles only one task at a time, either adding a whole state or merging child transitions.
- **Merging Child Transitions (lines 12-18)**: If the node requires transition merging (`getTransitionVal` is present in `extMapMergeTransition`), the algorithm first retrieves `selectCases` from the base node's `selectExpression`, representing existing child transitions. It then iterates over each child transition in `extObj` and, if it requires merging, appends it to `selectCaseVector`. Finally, it updates the base node's `selectExpression` with the expanded `selectCaseVector`, ensuring that new transitions from the extension are correctly merged.

5) SUB-PARSER COMPARISON AND MERGING ALGORITHM

In scenarios where a sub-parser (callee parser) is invoked within another parser (caller parser), we analyze the inlined structure produced by the standard inlining P4C passes. After applying P4C's inlining passes, we obtain inlined graphs for both the base and extension parsers. As shown in Figure 5, the inlining process generates graph 'd' from graphs 'b' and 'c'. Our passes then merge graph 'a' with the inlined graph 'd', producing graph 'e'. The states "ethernet" and "ethernetParser_start" as the sub-parser's start are treated as mergeable nodes, as both extract the same ethernet header and neither has a transition value. Since graph 'a' is the base, the algorithm considers the default transition of "ethernet"

as ''accept'', since the algorithm gives priority to the base (shown on Figure 5e as blue arrow). However, if graph 'd' were the base, the default transition would instead be considered as ''start_0'' (shown on Figure 5e as red arrow). Additionally, ''start_0'', which represents the inlined sub-parser's accept state, shares the same transition value (16w0× 86DD) as ''parse_ipv6''. However, if ''parse_ipv6'' appears in graph 'a', the algorithm does not consider it mergeable with ''start_0'' because the latter does not extract any header. Instead, ''parse_ipv6'' from graph 'a' is matched with its counterpart in graph 'd' as equivalent nodes, based on their similar transition value and header extraction. This illustrates the importance of carefully analyzing mergeable and equivalent nodes in the start and accept states of sub-parsers.

6) PARSER VERIFICATION

To ensure the correctness of the merged parser, a static analyzer rechecks each state's transition values and extracted headers and verifies that the merging process aligns with the expected structure. This verification process is essential to ensure the parser is deterministic (no ambiguity in state transitions for a given input) and loop-free (no infinite loops during parsing). The static analyzer performs checks to confirm that every possible parsing path terminates correctly and that each input leads to a single, unambiguous outcome, ensuring the reliable operation of the merged parser.

B. METHODOLOGY FOR THE STRUCT AND HEADER DATA TYPE

1) MAPPING PARSER STATES TO STRUCTS AND HEADERS

Each parser state object includes information about the struct and header data types it uses. The algorithm analyzes ''packet.extract()'' calls within that state to identify the struct headers associated with each parser state. For instance, if a parser state extracts ''hdr.myTunnel'', we can determine that it uses the ''myTunnel_t'' header.

This mapping process allows the algorithm to associate each parser state with its corresponding struct and header types. The header name and the struct that references this header are stored as attributes of the parser state object, ensuring clear associations between states and their data types.

In Figure 6, for example, the ''packet.extract(hdr.myTunnel)'' call in the ''parse_myTunnel'' state indicates the use of ''myTunnel_t''. From the declaration ''out headers hdr'' and the extraction of ''hdr.myTunnel'', our algorithm can identify ''myTunnel_t myTunnel'' as part of the merged 'headers' struct. This way, ''myTunnel_t'' is recognized as a merged header type within the final parser structure.

2) NAME STANDARDIZATION FOR STRUCTS AND HEADERS

As shown in Figure 2, two ''parse_ipv4'' parser states exist, which may refer to different types of IPv4 headers. To resolve these differences, we apply the 'compareHeader' function described in algorithm 1, which determines whether headers are equivalent by merging consecutive subparts when their combined sizes match exactly. As shown in Figure 7, when one IPv4 header defines the 'diffServ' field as a single 8-bit subpart, and another splits it into two consecutive subparts—'diffServ' (6 bits) and 'ecn' (2 bits)—we preserve the header with finer granularity (i.e., the two-part version). Throughout the source code where the original 8-bit 'diffServ' field was used, we explicitly concatenate the two subparts (6 + 2 bits) and replace references to the original 8-bit field with this concatenated version. This method standardizes header definitions, ensuring consistency across the merged code while preserving functional correctness and flexibility. We ensure that any concatenation of subparts respects the required size constraints for the target hardware because we assume that the input headers are already compliant with hardware requirements.

3) THE COMPARISON AND MERGING ALGORITHM

After identifying and standardizing the header and struct names, we combine the structs of the extension code by merging their corresponding parser states. By following the parser's Finite State Machine (FSM) order, the algorithm identifies which headers and structs should be unified, preserving the logic and sequence of each parsing path within the merged structure.

4) STRUCT AND HEADER VERIFICATION

We implemented a static analyzer in the compiler to ensure the merged code's correctness. This analyzer verifies the order of the parser's FSM, the merged states and their corresponding structs and headers. Additionally, it verifies that concatenation-based replacements for header subparts are correctly applied when differing protocol formats, such as those illustrated in Figure 7, are present.

C. METHODOLOGY FOR THE DEPARSER

1) MAPPING PARSER STATES TO DEPARSER EMISSIONS

To maintain consistency, the deparser must follow the same header emission order as the parser. For example, the system should not emit an ''IPv4'' header before a ''UDP'' header, as this could lead to out-of-order reads/writes at the next network hop. We follow the FSM order of both the extension and base parsers to determine the correct sequence for each 'emit' statement.

In Figure 6, the deparser sequence is aligned with the merged parser states. For instance, 'packet.emit(hdr.myTunnel)' is determined by identifying 'myTunnel' as a shared header in both the parser and deparser states. It should be noted that in the P4 language, headers not marked as valid are not emitted.

2) STANDARDIZING DEPARSER NAMES

The first step in merging deparsers involves aligning the input and output names of the two deparsers, as different

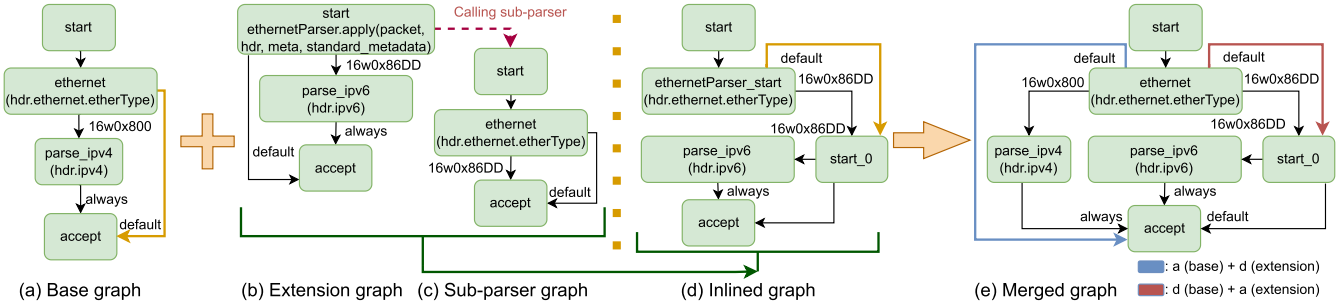


FIGURE 5. Parser graphs transformation, including sub-parser inlining and final merging.

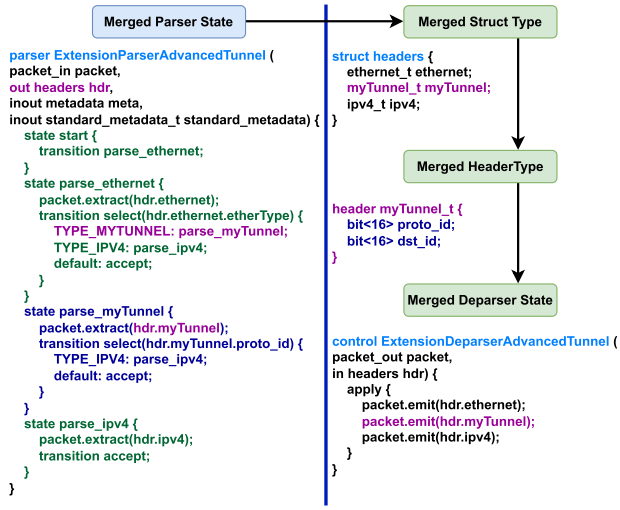


FIGURE 6. Parser, struct, header, and deparser blocks of advanced tunnel P4 code.

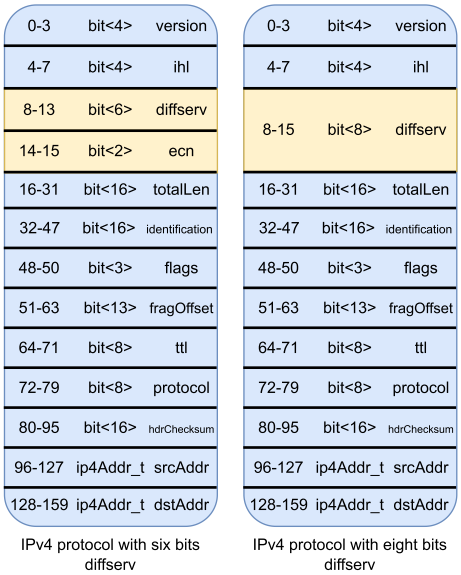


FIGURE 7. Comparison of IPv4 header formats with two kinds of DiffServ field lengths.

programmers may have used varying names. We adjust the names in the second deparser to match those in the first, ensuring consistency across packets and headers.

3) THE COMPARISON AND MERGING ALGORITHM APPLIED TO THE DEPARSER

We merge the deparser states corresponding to each merged parser state, preserving the parser order while considering the deparser orders of both the extension and base codes. We use an object-based approach to structure the merged deparser as other parts. In this approach, each deparser state from the extension code is merged with its corresponding parent state in the base deparser by comparing the parent states in both deparsers.

4) DEPARSER VERIFICATION

Our static analyzer verifies the correctness of the merged deparser by checking the order of parser states and deparser sequences from both the extension and base codes. This ensures that the merged deparser emits headers in the correct order, maintaining data consistency throughout the network pipeline.

D. METHODOLOGY FOR THE CONTROL

1) THE DATA COLLECTING ALGORITHM

We collect data from the control tables in both the base and extension codes. Tables are compared based on their keys, maximum sizes, and actions. If two tables share the same key, size, and match-action method, they can be merged into a single table. Additionally, we compare each action node referenced in the tables to verify whether the actions are identical. Also, we compare apply sub-blocks including if-condition and applying tables. To avoid redundant comparisons, the components of each action, table, and apply sub-block are evaluated until the first unmatched node is found.

2) STANDARDIZING CONTROL NAMES

The initial step in merging controls is to standardize the input and output names in the control sections, as the original programmers may have used different conventions. We adjust the input and output names in the second control section to match those in the first, ensuring consistency across local metadata, standard metadata, and headers.

3) THE COMPARISON AND MERGING ALGORITHM APPLIED TO CONTROL

To ensure an efficient and structured merging process, we compare actions, tables, and apply sub-blocks using

a recursive node-by-node approach. The merging process consists of three main steps:

- **Comparing and Merging Actions**: We compare two actions by recursively analyzing them node by node, considering the action parameters and the components of the action body. This approach reduces comparison overhead and helps identify identical actions. When identical actions are found, we retain only one instance in the merged code and ensure that any references to the action within tables call the retained instance.
- **Comparing and Merging Tables**: We compare tables by examining their keys, maximum sizes, and actions. The actions are compared using the algorithm 3 presented in the previous step. For similar tables with different actions, we avoid merging the actions to preserve both functionalities in the data plane. Merging actions between tables with differing functionalities could risk losing some capabilities, so we retain both versions in such cases, ensuring that both the base and extension control planes remain reusable. Also, for LPM (Longest Prefix Match) tables, we provide a command-line option to prevent their merging due to the potential impact on update times caused by internal reorganization.
- **Comparing and Merging Apply Sub-Blocks**: We compare apply sub-blocks, ensuring that identical ones from the base and extension code, including their if-conditions or apply statements within if-conditions, are retained as a single instance. This comparison is done recursively, node by node, prioritizing the base code. If the sub-blocks are not identical, we maintain the order by placing the apply block from the extension code after the apply block from the base code. This prioritizes the base code while ensuring both functionalities remain fully available.

4) CONTROL VERIFICATION

Our static analyzer verifies the control flow graphs (CFG) of both the base and extension code against the merged code's CFG. This ensures that the merged CFG accurately incorporates the base and extension CFGs.

In algorithm 3, actionMatching constructs a set of matched action pairs by comparing actions from an extension object actions set `extObjActs` with those in the base object actions set `baseObjActs`. It separates the matching process into two key components: comparing entire actions and comparing their internal structures, ensuring better readability and maintainability.

The algorithm iterates over all actions in `extObjActs` and checks whether an action's `nodeID` is already present in `baseObjActs`. If it is not, the algorithm attempts to match it with an existing base action. If a match is found, the pair `(actionBase, actionExt)` is added to `matchedPairs`, ensuring that only structurally equivalent actions are considered, regardless of their names.

Algorithm 3 Action Matching Algorithm

```
Input: extObjActs, baseObjActs
Output: Matched action pairs
1 matchedPairs ← {(actionBase, actionExt) for actionExt in
    extObjActs, actionExt.nodeID not in baseObjActs if
    compareActionNodes(baseObjActs[actionExt.nodeID],
    actionExt)};
2 Function compareActionNodes(actionBase,
    actionExt):
3     return compareNodes(actionBase.parameters,
        actionExt.parameters) and
        compareNodes(actionBase.body, actionExt.body);
4 Function compareNodes(nodeBase, nodeExt):
5     return (size(nodeBase.subNodes) ==
        size(nodeExt.subNodes) and all of
        (compareNodes(subNodeBase, subNodeExt) for
        each (subNodeBase, subNodeExt) in
        (nodeBase.subNodes, nodeExt.subNodes))) if
        nodeBase.hasSubNodes() and
        nodeExt.hasSubNodes() else (nodeBase.type ==
        nodeExt.type and nodeBase.value == nodeExt.value);
```

- **`Comparing Whole Actions` (lines 2-3)**: This step determines whether an action in `extObjActs` matches an action in `baseObjActs`. The algorithm does this by comparing their parameters and body structure. It first checks whether the parameters of both actions are equivalent. Then, it verifies whether their body structures are the same. If both conditions are met, the two actions are considered equivalent.
- **`Comparing Internal Structures` (lines 4-5)**: This step ensures that the individual components within an action, such as parameters and body sub-structures, are structurally equivalent. The algorithm first checks whether the number of sub-nodes in `nodeBase` and `nodeExt` are the same. If they are, it then verifies that all corresponding sub-nodes match recursively. If neither node has sub-nodes, the algorithm compares their types and values to determine equivalence. This hierarchical comparison allows the algorithm to match actions accurately at both a high level and a more detailed structural level.

In algorithm 4, tableMatching updates the matching status of tables in `extObjTables` by comparing them with those in `baseObjTables`. The algorithm ensures that the tables are matched according to their keys, sizes, and associated actions by using algorithm 3, improving modularity and maintainability.

- **Comparing All Tables (lines 1-2)**: The algorithm iterates over each table in `extObjTables` and determines whether it has a corresponding match in `baseObjTables`. For each table in the extension, it checks all tables in the base. If at least one base table satisfies the matching conditions defined by `areTablesMatching`, the extension table is marked as matched.

Algorithm 4 Table Matching Algorithm

Input: extObjTables, baseObjTables
Output: Updated extObjTables
1 **foreach** *extTable* ***in*** *extObjTables* **do**
2 extTable.matched ← any of (areTablesMatching(extTable, baseTable) foreach baseTable in baseObjTables) is true;
3 **Function** `areTablesMatching`(*extTable, baseTable*) **:**
4 **return** *(extTable.size == baseTable.size)* **and** (extTable.keys == baseTable.keys) **and** `areAllActionsMatching`(*extTable, baseTable*) ;
5 **Function** `areAllActionsMatching`(*extTable, baseTable*) **:**
6 **return** *all of (action.matched for each action in extTable.actions) are true*;

- **Comparing Two Tables (lines 3-4)**: To determine whether two tables match, the algorithm verifies three conditions. First, both tables must have the same maximum size, ensuring that they contain the same number of entries. Second, their keys must be identical, meaning they match the same criteria. Third, their associated actions must match, which is determined by calling `areAllActionsMatching`. If all three conditions hold, the tables are considered equivalent.
- **Comparing Actions (lines 5-6)**: The algorithm ensures that all actions within `extTable` have corresponding matched actions in `baseTable` by using algorithm 3. It iterates through each action in the extension table and checks whether it has a corresponding match in the base table. The two tables are considered structurally equivalent if all actions in the extension table are matched.

E. CONTROL PLANE MANAGEMENT OF MERGED DATA PLANE FUNCTIONS

We implemented three analyzers and a helper tool to assist programmers in understanding and troubleshooting the merged code. These tools enable the reuse of control plane functionality from both the base and extension codes, allowing simultaneous functionality in the data plane.

1) PARSER CFG ANALYZER

The parser CFG (Control Flow Graph) analyzer helps programmers explore the control flow of both individual parsers and the merged parser. It enables them to identify overlapping paths in the merged CFG where conflicting functionalities may prevent both features from being active simultaneously. This analysis provides insight into potential conflicts, helping programmers address and resolve them.

2) APPLY CFG ANALYZER

Programmers can use the apply CFG analyzer to view the control flow graphs of the apply blocks in the base, extension, and merged code. This analysis highlights any shared paths between the base and extension apply blocks, identifying areas where conflicting actions could inhibit simultaneous functionality. This information allows programmers to adjust the control plane to prevent such conflicts.

3) SHARED DATA ANALYZER

Shared data is crucial in P4 programs and can impact functionality when modified by multiple actions. For example, if one action modifies 'standard_metadata.egress_spec' early on, and another action overwrites it later, it could prevent both functionalities from being applied simultaneously. The shared data analyzer identifies such data conflicts, enabling programmers to configure the control plane to allow both functionalities to coexist or prioritize one functionality over another.

4) CONTROL PLANE HELPER TOOL

We implemented a control plane helper tool in Python to enable simultaneous reuse of control plane code across different switches to populate tables. The helper includes a logging system that reports each success or error during the table-filling process. In addition to populating tables, it can verify whether each P4 code is correctly configured on each switch, ensuring reliable control plane setup across devices.

V. LEVELS OF MODULARITY IN THE PROPOSED P4 SOLUTION

The proposed P4 solution supports three distinct levels of modularity. The first level is at the construct level, encompassing headers, structs, parser states, and other foundational P4 components. The second level is at the protocol level, covering individual protocols such as IPv4 and IPv6. The third level is at the application level, focusing on high-level network functions like firewall, load balancing, and other application-specific modules. This layered approach allows programmers to design modular P4 programs that can be adapted or extended at various stages.

A. CONSTRUCT-LEVEL MODULARITY

Construct-level modularity involves integrating new elements into the main codebase, such as parser states, tables, actions, headers, and structs. By modularly merging these components, the solution enables more flexible, maintainable, and scalable P4 programs. This level of modularity improves code organization and enhances the ability to add or modify components within the P4 data plane.

B. PROTOCOL-LEVEL MODULARITY

Protocol-level modularity allows for integrating multiple network protocols, such as IPv4 and IPv6, into a single, cohesive data plane. For example, the IPv4 parser includes

states like 'start', 'ethernet', 'ipv4', and 'accept'. The IPv6 parser adds a unique 'ipv6' state while sharing other states with IPv4, enabling both protocols to coexist in the merged code. Similarly, the corresponding deparser, headers, and structs for IPv6 are also incorporated. This allows the compiler to integrate protocol-specific actions, tables, and apply sub-blocks, enabling the data plane to support multiple protocols within a unified framework.

C. APPLICATION-LEVEL MODULARITY

Application-level modularity enables the integration of distinct functional modules, such as firewalls, load balancers, or tunnels, within a single P4 program. Figure 2 illustrates this concept with three configurations of P4 parser code: the base parser, the extension parser, and the merged parser. The left side of the figure shows an advanced tunnel parser with states like 'start', 'parse_ethernet', 'parse_myTunnel', and 'parse_ipv4'. The center represents a firewall parser with states such as 'start', 'parse_ethernet', 'parse_ipv4', and 'parse_tcp'. The merged parser combines these functionalities on the right, integrating 'parse_myTunnel' and 'parse_tcp' states. This setup demonstrates how P4 can seamlessly unify multiple network functions into a single, cohesive network processing solution.

VI. CLASSES OF USE CASES FOR P4 MODULARITY

We present three classes of use cases demonstrating the advantages of P4 modularity from different perspectives, including those of programmers and network managers who wish to implement modular design in P4 development.

A. CODE REUSABILITY IN TEAM ENVIRONMENTS

Code reusability is essential in team settings as it reduces both programming time and complexity, leading to more reliable and less error-prone software. Emphasizing a modular compiler enables developers and network operators to benefit from others' well-tested code. By using a GitHub repository that hosts the primary code, secondary code, and modular compiler, we aim to demonstrate the potential of code reusability within this framework. This is achieved by compiling the main code, which seamlessly incorporates the secondary code using the '#include' directive.

B. DATA PLANE PIPELINE COMPOSABILITY FRAMEWORK

Network operators may often lack expertise in P4 or other network programming languages. However, they often need to compose various functionalities within the data plane pipeline, such as advanced tunneling, firewalls, or quality of service (QoS). As shown in Figure 8, the data plane pipeline composability framework enables network operators to specify and compose these functionalities through a client/server framework.

On the client side, network operators can select desired applications and configurations (e.g., "Advanced Tunnel", "Firewall", "MRI", "QoS") without requiring deep programming knowledge. The "P4Muse Interface" on the server side acts as a code orchestrator, integrating the selected applications by managing '#include' statements and configuration details. The "P4Muse Compiler" then merges these components into a unified codebase supporting multiple functionalities within the data plane pipeline. This architecture enables network operators to specify which component acts as the primary foundation and which serves as extensions, simplifying the deployment of complex data plane pipelines.

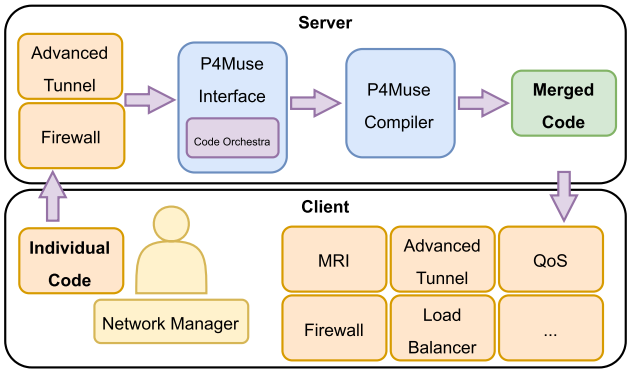


FIGURE 8. Client-server framework for data plane pipeline composability.

C. VENDOR-CUSTOMER FRAMEWORK AND INCREMENTAL PROGRAMMING

With the development of the P4 programming language, vendors have offered platforms with built-in programmability. At the same time, some operators have opted for fully customizable switches, managing both data and control plane development. However, a balanced approach has become the most common, where the vendor provides a switch with preconfigured data plane features that the customer can extend or customize as needed. This approach supports incremental and modular network programming.

However, this framework poses integration challenges. Vendors are often reluctant to share proprietary code, while customers must ensure that customization integrates seamlessly without compromising switch functionality. Business interests, intellectual property (IP) protection, and system reliability are essential considerations for both parties.

To address these challenges, we extend the vendor-customer framework which [10] used for P4Muse without a new annotation system, illustrated in Figure 9, which employs a client/server setup to support incremental P4 programming. In this framework, the "vendor/server" provides preconfigured data plane features through the "P4Muse Interface" and "P4Muse Compiler", which integrate both vendor and customer code into a merged output.

VII. CASE STUDIES OF P4Muse MODULARITY

To illustrate the benefits and effectiveness of P4Muse, we present six case studies based on the P4 tutorials [15] and the Next-Gen SDN tutorial [16], exploring how P4Muse enhances modularity at both the protocol and application

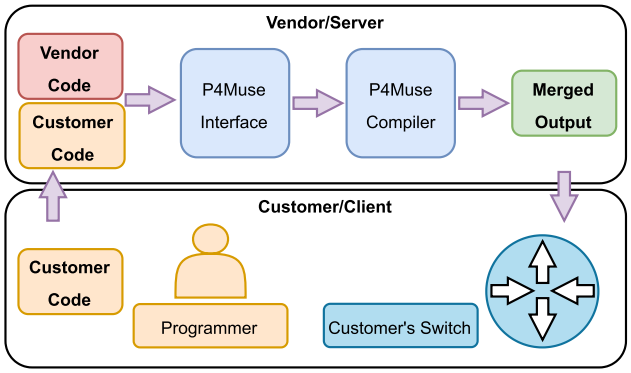


FIGURE 9. Client-server framework for vendor-customer collaboration.

levels, improves security and performance, and demonstrates its extensibility across three different system configurations.

A. IPv4 AND IPv6 PROTOCOLS: PROTOCOL-LEVEL MODULARITY

Our goal in this case study is to achieve a high level of modularity at the protocol level. We focus on IPv4 functionality and then seamlessly integrate an IPv6 module. This modular approach increases the system's flexibility and versatility, enabling it to support multiple protocols.

To promote code reusability, we allow a second programmer to easily integrate the IPv6 protocol module using '#include IPv6.p4'. A GitHub repository was created to host the base and extension code and the P4Muse compiler to facilitate code sharing and reuse in this setup.

We employ a client-server model in the context of data plane pipeline composability. Here, the network operator first selects the IPv4 module, followed by the IPv6 module, and sends these selections from the client to the server. A module orchestrator dynamically integrates the selected modules on the server side by adding '#include IPv6.p4' to the configuration.

In our third configuration, a vendor-customer model safeguards intellectual property while enabling modular integration. In this setup, the vendor (on the server side) retains proprietary IPv4 code, which the customer (client) cannot directly access. The client sends their custom code to the vendor's server, where the proprietary code is merged with the customer's code by adding '#include IPv6.p4', and the vendor sends the merged compiler output instead of the merged P4 code to the customer. This demonstrates how P4Muse's vendor-customer model protects sensitive code while enabling modular customization.

It is important to note that standard #include directives in P4 require the programmer to invoke the included components within the code manually. Merely including a file does not automatically incorporate its functionality. In contrast, our approach automatically injects usage of the included code at the appropriate locations in the struct, parser, and control blocks. Additionally, we provide a command-line option to specify extension code explicitly, enabling us to distinguish modular extensions from standard includes.

B. FIREWALL AND ADVANCED TUNNEL: APPLICATION-LEVEL MODULARITY

This case study focuses on achieving a high level of modularity at the application level. We begin with the firewall module and then seamlessly integrate the advanced tunnel module, increasing the flexibility and adaptability of our application modules.

We have implemented a simple integration approach to promote code reusability within our team. When using that approach, a second programmer can easily incorporate the advanced tunnel code using '#include advancedTunnel.p4', facilitating efficient collaboration and code reuse.

For data plane pipeline composition, we employ a client-server model. In this case study, the network operator selects both the firewall and advanced tunnel modules, sending these choices from the client to the server. A module orchestrator dynamically integrates these selected modules by adding '#include advancedTunnel.p4' within the firewall module on the server side. This demonstrates the extensibility and adaptability of our composition model.

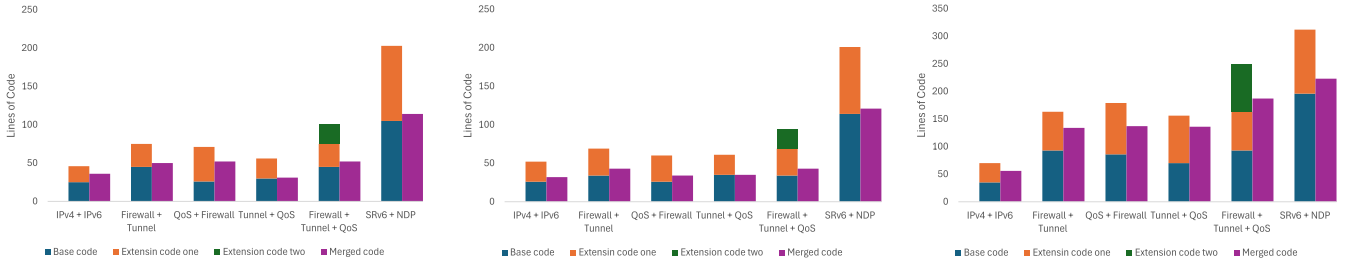
In the vendor-customer model, we also apply a client-server approach, similar to the data plane pipeline composition model. In this setup, the vendor (on the server side) holds proprietary code for the firewall module. Due to intellectual property considerations, the customer cannot directly access or view the vendor's code. Instead, the customer indirectly includes the vendor's code through '#include advancedTunnel.p4', which protects the proprietary implementation. This example highlights the system's ability to protect proprietary code while enabling secure and customizable integration.

C. QUALITY OF SERVICE AND FIREWALL: SECURITY ENHANCEMENT

This case study focuses on enhancing security by integrating a firewall module. To achieve this, we evaluate three specific classes of use cases, each incorporating the directive '#include firewall.p4' into the existing quality-of-service (QoS) codebase. In this setup, the firewall module strengthens the security of the QoS system, serving as a reliable base code. By systematically integrating the firewall module, we streamline the process, creating a more secure framework that effectively mitigates potential vulnerabilities in the QoS system.

D. ADVANCED TUNNEL AND QUALITY OF SERVICE: PERFORMANCE OPTIMIZATION

This case study's primary goal is optimizing system performance by incorporating a modular quality-of-service (QoS) application. We evaluate this setup through three distinct test cases, each involving the integration of '#include qos.p4' into our advanced tunneling system. This modular addition is crucial for efficient management and performance improvement within the tunneling system. By incorporating the QoS module, we enable simultaneous support for both



(a) Header and struct components (b) Parser and deparser components (c) Ingress and egress components

FIGURE 10. Comparison of lines of code across components in base, extension, and merged codes.

functionalities and address performance bottlenecks through modular design.

E. FIREWALL, ADVANCED TUNNEL, AND QUALITY OF SERVICE: EXTENSIBILITY OF MODULAR DESIGN

This case study demonstrates the extensibility of the modular framework by combining multiple applications. Here, we integrate three modules: firewall, advanced tunnel, and quality of service, showing that the modular framework can easily scale to support additional functionalities. Programmers can merge two P4 modules, such as firewall and advanced tunnel, and then extend this configuration by incorporating quality of service. This approach demonstrates how multiple functionalities can be seamlessly integrated, leveraging modularity for reusability and flexibility.

F. SRv6 AND NDP: COMPLEX APPLICATION INTEGRATION

A major objective of P4 modularity is the ability to merge complex applications. In this case study, we explore combining advanced applications—SRv6 (Segment Routing over IPv6 data plane) and NDP (Neighbor Discovery Protocol). This integration highlights the potential of the modular framework to support complex, high-level applications, expanding the capabilities of the data plane. P4Muse scales well. It could easily handle SRv6, the largest P4 benchmark available to us. P4Muse also handles multiple modules. This was shown using various scenarios and combined applications.

VIII. EVALUATION RESULTS OF P4Muse

In P4Muse, we first evaluate the modularity by comparing the number of code lines between each case study and the merged version. We then assess the performance impact by comparing the throughput and latency of each individual case study with the merged configuration. Additionally, we compare P4Muse with five related works, including μP4, Lyra, P4Weaver, P4Ansible, and daPIPE.

A. CODE COMPLEXITY EVALUATION

To assess the efficiency of modularity of P4, we analyzed the code lines in the merged code against its base and extension code in six case studies. In Figure 10a, we compare the code lines in the header and struct sections between the merged code and the combined base and extension code. Figure 10b shows a similar comparison for the parser and deparser sections. In contrast, Figure 10c reports the number of code lines in the ingress and egress sections, highlighting the code reduction achieved through merging.

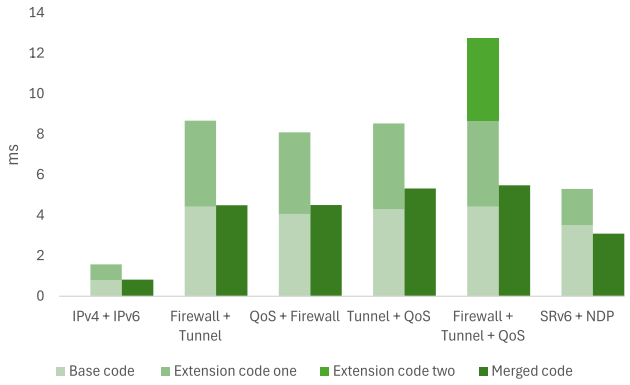


FIGURE 11. Comparison of latency between base, extension, and merged codes.

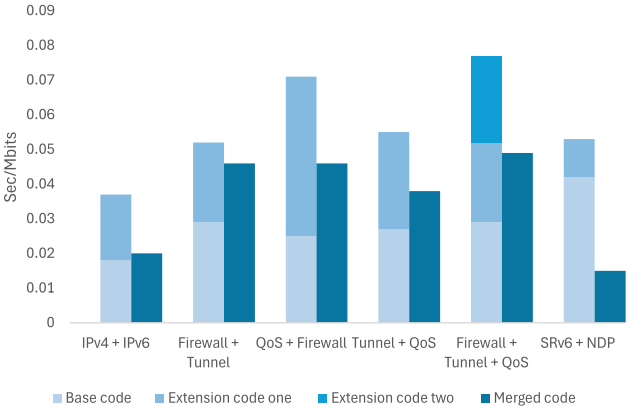


FIGURE 12. Comparison of reciprocal of throughput between base, extension, and merged codes.

B. PERFORMANCE ANALYSIS OF P4Muse

To assess the performance impact of P4Muse, we analyzed the throughput and latency of the merged code compared to

TABLE 1. Feature comparison of P4 modularity solutions.

P4 Solutions	Automatic Merging	Backward Compatibility	Vendor-Customer Compatibility	No New Syntax & Annotation
µP4 [12]	No	No	N/A	No
Lyra [13]	No	N/A	N/A	No
P4Weaver [10]	No	Yes	Yes	No
P4Ansible [11]	No	Yes	Yes	No
daPIPE [9]	No	Yes	Yes	No
P4Muse	Yes	Yes	Yes	Yes

the separate base and extension codes across six case studies. In Figure 11, the latency of the merged code is compared with that of the base and extension codes combined. Similarly, Figure 12 compares the throughput between the merged code and the combined performance of the base and extension codes.

The comparison reveals that P4Muse effectively merges functionalities while maintaining competitive performance. As shown in Figure 11, the merged code generally achieves lower latency than the sum of the base and extension codes, demonstrating the efficiency of the integration. In particular, for complex case studies such as "Firewall + Tunnel + QoS," the merged code exhibits a noticeable reduction in latency compared to the combined individual implementations, indicating reduced processing overhead and improved optimization in packet handling.

Similarly, Figure 12 presents the reciprocal of throughput, where lower values indicate higher throughput performance. The merged code consistently achieves throughput comparable to or better than the sum of its components, reinforcing the effectiveness of P4Muse in optimizing data plane execution. These results confirm that P4Muse introduces no runtime overhead. Notably, in the "SRv6 + NDP" case, the merged code outperforms the combined implementations, suggesting that P4Muse successfully eliminates redundancies and streamlines packet forwarding.

We ran all experiments on an Ubuntu 22.04 LTS system with an 11th Gen Intel Core i5-1135G7 processor running at 2.4GHz (up to 4.2GHz turbo). The processor has 4 physical cores and 8 logical threads, with 32KB L1 data cache, 32KB L1 instruction cache per core, 1.25MB L2 cache per core (5MB total), and an 8MB shared L3 cache. The system is equipped with 16GB of DDR4 main memory.

C. COMPARISON WITH RELATED WORK

Table 1 compares P4Muse to five related works—μP4, Lyra, P4Weaver, P4Ansible, and daPIPE—across features such as Automatic Merging, Backward Compatibility, and Vendor-Customer Compatibility. Unlike P4Muse, each of these works lacks at least one of these essential features and often requires the programmer to learn additional syntax or methods beyond the standard P4 language to utilize them effectively.

IX. MODULARITY CHALLENGES IN P4: CURRENT LIMITATIONS

This section examines current limitations in the P4's language or intermediate representation (IR) and how modularity might be more effectively integrated across different language components. Addressing these limitations could guide future research and development efforts within the P4 community to enable built-in modular and object-oriented support.

A. PARSER LIMITATIONS

In the parser, there is no direct link between each transition value and its associated state; instead, the connection relies on the parent state of that protocol. Achieving modularity within parser states requires more than protocol names alone, as different state names are often used. Transition values, however, allow the compiler to manage modularity even with varying state names. Improving this association in IR could enhance modularity within parser structures.

B. HEADER AND STRUCT LIMITATIONS

Establishing a direct link between parser states, their associated headers, and relevant struct sections would significantly improve modularity for headers and structs. Using transition values and these connections in IR to ensure unique associations between parser states, headers, and structs could streamline modularity and reusability in header and struct data types. Although the P4 language provides a union header structure to hold two or more headers, it does not have a union structure for sub-protocols to accommodate two or more sub-protocols and achieve modularity among them.

C. DEPARSER LIMITATIONS

As with parsers, there is no direct connection between parser states and their corresponding deparser states. Creating these connections in IR would allow the use of transition values to ensure unique associations between parser and deparser states, facilitating modular design within the deparser.

D. CONTROL LIMITATIONS

Integrating connections between parser states and the tables in IR they invoke would improve modularity by allowing the compiler to identify reusable tables and actions tied to specific protocols or applications. Additionally, establishing links between tables and the 'apply' sub-blocks that use them would enable the compiler to reuse tables, actions, and apply blocks when reusing protocols or applications, making modular control flow more feasible.

E. STRUCTURE LIMITATIONS

If the compiler could retain these associations, memory usage would remain minimal, and the connections could support additional P4 constructs, such as externs, 'verifyChecksum', and 'computeChecksum' sections. As demonstrated in P4Muse, it is feasible to establish these connections for

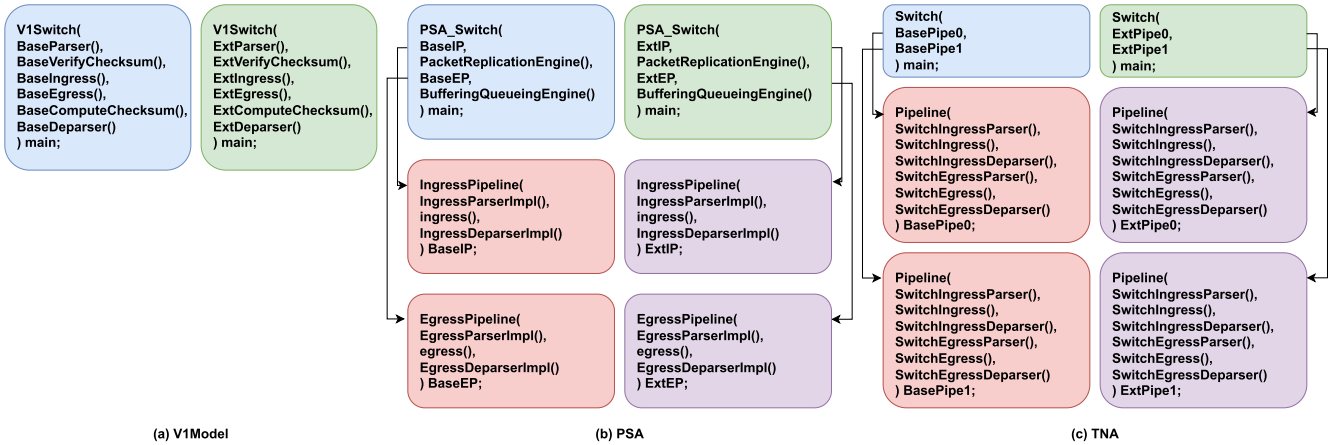


FIGURE 13. Modular comparison workflow across V1Model, PSA, and TNA architectures.

preexisting P4 code, thus ensuring backward compatibility and simplifying code reusability.

X. LIMITATIONS AND FUTURE WORK

While P4Muse significantly improves modularity, some areas could benefit from further optimization. In the parser component, we have assumed that a transition value connects two parser states; therefore, if there is an alternative method of connecting states, P4Muse cannot support it. Additionally, P4Muse treats two nodes as equivalent in the same way P4C does. For instance, P4Muse cannot identify logically equivalent expressions that are written differently using Boolean arithmetic. Therefore, if P4C cannot recognize them as the same node, P4Muse cannot either. Additionally, although P4Muse minimizes the need for new syntax, there may still be a learning curve for users unfamiliar with modular P4 development.

A. FUTURE DIRECTIONS

Future work will focus on streamlining the modularity process by introducing keywords to reference and manage modular objects more efficiently. These keywords aim to simplify integration for new users, enhance scalability, and reduce computational overhead in merging complex protocols and applications. Another priority will be exploring optimizations in the merging algorithms to improve performance in larger systems and refining error-handling mechanisms for seamless integration.

Regarding the design considerations for architecture-agnostic support, P4Muse is primarily designed and validated with the V1Model. However, its modular architecture is intentionally developed to enable potential support for different P4 data plane programming architectures—such as PSA and TNA—with minimal or no modifications in future work (currently under development). As illustrated in Figure 13, the algorithm begins by aligning the top-level packages: main in V1Model, the ingress/egress engine packages in PSA, and the switch/pipeline packages in TNA. It then recursively compares corresponding components, including parsers, ingress, egress, and deparsers. This consistent traversal enables P4Muse to locate and merge equivalent parser, match-action, and deparser sections across architectures by identifying their correct positions within each processing pipeline. Subsequently, similarly as shown in Figure 6, all corresponding ingress and egress parsers of the pipelines are first merged. It then applies the merged states to enable modularity in structs and headers. Subsequently, these are integrated into all ingress and egress deparsers. Finally, the integration extends to the ingress and egress controls.

XI. CONCLUSION

This paper introduces P4Muse, a modular framework developed to enhance modularity in the P4 programming language using the open-source P4 compiler platform. Modularity is essential in programming languages, enabling code reusability, data plane pipeline composability, and vendor-customer compatibility.

P4Muse incorporates modularity into core P4 components, including parser states, headers, structs, deparsers, actions, tables, and apply, while also providing analytical tools to support modular control plane development. Key features of P4Muse include backward compatibility with existing P4 code, automatic merging capabilities, and a vendor-customer model for secure code sharing. Notably, P4Muse achieves these benefits without requiring new syntax or annotations, allowing seamless integration.

Evaluation results demonstrate P4Muse's effectiveness in reducing code complexity and enhancing performance in terms of throughput and latency across several case studies, underscoring its potential as a robust modular solution for P4 development.

ACKNOWLEDGMENT

Intel, Noviflow, Kaloom, Prompt Quebec, and the Natural Sciences and Engineering Research Council of Canada partly supported this work.

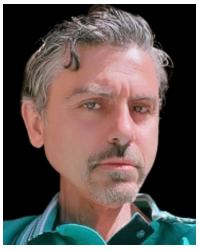
REFERENCES

[1] 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," *J. Netw. Comput. Appl.*, vol. 212, Mar. 2023, Art. no. 103561.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[3] *Open Networking Foundation. OpenFlow Switch Specifications Version 1.5.1*. Accessed: Jul. 15, 2025. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf
[4] 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 Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.
[5] (2023). *P4 16 Language Specification 2023. P416 Language Specification. Version 1.2.4*. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.4-spec.html
[6] (2021). *GitHub: NPL-Spec*. Accessed: Apr. 2, 2021. [Online]. Available: https://github.com/nplang/NPL-Spec
[7] (2013). *The P4 Language Consortium*. [Online]. Available: https://github.com/p4lang/switch/
[8] (2020). *Internet Engineering Task Force (IETF)*. [Online]. Available: https://tools.ietf.org/html/draft-ietf-spring-srv6-network-programming-15
[9] M. Baldi, "DaPIPE a data plane incremental programming environment," in *Proc. ACM/IEEE Symp. Architectures Netw. Commun. Syst. (ANCS)*, Sep. 2019, pp. 1–6.
[10] A. Fattaholmanan, M. Baldi, A. Carzaniga, and R. Soul, "P4 Weaver: Supporting modular and incremental programming in P4," in *Proc. ACM SIGCOMM Symp. SDN Res. (SOSR)*, New York, NY, USA, Jul. 2021, pp. 54–65.
[11] (2021). *MNK Lans and Consulting*. Accessed: Mar. 8, 2023. [Online]. Available: https://mnkcg.com/products/p4-ansible/
[12] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing dataplane programs with μP4," in *Proc. Annu. Conf. ACM Special Interest GroupData Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, New York, NY, USA, Apr. 2020, pp. 329–343.
[13] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, New York, NY, USA, Apr. 2020, pp. 435–450.
[14] (2019). *The P4 Language Consortium*. [Online]. Available: https://github.com/p4lang/p4c
[15] *P4 Language Consortium. P4 Tutorials*. Accessed: Jul. 15, 2025. [Online]. Available: https://github.com/p4lang/tutorials
[16] *Open Networking Lab. The Next-Gen SDN Tutorial*. Accessed: Jul. 15, 2025. [Online]. Available: https://github.com/opennetworkinglab/ngsdn-tutorial
[17] (2022). *Behavioral Model (BMV2)*. [Online]. Available: https://github.com/p4lang/behavioral-model
[18] Y. Zhou and J. Bi, "ClickP4: Towards modular programming of P4," in *Proc. SIGCOMM Posters Demos*, New York, NY, USA, May 2017, pp. 100—102.
[19] E. O. Zaballa, D. Franco, M. S. Berger, and M. Higuero, "A perspective on P4-based data and control plane modularity for network automation," in *Proc. 3rd P4 Workshop Eur.*, New York, NY, USA, Dec. 2020, pp. 59–61.
[20] E. O. Zaballa, D. Franco, E. Jacob, M. Higuero, and M. S. Berger, "Automation of modular and programmable control and data plane SDN networks," in *Proc. 17th Int. Conf. Netw. Service Manage. (CNSM)*, Izmir, Turkey, Oct. 2021, pp. 375–379.
[21] P. D. Bol, R. Lunardi, B. de França, and W. Cordeiro, "Modular switch deployment in programmable forwarding planes with switch (de)composer," in *Proc. SIGCOMM Poster Demo Sessions*, New York, NY, USA, Aug. 2021, pp. 30–32.
[22] S. Han, S. Jang, H. Choi, H. Lee, and S. Pack, "Virtualization in programmable data plane: A survey and open challenges," *IEEE Open J. Commun. Soc.*, vol. 1, pp. 527–534, 2020.
[23] D. Hancock and J. van der Merwe, "HyPer4: Using P4 to virtualize the programmable data plane," in *Proc. 12th Int. Conf. Emerg. Netw. Exp. Technol.*, Irvine, CA, USA, Dec. 2016, pp. 35–49.
[24] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "HyperV: A high performance hypervisor for virtualization of the programmable data plane," in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Vancouver, BC, Canada, Jul. 2017, pp. 1–9.
[25] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-performance virtualization of the programmable data plane," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 556–569, Mar. 2019.
[26] H. Soni, T. Turletti, and W. Dabbous. (2018). *P4Bricks: Enabling Multiprocessing Using Linker-Based Network Data Plane Architecture*. [Online]. Available: https://hal.inria.fr/hal-01632431
[27] P. Zheng, T. Benson, and C. Hu, "P4 Visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, Heraklion, Greece, Dec. 2018, pp. 98–111.
[28] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "P4VBox: Enabling P4-based switch virtualization," *IEEE Commun. Lett.*, vol. 24, no. 1, pp. 146–149, Jan. 2020.
[29] P. Zheng, T. A. Benson, and C. Hu, "Building and testing modular programs for programmable data planes," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 7, pp. 1432–1447, Jul. 2020.
[30] (2019). *The P4 High-Level Intermediate Representation (P4-HLIR)*. [Online]. Available: https://github.com/p4lang/p4-hlir
[31] R. Parizotto, L. Castanheira, F. Bonetti, A. Santos, and A. Schaeffer-Filho, "PRIME: Programming in-network modular extensions," in *Proc. IEEE/IFIP Netw. Operations Manage. Symp. (NOMS)*, Budapest, Hungary, Apr. 2020, pp. 1–9.
[32] R. Parizotto, L. Castanheira, F. Bonetti, A. Santos, and A. Schaeffer-Filho, "Consistent composition and modular data plane programming," *IEEE Commun. Mag.*, vol. 59, no. 6, pp. 60–65, Jun. 2021.
[33] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, "Modular switch programming under resource constraints," in *Proc. USENIX NSDI*, May 2022, pp. 193–207.
[34] D. Zhang, X. Chen, Q. Huang, X. Hong, C. Wu, H. Zhou, Y. Yang, H. Liu, and Y. Chen, "P4SC: A high performance and flexible framework for service function chain," *IEEE Access*, vol. 7, pp. 160982–160997, 2019.
[35] J. Ma, S. Xie, and J. Zhao, "P4SFC: Service function chain offloading with programmable switches," in *Proc. IEEE 39th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Austin, TX, USA, Nov. 2020, pp. 1–6.
[36] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "Compiling service function chains via fine-grained composition in the programmable data plane," *IEEE Trans. Services Comput.*, vol. 16, no. 4, pp. 2490–2502, Apr. 2023.
[37] J. Gao, J. Cao, Y. Li, M. Liu, M. Tang, D. Cai, and E. Zhai, "Sirius: Composing network function chains into P4-capable edge gateways," in *Proc. 21st USENIX Symp. Networked Syst. Design Implement.*, Jun. 2024, pp. 477–490.
[38] T. Wang, X. Yang, G. Antichi, A. Sivaraman, and A. Panda, "Isolation mechanisms for high-speed packet-processing pipelines," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement.*, May 2022, pp. 1289–1305.
[39] Y. Feng, H. Song, J. Li, Z. Chen, W. Xu, and B. Liu, "In-situ programmable switching using rP4: Towards runtime data plane programmability," in *Proc. 20th ACM Workshop Hot Topics Netw.*, Nov. 2021, pp. 69–76.
[40] J. Xing, Y. Qiu, K.-F. Hsu, H. Liu, M. Kadosh, A. Lo, A. Akella, T. Anderson, A. Krishnamurthy, T. S. E. Ng, and A. Chen, "A vision for runtime programmable networks," in *Proc. 20th ACM Workshop Hot Topics Netw.*, New York, NY, USA, Nov. 2021, pp. 91–98.
[41] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime programmable switches," in *Proc. USENIX Symp. Networked Syst. Des. Implement. (NSDI)*, 2022, pp. 651–665.
[42] J. Xing, Y. Qiu, K.-F. Hsu, S. Sui, K. Manaa, O. Shabtai, Y. Piasetzky, M. Kadosh, A. Krishnamurthy, T. S. E. Ng, and A. Chen, "Unleashing SmartNIC packet processing performance in P4," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, Sep. 2023, pp. 1028–1042.
[43] *Internet Assigned Numbers Authority (IANA). Protocol Numbers*. Accessed: Jul. 15, 2025. [Online]. Available: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

MOHSEN RAHMATI received the dual B.Sc. degree in computer engineering from Semnan University, Semnan, and Iran University of Science and Technology, Tehran, in 2017, and the M.Sc. degree in computer engineering from Shahid Beheshti University, Tehran, in 2022. He is currently pursuing the Ph.D. degree in computer and software engineering with Polytechnique Montréal. His Ph.D. research focuses on improving the P4 programming language by using compiler design. His research interests include compiler design, software-defined networking, computer networks, and computer architecture.

FRANÇOIS-RAYMOND BOYER received the B.Sc. and Ph.D. degrees in computer science from Université de Montréal, Montreal, QC, Canada, in 1996 and 2001, respectively. Since 2001, he has been with Polytechnique Montréal, Montréal, where he is currently a Professor with the Department of Computer and Software Engineering. He has authored or co-authored more than 30 conference and journal papers. His current research interests include microelectronics, performance optimization, parallelizing compilers, digital audio, and body motion capture. He is a member of Regroupement Stratégique en Microélectronique du Québec, Groupe de Recherche en Microélectronique et Microsystèmes, and Observatoire Interdisciplinaire de Création et de Recherche en Musique.



BILL PONTIKAKIS received the B.Eng. degree in computer engineering and the M.A.Sc. degree in electrical and computer engineering from Concordia University, Montreal, QC, Canada, in 2002 and 2003, respectively, and the Ph.D. degree in electrical engineering from Polytechnique Montréal, QC, which he was close to completing before deciding to terminate his studies prior to submitting his final thesis. From July 2020 to August 2024, he was a Research Associate with Canada NSERC Industrial Research Chair for High-Speed and Programmable Packet Processors, Department of Electrical Engineering, Polytechnique Montréal. His contributions to the research chair were vital to its success, as he actively participated in the scientific, operational, budgeting, public relations, and research vision aspects of the chair. His research interests include programmable network data planes and the P4 language. He presented his first paper at the IEEE ISCAS Conference during his bachelor's studies, which earned him the ReSMIQ Undergraduate Research Award in Microelectronics, Montreal.

JEAN PIERRE DAVID (Member, IEEE) received the Ph.D. degree from Université Catholique de Louvain, Louvain-la-Neuve, Belgium, in 2002. Following that, he was an Assistant Professor with Université de Montréal, Montreal, QC, Canada, for three years before transitioning to Polytechnique Montréal, Montreal, QC, in 2006. In 2021, he attained the position of a Full Professor. His research interests include digital system design, reconfigurable computing, high-level synthesis, dedicated arithmetic, and practical applications, including signal processing, real-time simulation, and network communications.

YVON SAVARIA (Life Fellow, IEEE) received the B.Ing. and M.Sc.A. degrees in electrical engineering from École Polytechnique Montreal, in 1980 and 1982, respectively, and the Ph.D. degree in electrical engineering from McGill University, in 1985. Since 1985, he has been with Polytechnique Montréal, where he is currently a Professor with the Department of Electrical Engineering. He has carried out work in several areas related to microelectronic circuits and microsystems, such as testing, verification, validation, clocking methods, defect and fault tolerance, effects of radiation on electronics, high-speed interconnects, and circuit design techniques, CAD methods, reconfigurable computing and applications of microelectronics to telecommunications, aerospace, image processing, video processing, radar signal processing, and the acceleration of digital signal processing. He is involved in several projects related to embedded systems in aircraft, wireless sensor networks, virtual networks, software-defined networks, machine learning (ML), embedded ML, computational efficiency, and application-specific architecture design. He has been a Consultant or was sponsored for carrying out research with Bombardier, Buspass, CNRC, Design Workshop, Dolphin, DREO, Ericsson, Genesis, Gennum, Huawei, Hyperchip, Intel, ISR, Kaloom, LTRIM, Miranda, MiroTech, Nortel, Octasic, PMC-Sierra, Space Codesign, Technocap, Thales, Tundra, and Wavelite. He holds 16 patents, has published 211 journal articles and 495 conference papers, and was the thesis advisor of 190 graduate students who completed their studies. He is a fellow of the Canadian Academy of Engineering. He is the Co-Director of the Regroupement Stratégique en Microélectronique du Québec (RESMIQ) and a member of the Ordre des Ingénieurs du Québec (OIQ). In 2001, he was awarded the Tier 1 Canada Research Chair (www.chairs.gc.ca) on the design and architecture of advanced microelectronic systems, which he held until June 2015. He also received the Synergy Award from Canada's Natural Sciences and Engineering Research Council, in 2006. He was the Program Co-Chairperson of NEWCAS'2018 and the General Chairperson of NEWCAS'2020. Since June 2019, he has been the NSERC-Kaloom-Intel-Noviflow (KIN) Chair Professor.