Titre: Title:	P4 Modularity with Compiler Design
Auteur: Author:	Mohsen Rahmati
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Rahmati, M. (2025). P4 Modularity with Compiler Design [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/72091/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

P4 Modularity with Compiler Design

MOHSEN RAHMATI
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Décembre 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

P4 Modularity with Compiler Design

présentée par **Mohsen RAHMATI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Guy BOIS, président
François-Raymond BOYER, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
Jean Pierre DAVID, membre et codirecteur de recherche
Zohreh SHARAFI, membre
Wessam AJIB, membre externe

DEDICATION

To my family and my friends

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to everyone who has supported me throughout my PhD journey.

I am deeply grateful to my supervisors, Professor Yvon Savaria, Professor Jean-Pierre David, and Professor François Reymond Boyer, for their invaluable guidance, expertise, and support. Their mentorship, constructive feedback, and encouragement have been pivotal in shaping my research skills and preparing me for the challenges of a research career.

I would also like to sincerely thank my project manager, Bill Pontikakis, for his leadership.

I would like to thank the members of my committee, Prof. Guy Bois, Prof. Zohreh Sharafi, and Prof. Wessam Ajib, for their insightful comments, support, and encouragement throughout my doctoral journey.

A special thanks to all of my labmates—El-Mehdi, Parisa, Mohammad, Mostafa, Karl, Atabak, Jörg, and others—for their collaboration, support, and friendship. Your insights, encouragement, and shared experiences have made this journey both productive and enjoyable.

Finally, I owe my deepest gratitude to my family and friends. Your unwavering support, patience, and belief in me have been my greatest source of strength throughout this journey. This accomplishment is as much yours as it is mine.

RÉSUMÉ

Les réseaux traditionnels étaient construits à l'aide de matériel à fonction fixe et d'un couplage étroit entre les plans de contrôle et de données, ce qui les rendait peu flexibles face à l'évolution des exigences. Le concept de réseau défini par logiciel ou Software-Defined Networking (SDN) a émergé pour répondre à cette limitation en découplant les plans de contrôle et de données, avec OpenFlow comme l'un des premiers protocoles permettant un contrôle centralisé. Toutefois, OpenFlow et les approches similaires offraient une programmabilité limitée du plan de données. Pour dépasser ces limites, des langages de programmation du plan de données tels que P4 ont été introduits, offrant un traitement des paquets à la fois flexible et performant. À mesure que les programmes P4 sont devenus plus complexes, la nécessité de disposer d'un code modulaire, réutilisable et maintenable est devenue cruciale pour le développement de fonctions réseau évolutives et robustes.

Cependant, malgré sa flexibilité, P4 ne prend pas en charge la modularité de manière native. Dans sa forme standard, les programmes P4 sont souvent développés sous forme de bases de code monolithiques, dans lesquelles l'ensemble de l'analyse de protocoles, des définitions d'en-têtes, de la logique de contrôle et du traitement de données est implémenté dans un seul fichier fortement couplé. Cette structure monolithique présente plusieurs défis. Premièrement, les développeurs doivent souvent dupliquer du code pour implémenter des fonctionnalités similaires dans différents programmes P4, ce qui entraîne une charge de maintenance accrue et des implémentations plus sujettes aux erreurs. Deuxièmement, les composants P4 tels que les analyseurs (parsers), les tables et les actions ne peuvent pas être facilement réutilisés entre différents programmes, obligeant les développeurs à les répliquer et à les intégrer manuellement. Troisièmement, à mesure que les programmes P4 gagnent en complexité, leur maintenance et leur mise à jour deviennent de plus en plus difficiles, en particulier lorsque les modifications doivent être soigneusement propagées à travers plusieurs sections interdépendantes. Enfin, l'intégration de deux ou plusieurs programmes P4 dans une seule base de code unifiée nécessite un effort manuel considérable pour résoudre les conflits, ce qui est à la fois chronophage et susceptible d'introduire des bogues.

Ces défis limitent la possibilité d'étendre à l'échelle et la maintenabilité des programmes P4, en particulier dans les environnements réseau modernes où la capacité de déployer rapidement des fonctions réseau personnalisables et riches en fonctionnalités est essentielle. Cette recherche s'attaque à ces limitations en proposant une stratégie complète pour améliorer la modularité dans P4, tout en maintenant la compatibilité avec les programmes P4 existants.

La première contribution de cette thèse présente P4Muse (P4 Modularity and Unification for Seamless Extensibility), une extension à source ouverte du compilateur P4C qui permet le développement modulaire de programmes P4 sans introduire de nouvelle syntaxe ou d'annotations spécifiques. P4Muse repose sur l'intégration de nouvelles passes de compilation qui analysent et fusionnent automatiquement plusieurs fichiers source P4, facilitant ainsi la réutilisation de composants tels que les analyseurs, les en-têtes, les actions et les tables. En préservant la structure et la sémantique des programmes originaux, P4Muse garantit une compatibilité ascendante et une intégration fluide. L'approche est évaluée à travers trois catégories de cas d'usage : la réutilisabilité du code, la composition de pipelines, et l'interopérabilité entre fournisseurs et clients. Ces cas sont démontrés à l'aide de six études de cas détaillées utilisant l'architecture V1Model et le commutateur logiciel BMv2, mettant en évidence la capacité de P4Muse à soutenir le développement de programmes P4 évolutifs, maintenables et extensibles.

La deuxième contribution de cette thèse introduit P4O2 (P4 Object-Oriented-Inspired), une extension légère et intuitive du compilateur P4C qui améliore la modularité du langage P4 grâce à des constructions syntaxiques explicites et structurées. Inspirée des paradigmes classiques de programmation orientée objet et modulaire, P4O2 permet aux développeurs de définir et d'étendre progressivement des composants réutilisables tels que les en-têtes, les analyseurs, les contrôles et les dé-analyseurs. Cette conception modulaire facilite le développement de bibliothèques évolutives de fonctionnalités réseau standard, simplifie l'intégration de logiques spécifiques aux applications, et réduit la duplication de code. En soutenant la composition modulaire au niveau du langage, P4O2 favorise la séparation des préoccupations, améliore la maintenabilité du code et permet un contrôle précis sur la réutilisation des composants. L'efficacité de P4O2 est démontrée à travers trois études de cas impliquant des piles de protocoles et des applications réseau, toutes évaluées sur le commutateur logiciel BMv2 avec l'architecture V1Model.

Cette recherche démontre que la modularité dans P4 peut être significativement améliorée grâce à deux approches complémentaires : la fusion de composants réutilisables au niveau du compilateur, et une extension légère du langage inspirée de la programmation orientée objet. Les évaluations expérimentales montrent que ces deux approches conduisent à des améliorations mesurables en termes de modularité du code, y compris une réduction de la duplication, une latence plus faible et un débit plus élevé. En particulier, la conception orientée objet permet la création d'un plus grand nombre de modules réutilisables, mettant en évidence son potentiel pour construire des programmes de plan de données évolutifs et maintenables dans des réseaux programmables modernes.

ABSTRACT

Traditional networks were built with fixed-function hardware and tightly coupled control and data planes, making them inflexible to evolving requirements. Software-Defined Networking (SDN) emerged to address this by decoupling the control and data planes, with OpenFlow as an early protocol enabling centralized control. However, OpenFlow and similar approaches offered limited programmability in the data planes. To overcome this issue, data plane programming languages like P4 have been introduced, offering flexible and high-performance packet processing. As P4 programs grow in complexity, the need for modular, reusable, and maintainable code becomes critical for developing scalable and robust network functions.

However, despite its flexibility, P4 lacks native support for modularity. In its standard form, P4 programs are often developed as monolithic codebases, where all protocol parsing, header definitions, control logic, and data processing are implemented in a single, tightly-coupled file. This monolithic structure presents several challenges. First, developers must frequently duplicate code to implement similar functionality across different P4 programs, leading to increased maintenance overhead and more error-prone implementations. Second, P4 components such as parsers, tables, and actions cannot be easily reused across different programs, forcing developers to manually replicate and integrate these components. Third, as P4 programs grow in complexity, maintaining and updating them becomes increasingly difficult, especially when modifications must be carefully propagated across multiple interdependent sections. Finally, integrating two or more P4 programs into a single, unified codebase requires significant manual effort for conflict resolution, which is both time-consuming and susceptible to introducing bugs.

These challenges limit the scalability and maintainability of P4 programs, particularly in modern network environments where the need for rapidly deployable, customizable, and feature-rich network functions is critical. This research addresses these limitations by introducing a comprehensive strategy to enhance P4's modularity while maintaining backward compatibility with existing programs.

The first contribution of this thesis introduces P4Muse (P4 Modularity and Unification for Seamless Extensibility), an open-source extension to the P4C compiler that enables modular development of P4 programs without requiring new language syntax or annotations. P4Muse is built by integrating new compiler passes that automatically analyze and merge multiple P4 source files, facilitating the reuse of components such as parsers, headers, actions, and tables. By preserving the structure and semantics of the original programs, P4Muse ensures

backward compatibility and seamless integration. The approach is evaluated across three categories of use cases: code reusability, pipeline composition, and vendor-customer interoperability. These use cases are demonstrated through six detailed case studies using the V1Model architecture and the BMv2 software switch, highlighting the capability of P4Muse to support scalable, maintainable, and extensible P4 program development.

The second contribution of this thesis introduces P4O2 (P4 Object-Oriented-Inspired), a lightweight and intuitive extension to the P4C compiler that enhances the modularity of the P4 language through explicit and structured language constructs. Inspired by classical object-oriented and modular programming paradigms, P4O2 enables developers to define and incrementally extend reusable components such as headers, parsers, controls, and deparsers. This modular design facilitates the development of scalable libraries of standard network functionalities, simplifies the integration of application-specific logic, and reduces code duplication. By supporting modular composition at the language level, P4O2 promotes separation of concerns, improves code maintainability, and enables fine-grained control over component reuse. The effectiveness of P4O2 is demonstrated through three case studies involving protocol stacks and network applications, all evaluated on the BMv2 software switch using the V1Model architecture.

This research demonstrates that P4 modularity can be significantly improved through two complementary approaches: compiler-level merging of reusable components and a lightweight, object-oriented-inspired extension to the P4 language. Experimental evaluations show that both approaches lead to measurable improvements in code modularity, including reduced code duplication, lower latency, and higher throughput. In particular, the object-oriented-inspired design enables the creation of a larger number of reusable modules, highlighting its potential for building scalable and maintainable data plane programs in modern programmable networks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

SDN	Software-Defined Networking
P4	Programming Protocol-independent Packet Processors
PSA	Portable Switch Architecture
TNA	Tofino Native Architecture
PISA	Protocol Independent Switch Architecture
RMT	Reconfigurable Match-Action Tables
BMv2	Behavioral Model version 2
P4c	P4 compiler
QoS	Quality of Service
IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
TCAM	Ternary Content Addressable Memory
SRAM	Static Random Access Memory
NPL	Network Programming Language
ALU	Arithmetic Logic Unit
RTP	Real-time Transport Protocol
GPRS	General Packet Radio Service
ECMP	Equal-Cost Multi-Path
SMT	Satisfiability Modulo Theories (Solver)

CHAPTER 1 INTRODUCTION

1.1 Motivation

Traditional networks were built with fixed-function hardware and tightly coupled control and data planes, making them inflexible to changing requirements [1, 3]. Software-Defined Networking (SDN) addressed this by decoupling the planes, and protocols like OpenFlow [4, 5] enabled centralized control but offered limited data plane programmability. To overcome this, programmable data planes emerged, and Programming Protocol-independent Packet Processors (P4) [6, 7] was introduced as a domain-specific language to describe packet processing in a flexible and target-independent way. However, despite its expressiveness, P4 lacks native support for modularity. Modularity is a cornerstone of software engineering, enabling better code organization, reuse, and maintenance. In the context of P4, the absence of native modularity leads to various technical and collaborative difficulties. Among the key benefits of modularity are:

- **Complexity Management:** Decomposing large programs into smaller, more manageable components becomes difficult without modularity, resulting in monolithic and less maintainable P4 code.
- **Reuse:** The lack of modular constructs leads to code duplication instead of component reuse, increasing development time and inconsistency.
- **Concurrent Development:** Modularity enables parallel development efforts. Without it, collaboration becomes error-prone and inefficient.
- **Maintenance:** Isolated updates are harder to implement, as changes may ripple across multiple parts of the codebase.
- **Testing and Verification:** Without modular separation, isolated testing of individual functionality becomes challenging.
- **Clear Abstractions:** A modular structure improves readability and separation of concerns. In P4, tightly coupled logic reduces clarity.
- **Scalability:** Programs grow more rigid and harder to extend when modular design is not enforced.

- **Collaboration:** Without clear module boundaries, team integration becomes more complex and error-prone.

Given these challenges, P4 developers are often forced to rely on manual structuring techniques—such as naming conventions, file-level includes, and code duplication—to simulate modularity. These workaround-based practices increase the risk of inconsistency, hinder code reuse, and lead to technical debt over time.

For example, consider a scenario where multiple P4 applications—such as tunneling, load balancing, and monitoring—must be composed into a single data plane program. Without native modular support, developers are forced to manually integrate these functionalities, often duplicating logic, mishandling shared metadata, and creating fragile interdependencies between components that are difficult to test and maintain.

Despite its increasing adoption in production-scale networks, P4 lacks syntax-level constructs and compiler-level mechanisms for modular composition and reuse. This limitation results in error-prone, inflexible, and hard-to-maintain data plane programs, highlighting the pressing need for principled modularity in the P4 programming ecosystem.

1.2 Challenges

Achieving modularity in P4 presents multifaceted challenges that span the language, compiler, and system levels. Originally designed as a monolithic data plane programming language, P4 prioritizes performance and target independence, but lacks built-in constructs for modularity and code reuse. This absence makes it challenging to compose independently developed components without significant manual intervention.

From a compiler perspective, the current P4 intermediate representation (IR) lacks explicit semantic links between related program elements—such as headers, parsers, controls, tables, and actions—limiting the compiler's ability to automatically reason about dependencies, resolve conflicts, and generate a unified program. Introducing modularity features at the language design level requires balancing expressiveness with simplicity, ensuring that new constructs align with P4's minimalistic philosophy and do not break backward compatibility. Any solution must integrate seamlessly into diverse networking environments at the system level while maintaining performance guarantees and supporting incremental adoption.

These interdependent challenges must be addressed collectively to enable robust, automated, and scalable modularity in P4, unlocking new possibilities for code reuse, maintainability, and collaborative development in programmable data planes.

The following subsections examine these challenges in greater detail.

1.2.1 Modular Code Composition in P4

P4 was originally designed as a monolithic data plane programming language, lacking built-in constructs for modularity and code reuse. This absence creates several challenges in developing and maintaining large-scale or collaborative P4 programs. In particular, achieving modular composition requires the reliable identification, integration, and linkage of code elements—such as headers, parsers, controls, tables, and actions—across independently developed modules. However, the lack of explicit interfaces, standardized naming conventions, and semantic links between these elements makes it difficult to automate composition and avoid ambiguities.

In addition, P4 programs often involve tightly coupled logic, where protocol-specific behaviors are intertwined. This structure complicates the separation of concerns and increases the risk of code duplication and inconsistencies when similar functionality must be shared or reused. The challenge is further amplified when integrating code from different sources, such as protocol libraries, vendor modules, or customer-developed features, each potentially following its own conventions and abstractions.

A modularity solution must overcome these obstacles to be effective by introducing mechanisms for module boundary definition, automated merging of code sections, and semantic correctness during composition. It must also ensure that the composed program remains efficient and backward compatible, without introducing runtime or compilation overheads that would deter adoption in performance-sensitive networking environments.

Enhancing IR Connectivity for Automated Code Merging

The development of P4Muse (chapter 3) revealed several limitations in the P4 language and its intermediate representation (IR) that hinder fully automated program composition. These limitations are not merely implementation issues; they reflect a deeper absence of explicit semantic connections across language components.

In the current P4 IR, parser states are not explicitly linked to their corresponding headers, structs, or deparser states. Similarly, control blocks lack direct associations to the parser states and tables they depend on. This lack of connectivity makes it difficult for a compiler to reason about code relationships and to perform automated merges.

If such associations were maintained, the compiler could exploit FSM-based parser comparisons to automatically produce a merged FSM and, by extension, merge related headers,

structs, deparser sections, and controls without manual intervention. This capability would enable a new level of compiler-managed modularity, allowing two or more independently developed P4 programs to be composed with minimal programmer effort.

1.2.2 Modularity Abstractions and Language Extension

Beyond the technical hurdles of code merging and reuse, a second category of challenges involves designing and adopting language-level abstractions that support modularity in a scalable and extensible manner. While many traditional programming languages leverage object-oriented constructs such as inheritance, encapsulation, and polymorphism, P4 does not natively support such features.

Introducing modularity abstractions into P4 thus requires careful consideration of several factors. First, new syntax or constructs must be minimal, intuitive, and consistent with the design principles of P4, avoiding unnecessary complexity or breaking changes to legacy code. Second, the language extensions must be expressive enough to enable advanced composition patterns—such as selective inheritance, multiple inheritance, and fine-grained extension—while providing clear rules for conflict resolution and semantic preservation.

At the same time, these abstractions must not compromise P4's target-independence or high-performance goals and should be fully supported by the reference compiler, toolchains, and verification frameworks. A further challenge is ensuring that new modularity features can be adopted incrementally, allowing existing P4 applications and libraries to benefit from modular programming without extensive rewrites.

In summary, the challenges of enabling modularity in P4 span from low-level code integration and merging to the high-level definition and adoption of scalable, maintainable, and backward-compatible language abstractions. Addressing these challenges is essential for unlocking the full potential of programmable data planes in modern networking environments.

1.3 Problem Statements

While modularity is a foundational principle in software engineering, the P4 language lacks native constructs for modular programming. This limitation presents significant challenges in developing, maintaining, and evolving large-scale and reusable P4 programs.

More specifically, this research addresses the following problems:

- The P4 language provides no built-in mechanism for composing multiple modules or integrating reusable program components in a structured way.

- The existing P4 reference compiler (P4C) lacks support for modular merging or incremental integration of program logic.
- There is no syntax or abstraction for declaring reusable components or extending existing constructs such as headers, parsers, or controls.

These limitations restrict data plane applications' scalability, reusability, and testability. In the absence of modular support, P4 developers are forced to manually duplicate code, adopt naming conventions, or rely on fragile and file-level include techniques.

1.4 Research Objectives

1.4.1 General Objective

The overarching objective of this doctoral research is to enhance modularity in P4 programming within the Protocol Independent Switch Architecture (PISA) by enabling the composition and reuse of independently developed data plane components. This objective is pursued through two complementary approaches: (i) a compiler-based solution that supports modularity without introducing new language syntax, and (ii) a syntax-level modularity framework inspired by object-oriented programming principles. Together, these approaches aim to provide robust, scalable, and backward-compatible mechanisms for modular development, ensuring that reusable P4 components can be efficiently integrated into diverse networking environments without sacrificing performance.

1.4.2 Specific Objectives

Objective 1: Enable compiler-managed modularity without syntax changes. In this context, we design and implement new compiler passes to analyze and merge multiple P4 source files into a unified program while preserving program semantics. This approach requires no language changes, thus ensuring full backward compatibility with existing P4 programs.

Sub-objective 1.1 – Modular parser integration: Automatically merge and compose parsers from different P4 files into a coherent parsing pipeline. This work is reported in:

Article 1.1: Rahmati, M., Boyer, F.-R., Pontikakis, B., David, J.-P., Savaria, Y., *"Modular Code Parser for the P4 Language,"* Proceedings of the P4 Workshop, 2023.

Sub-objective 1.2 – Modular integration of non-parser elements: Extend the merging process to other PISA components, including headers, controls, actions, tables, and

deparsers. This work is reported in:

Article 1.2: Rahmati, M., Boyer, F.-R., Pontikakis, B., David, J.-P., Savaria, Y., *"P4Muse: Enabling Modular P4 Programming via Compiler-Managed Code Merging Without Syntax Modifications,"* IEEE Access, vol. 13, pp. 124138–124157, 2025.

Objective 2: Introduce syntax-level modularity inspired by object-oriented design.

Here, we propose lightweight extensions to the P4 language that allow developers to explicitly define and extend modular components. The design incorporates inheritance-like mechanisms for reusing and extending core PISA elements while keeping the syntax minimal and compatible with P4's design philosophy.

Sub-objective 2.1 – Parser inheritance: Provide mechanisms to incrementally extend parser logic from existing parsers, enabling structured reuse of parsing states.

Sub-objective 2.2 – Modular design across the PISA pipeline: Extend inheritance-like constructs to other elements such as headers, controls, and deparsers, allowing developers to build layered and extensible data plane architectures. Two sub-objectives are reported in:

Article 2: M. Rahmati, F.-R. Boyer, B. Pontikakis, J.-P. David, and Y. Savaria, *"P4O2: Enabling Object-Oriented-Inspired Modularity in P4,"* IEEE Access, submitted (September 2025).

Collectively, these objectives establish a comprehensive framework for P4 modularity that bridges compiler-based automation and developer-driven design, fostering maintainability, scalability, and reusability in programmable data planes.

1.5 General Organization of the Thesis

The structure of this thesis is as follows:

- **chapter 1 — Introduction**
  This chapter presents the motivation behind this research, outlines the problem statements, and defines the general and specific objectives pursued in this thesis. It details the challenges of achieving modularity in P4 from language, compiler, and system perspectives, and introduces the two complementary approaches developed in this work: a compiler-based solution without syntax changes and a syntax-level modularity framework inspired by object-oriented principles. It also provides the organization of the thesis.

- **chapter 2 — Literature Review**
  This chapter reviews and summarizes prior work on enhancing modularity in the P4 programming language, including SDN automation, data plane composition and virtualization, compiler design, incremental programming, service function chaining, and runtime programmability. Existing approaches, ranging from hypervisor-based to compiler-based solutions, have improved flexibility, portability, and performance. A common limitation across these efforts is the absence of compiler-native, backward-compatible modularity integrated directly into the P4 language and intermediate representation. We conclude the chapter with a summary of the literature, identifying key limitations and outlining the motivations that drive the research contributions of this thesis.

- **chapter 3 — Article 1: P4Muse: Enabling Modular P4 Programming via Compiler-Managed Code Merging Without Syntax Modifications**
  This chapter presents P4Muse, a compiler-based approach that enables modular composition of multiple P4 programs without altering the standard language syntax. P4Muse implements compiler-managed code merging to integrate independently developed P4 modules, facilitating greater code reuse, maintainability, and scalability. The chapter details the system architecture, integration workflow, and evaluation results, demonstrating how P4Muse addresses key limitations in existing modularity solutions while preserving full compatibility with the P4 compiler.

- **chapter 4 — Article 2: P4O2: Enabling Object-Oriented-Inspired Modularity in P4**
  This chapter introduces P4O2, a backward-compatible extension to the P4 language that incorporates object-oriented-inspired features to enhance modularity in data plane programming. P4O2 extends the standard syntax to support inheritance and composability across critical language constructs such as structs, parsers, and control blocks. The chapter outlines the language design, compiler modifications, and inheritance mechanisms, and evaluates P4O2's ability to streamline development, improve reusability, and enable scalable composition of complex data plane functionalities.

- **chapter 5 — General Discussion**
  This chapter offers a comprehensive analysis of the two modularity frameworks developed in this thesis—P4Muse and P4O2—in the context of the challenges and requirements identified in the literature review. It contrasts P4Muse's compiler-managed, syntax-free merging with P4O2's lightweight, object-oriented-inspired syntax extensions, evaluating their respective strengths, trade-offs, and areas of applicability. The discussion addresses integration with the P4 compiler's IR, conflict resolution strate-

gies, backward compatibility, and the constraints imposed by P4's original monolithic design. Performance results from diverse case studies are reviewed to assess practicality, and potential adoption barriers are examined, including dependency on compiler modifications and limited testing on hardware targets. The chapter concludes by identifying opportunities to extend these frameworks to dynamic modularity, integration across control planes, and deployment on emerging P4 architectures and high-performance platforms.

- **chapter 6 — Conclusion**
  This chapter summarizes the thesis's core contributions and their significance for advancing compiler-native modularity in the P4 language. It highlights P4Muse, a syntax-free, compiler-managed framework for automatic module composition, and P4O2, a backward-compatible language extension introducing inheritance for structs, parsers, and controls. Together, these approaches enable scalable, maintainable, and reusable data plane programming while preserving compatibility with existing P4 codebases. The chapter discusses the implications for P4 language evolution, standardization, and best practices. It also outlines future research directions—including dynamic and runtime modularity, ecosystem integration, enhanced developer tooling, adaptation to new architectures, advanced conflict resolution, formal verification, and integration with control plane modularity. These contributions lay a practical foundation for robust, scalable, and sustainable modular programming in programmable data planes.

CHAPTER 2 LITERATURE REVIEW

The growing demand for flexible, high-performance networks has driven research from traditional fixed-function designs toward programmable data planes. This shift has been enabled by Software-Defined Networking (SDN) and the P4 programming language, which together provide new levels of control and customization in packet processing. At the same time, the complexity of network applications and the diversity of hardware targets have highlighted the need for modularity: the ability to compose, reuse, and extend P4 programs efficiently without rewriting entire pipelines.

This chapter is organized as follows. We begin with an overview of the evolution of networks and SDN in section 2.1, highlighting the foundations of programmability. We then introduce data-plane programmability, programming models, and the PISA architecture, followed by a detailed discussion of the P4 language, its software targets, and common architectures. Next, we review P4 compilers and their two-layer design. Building on this foundation, we examine prior research on modularity in section 2.2, covering daPIPE, μP4, Lyra, and P4 Weaver. Finally, we synthesize the limitations of existing approaches and outline the motivations for this thesis in section 2.3.

2.1 Background

2.1.1 Evolution of Networks and SDN

Early computer networks were relatively simple compared to today's advanced systems. These networks emphasized the independent functioning of each network element over long periods, due to limited interconnectivity and high failure rates. This required complex protocols and management methods. However, the advent of Cloud technologies brought about a shift towards highly interconnected networks for rapid connectivity, even though the fundamental structure of networks remained essentially unchanged [1, 3].

The initial wave of Software-Defined Networking (SDN) brought a significant change by separating the data plane from the control plane, placing the control plane within a logically centralized unit known as the controller. This controller takes on the role of managing the underlying network. Moreover, the interaction between the data plane and the control plane is facilitated by a clearly specified API, with OpenFlow [4] being the most prominent protocol associated with SDN [1, 3].

OpenFlow facilitates communication between the controller and the data plane devices. It

allows the controller to implement specific network policies through a set of predefined messages. However, OpenFlow's primary limitation lies in its fixed header fields. Each new iteration of the protocol must be sanctioned by the Open Networking Foundation (ONF) and subsequently implemented by hardware manufacturers. While this offers more flexibility than traditional network setups, the capabilities are still constrained by the predefined features of the OpenFlow protocol. Recently, OpenFlow development efforts have been shifted in favor of the P4 project [1, 3].

The latest generation of SDN introduces numerous advancements and capabilities. Key among these is hardware independence, which creates a unified abstraction layer where hardware specifics do not interfere with the switch's functionality. The programming approach has also evolved, becoming top-down; network operators define the desired outcomes, and the network autonomously determines the implementation details. This new era of SDN, often referred to as Next Generation SDN (NG-SDN), champions open interfaces and white-box hardware, moving away from closed, proprietary solutions. Figure 2.1 of the document provides a visual comparison of these networking paradigms [1, 3].

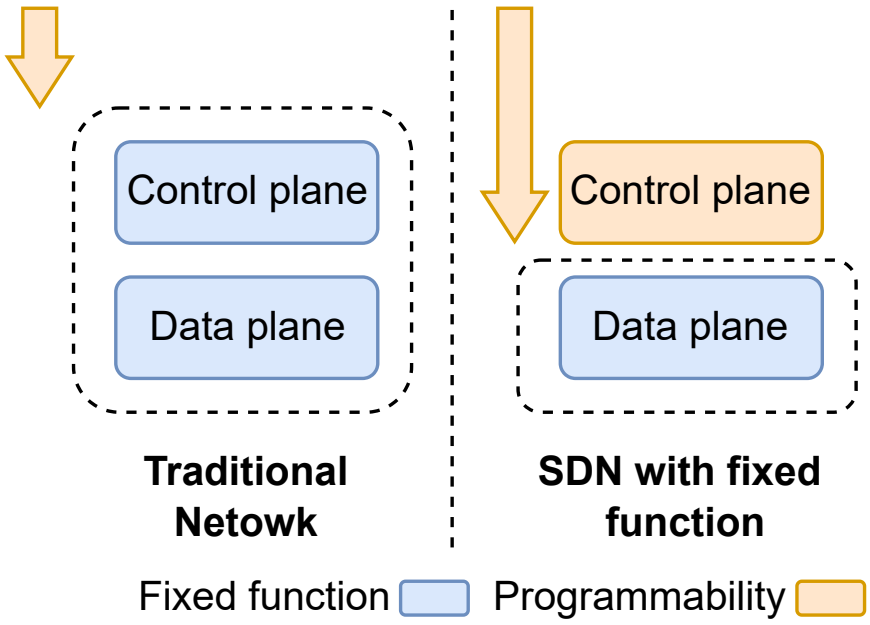


Figure 2.1 Traditional networking tightly integrates control and data plane functions within network devices, limiting flexibility. SDN decouples the control plane and introduces programmability through centralized control, but the data plane remains fixed-function and non-modular [1].

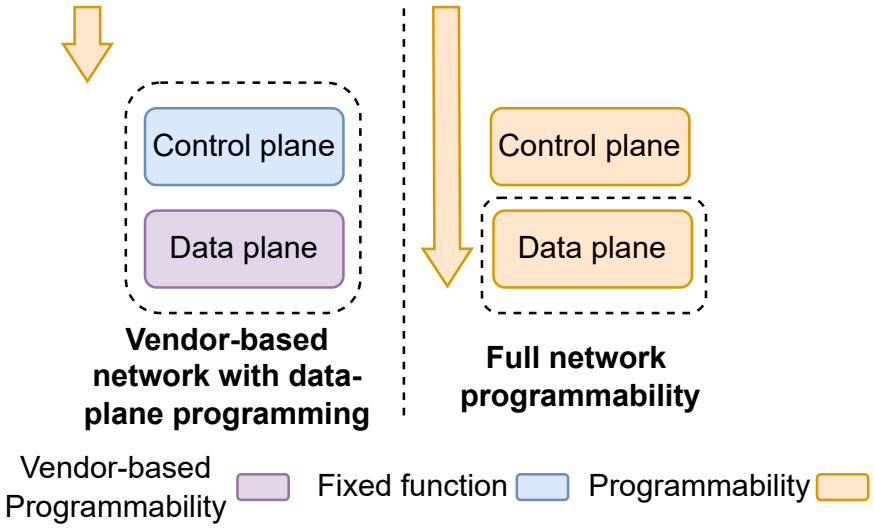


Figure 2.2 Illustration of different levels of data plane programmability. Vendor-based solutions provide limited programmability constrained by proprietary implementations, while full network programmability enables flexible control over both control and data plane behaviors [1].

2.1.2 Data plane programmability

Data plane programmability allows users, such as network operators or equipment designers working with packet processing Application-specific integrated circuit (ASIC)s, to define data plane algorithms. Historically, data plane programmability has been a part of the networking industry, with algorithms typically running on general-purpose CPUs. However, the emergence of high-speed links that exceeded CPU processing capabilities, along with the introduction of packet processing ASICs, brought the issue of data plane programmability to the forefront [1, 3].

Data plane algorithms play a crucial role in processing all packets within a telecommunications system, thereby determining the system's functionality, performance, and scalability. Implementing data plane functions in the control plane often leads to notable performance issues. When users have data plane programming capabilities, it significantly enhances their ability to develop custom network equipment without compromising performance, scalability, speed, or energy efficiency [1, 3].

This programmability enables the creation of tailor-made networks, control planes, and SDN applications with ideally suited data plane algorithms. Data plane programming does not inherently involve providing APIs for user access or supporting external control planes like OpenFlow (OF). Device vendors may choose to develop proprietary control planes and uti-

lize data plane programming for their own advantage, without necessarily opening up their systems. However, many are now moving towards more open systems. Figure 2.2 in the document illustrates both scenarios [1, 3].

2.1.3 Data plane programming models

Data plane algorithms, crucial for network operations, are typically written in standard programming languages but often face challenges when implemented on specialized hardware like high-speed ASICs. To address this, various data plane models have been developed as abstractions to better align these algorithms with the hardware capabilities. Data plane programming languages are specifically designed to interact with these models, allowing for the creation of abstract algorithms. These algorithms are then compiled into executable code for specific packet processing nodes that are compatible with the chosen data plane programming model [1, 3].

Among the notable data plane models are data flow graph abstractions and PISA. These models provide a framework for understanding and designing network data processing functions. PISA, in particular, is significant for the P4 programming language, as it serves as the foundational data plane programming model, enabling the effective implementation of data plane algorithms in P4 [1, 3].

Data flow graph abstractions

In data plane programming models, the process of packet processing is depicted as a directed graph. In this graph, each node symbolizes the basic reusable operations that are applied to packets, such as modifying packet headers. The directed edges connecting these nodes illustrate the path packets take through the graph, and decisions about packet traversal are made at each node. An example of such a graph can be seen in Figure 2.3 illustrating the forwarding of IPv4 and IPv6 packets, where the flow and processing steps are clearly outlined [1, 3].

PISA: Protocol Independent Switch Architecture

The Protocol-Independent Switching Architecture (PISA) is a sophisticated framework for modern switching hardware, evolving from the concepts of reconfigurable match-action tables (RMTs) and disaggregated reconfigurable match-action tables (dRMTs). Central to PISA are three main components: a programmable parser, a programmable deparser, and a match-action pipeline with multiple stages. The parser deals with organizing packet head-

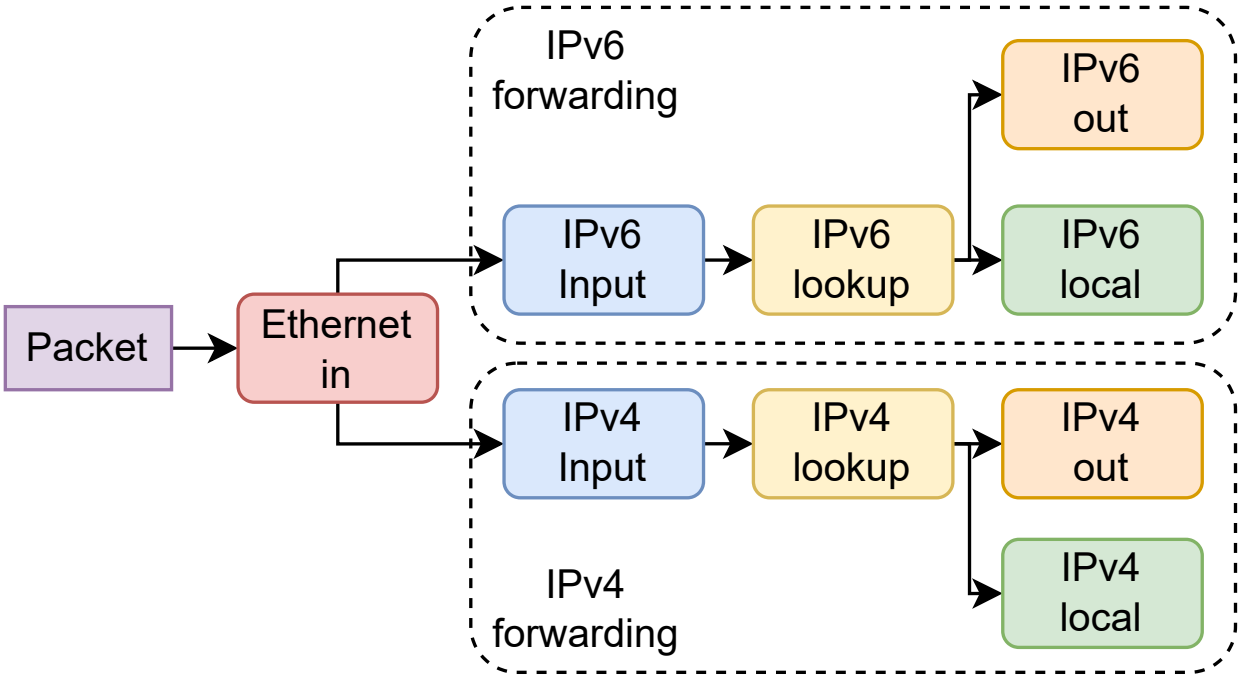


Figure 2.3 An example diagram illustrating the application of data flow graph abstractions for executing IPv4 and IPv6 packet forwarding [1].

ers using a finite-state machine, transforming serialized headers into a structured format. The pipeline consists of match-action units, each containing match-action tables (MATs) for packet matching and specific actions, supported by memory units like Static Random Access Memory (SRAM) or Ternary Content Addressable Memory (TCAM) and Arithmetic Logic Unit (ALU)s for processing. Stateful elements like counters and meters are also used for advanced functions. The deparser is responsible for the final packet serialization, and a control plane manages the system's runtime behavior [1, 3].

PISA processes packet metadata, including packet headers, intrinsic metadata, and user-defined metadata. Packet headers represent network protocol headers, intrinsic metadata relates to fixed-function components, and user-defined metadata acts like temporary storage for information used in the processing pipeline. PISA's architecture is versatile, allowing for various configurations of programmable and fixed-function components for advanced processing tasks. P4, a domain-specific programming language, is commonly used for describing data plane algorithms in PISA. Developed and standardized by the P4 Language Consortium under the Open Networking Foundation, P4 has evolved from its initial introduction in 2013 to its current specification, P4-16, introduced in 2016 [1, 3].

Figure 2.4 illustrates the Protocol-Independent Switching Architecture (PISA), showcasing

its main components: the programmable parser, match-action pipeline, and deparser. The programmable parser is shown at the start, which converts packet headers into structured formats. The central part of the Figure 2.4 probably focuses on the match-action pipeline, detailing the stages and match-action units, along with memory elements like SRAM or TCAM, and arithmetic logic units (ALUs) for processing. The deparser is depicted at the end, responsible for re-serializing packets after processing. The Figure 2.4 also possibly highlights how different types of packet metadata flow through this system, clearly delineating their roles and interactions within PISA [1, 3].

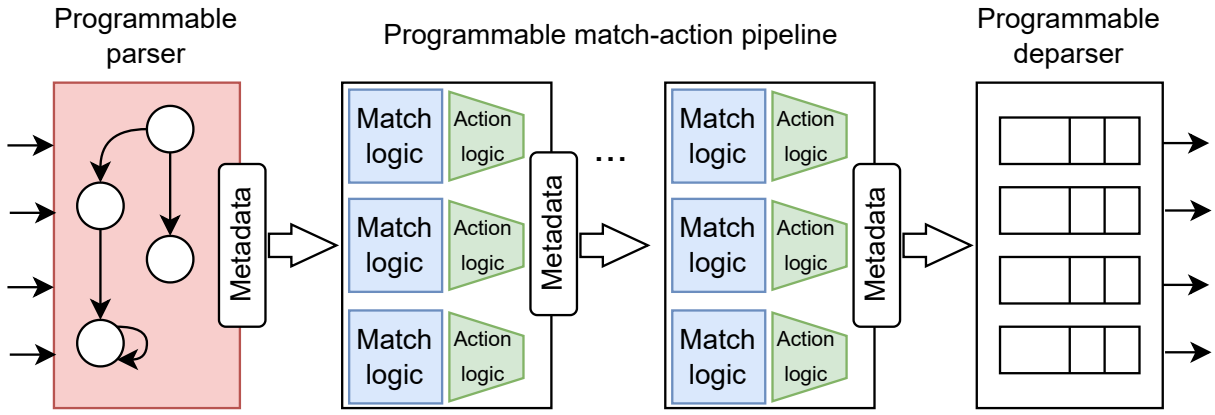


Figure 2.4 The Protocol-Independent Switch Architecture (PISA) is equipped with a programmable parser, a flexible match-action pipeline, and an adaptable deparser [1].

2.1.4 P4: Programming Protocol-independent Packet Processor

The P4 programming language is a domain-specific language designed for programming the data plane in network devices. It extends programmability beyond the control plane, as seen in SDN (Software-Defined Networking), into the data plane. This approach enables network devices, such as switches and routers, to perform more complex and tailored data processing functions. Unlike SDN, which mainly focuses on controlling network devices through external software-based control planes, P4 allows direct programming of the device's data plane. This provides greater flexibility and control over packet processing and forwarding decisions within network hardware [1, 3].

P4 has evolved through different standards, primarily P4-14 and P4-16. P4-14 introduced a mix of imperative and declarative programming constructs, with a focus on data-plane-specific functionalities like counters and checksum calculations. However, it had limitations in terms of architecture flexibility and program modularity. To address these issues, P4-16 was introduced. It separates core language components from architecture-specific components,

making it architecture-agnostic. P4-16 supports multiple pipeline architectures and target-specific functions, introduces strict typing, nested data structures, and has fewer language keywords compared to P4-14. This evolution has led to a more modular, flexible, and robust language structure, enabling better management and development of network data plane functionalities [1, 3].

The development and deployment process for P4 involves writing programs for specific P4 architecture models, which describe the structure of packet processing pipelines in network devices. These programs are then compiled by P4 compilers, specific to each device manufacturer, into target-specific code. This code is executed on P4-programmable network devices, which can be either hardware or software-based. The compiled P4 programs define the algorithms for data processing in the device's data plane, interacting with the network hardware's programmable and fixed-function components. This process highlights the flexibility and customizability offered by P4 in network data plane programming [1, 3].

2.1.5 Software-based P4 Targets

Software-based P4 targets refer to packet forwarding applications executed on conventional CPUs [1, 3].

p4c-behavioral

The initial offering in this category was p4c-behavioral, a tool that combines the Compiler of P4-14 language with a software target. p4c-behavioral converts P4-14 code into an executable C program [8].

Behavioral Model version 2 (bmv2)

The Behavioral Model version 2 (bmv2) was launched as an improved P4 software switch, aiming to overcome the constraints of p4c-behavioral and works with P4-16. Unlike p4c-behavioral, bmv2's source code was separated from the P4 compiler source code. Instead, P4 codes are translated into a JSON format and then integrated into bmv2 at runtime. To introduce external functionalities and enhancements, one can modify the C++ source code of bmv2. It is essential to note that bmv2 represents a group of targets, not just one [9].

- simple_switch is the most feature-rich target within bmv2. It encompasses all attributes from the P4-14 specification and is compatible with the v1model [10] architecture from P4-16. Additionally, simple_switch offers a Thrift API that does not rely on

a specific program for runtime management.

- simple_switc_grpc is an extension of simple_switch, incorporating the P4Runtime API that utilizes gRPC.
- psa_switch is akin to simple_switch but aligns with PSA rather than v1model.
- Both simple_router and l2_switch only accommodate parts of the standard metadata and lack support for P4-16. Their primary purpose is to demonstrate the versatility of bmv2 in implementing diverse architectures.

2.1.6 Overview Of Common P4-16 Architectures

We cover the four most prevalent P4-16 architectures, namely V1model, Portable Switch Architecture (PSA), Tofino Native Architecture (TNA), and SimpleSumeArchitecture [1, 3].

V1model

The v1model [10] architecture emulates the P4-14 processing pipeline, as shown in Figure 2.5. It includes a programmable parser, ingress and egress match-action pipelines, a traffic manager, and a deparser. This model facilitates converting P4-14 programs to P4-16, aligning with the advancements in the reference P4 software switch, bmv2 [1, 3].

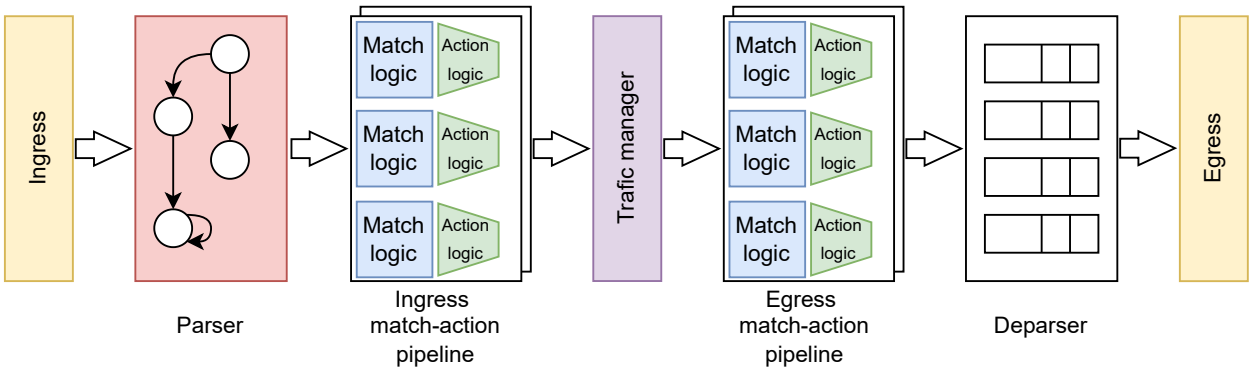


Figure 2.5 The v1model architecture features a programmable parser, ingress and egress match-action pipelines separated by a traffic manager, and concludes with another programmable deparser [1].

Portable Switch Architecture (PSA)

The PSA [11], developed by the P4 Architecture Working Group in the P4 Language Consortium, is based on the P4 architecture. This group focuses on discussing standard func-

tionalities, APIs, and externs necessary for every target that maps the PSA [11]. The PSA's structure, as shown in Figure 2.6, includes a P4 processing pipeline divided into ingress and egress sections, each comprising three programmable parts: a parser, multiple control blocks, and a deparser. Additionally, the architecture incorporates configurable fixed-function components [1, 3].

The PSA outlines various packet processing operations. These include sending packets to a unicast port, dropping packets, directing packets to a multicast group, and specific methods for packet handling, like resubmitting, recirculating, and cloning packets. Resubmitting moves a packet from the end of the ingress pipeline back to its start for re-parsing. Recirculating sends a packet from the egress pipeline's end to the ingress pipeline's start for recursive processing, like tunneling. Cloning involves duplicating the processed packet, with Clone Ingress to Egress (CI2E) creating a duplicate at the ingress pipeline's end, and Clone Egress to Egress (CE2E) duplicating the deparsed packet at the egress pipeline's end. In both cloning cases, the duplicates begin processing at the start of the egress pipeline. These cloning processes are useful for applications such as mirroring and telemetry [1, 3].

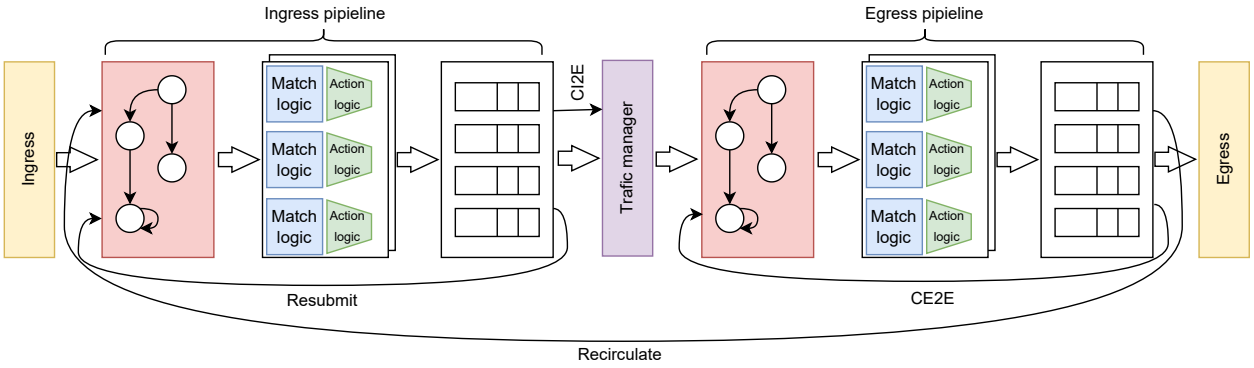


Figure 2.6 The Portable Switch Architecture (PSA) features an ingress and an egress pipeline, separated by a traffic manager. Each pipeline has a programmable parser, match-action units, and a deparser, alongside fixed-function components. Additionally, the PSA incorporates specific packet processing primitives for enhanced functionality [1].

SimpleSumeArchitecture

The SimpleSumeArchitecture, tailored for FPGA-based P4 targets, is a streamlined P4 architecture. Illustrated in Figure 2.7, it consists of a parser, a programmable match-and-action pipeline, and a deparser [1, 3].

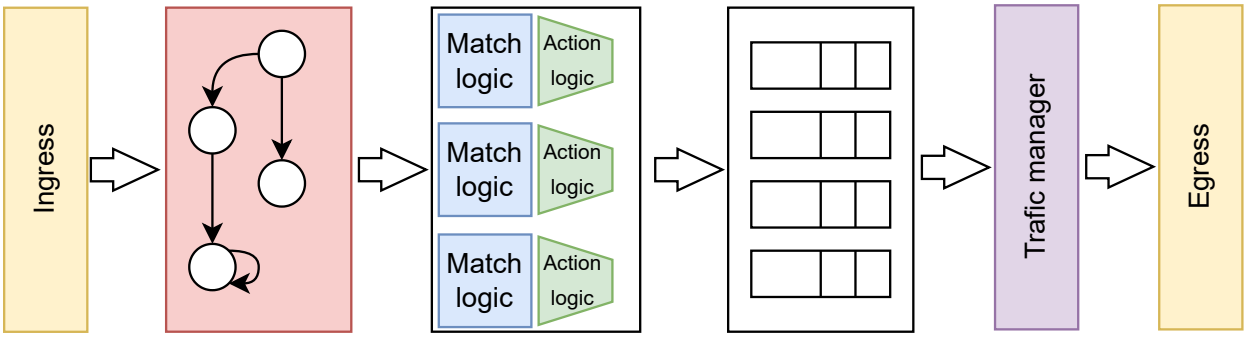


Figure 2.7 the SimpleSumeArchitecture, which includes a programmable parser, a match-action pipeline that is also programmable, followed by a traffic manager [1].

Tofino Native Architecture (TNA)

The Tofino Native Architecture (TNA) is Intel's proprietary P4-16 architecture, specifically designed for its Tofino switching ASICs. It represents a high-performance, industry-grade, and relatively complex device. The architecture's core is the Pipeline() package, an extended version of the Portable Switch Architecture (PSA) pipeline. This package includes six primary programmable components: ingress parser, ingress match-action control, ingress deparser, and their respective egress counterparts. Tofino devices, capable of housing two or four processing pipelines, allow for creating up to four distinct pipeline packages [1, 3].

TNA stands out for its extensive range of externs, which surpass those in other architectures. A unique feature is the TNA RegisterAction(), which allows small code fragments to be executed on registers and offers more than just simple read/write operations. This architecture also facilitates mirroring and resubmit operations with additional metadata through the packet byte stream, using the same method for passing intrinsic metadata, simplifying design [1, 3].

TNA introduces additional externs not found in other architectures, such as low-pass filters, weighted random early discard externs, advanced hash externs for CRC computation with custom polynomials, ParserCounter, among others. Moreover, Tofino's set of intrinsic metadata is more extensive than in most P4 architectures, featuring two-level multicasting with source pruning, copy-to-cpu functionality, and IEEE 1588 support [1, 3].

2.1.7 P4 Compiler

P4 compilers [12] are designed to convert P4 programs into specialized configuration binaries, tailored for execution on P4 targets. This explanation primarily focuses on compilers that

follow the commonly used two-layer model.

Two-layer compiler model

P4 compilers predominantly adopt a two-layer structure, comprising a universal frontend and a target-specific backend. The frontend, consistent across all targets, handles parsing, and syntactic and target-independent semantic analysis of the P4 program. This process results in an intermediate representation (IR), which the target-specific backend then utilizes to perform transformations tailored to the specific target [1, 3].

The initial P4 compiler for P4-14, developed in Python, utilized a high-level intermediate representation (HLIR) to depict P4-14 programs as a hierarchy of Python objects, known as p4-hlir. In contrast, the newer P4 compiler, p4c [12], is built in C++ and employs an IR based on C++ objects. This IR can be outputted as either a P4-16 program or a JSON file, facilitating program analysis tool development without necessitating compiler modifications. Figure 2.8 illustrates this structure and its operational principle [1, 3].

P4c [12] includes a generic frontend that supports both P4-14 and P4-16 codes for any architecture. It features several reference backends for bmv2 [9], eBPF, and uBPF P4 targets, along with a testing backend and another for generating control flow graphs of P4 programs. Additionally, p4c incorporates a "mid-end", a collection of generic transformation passes used by reference backends and available for vendor-specific backends. P4.org is responsible for the development and maintenance of this compiler [1, 3].

P4 target vendors create their own compilers incorporating the common frontend. This approach maintains the consistency of the language across various compilers [1, 3].

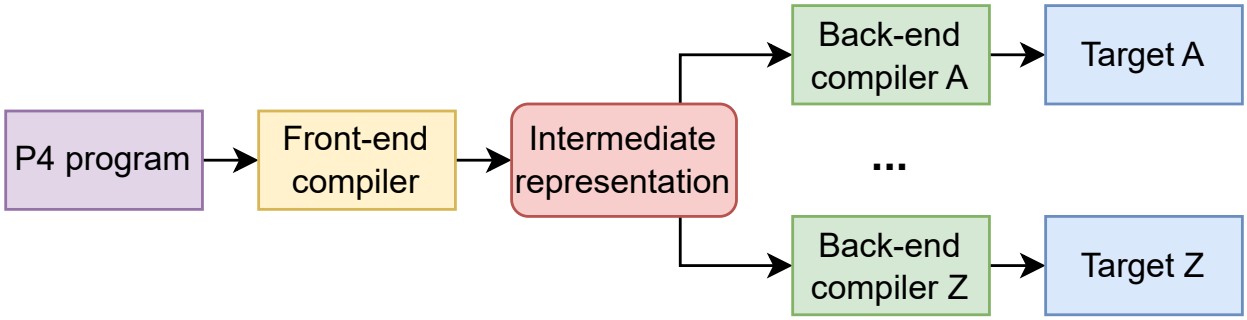


Figure 2.8 The two-layer model of P4 compilers operates with a structured approach. In this model, the front-end compiler first converts the P4 program into an intermediate representation. Following this, back-end compilers take over, compiling this intermediate representation into code specific to the target hardware [1].

2.2 Related Work on P4 Modularity

Research on P4 modularity has explored diverse approaches, including daPIPE, µP4, Lyra, P4 Weaver, and P4 Ansible. Each aims to improve code reuse, portability, and extensibility in programmable data planes through different compiler, language, or incremental programming techniques.

2.2.1 daPIPE

daPIPE [13] introduces one of the earliest concrete environments for incremental programming in P4, targeting hybrid deployments of vendor switches such as Cisco's Nexus 3400 series based on Barefoot's Tofino chip. The switch is shipped with a pre-installed base data plane program and a vendor operating system (NXOS) in this deployment model. Still, customers may wish to add new functions without discarding or rewriting the base program. Writing standalone P4 programs for white-box switches is possible, but this places a heavy burden on customers who would need to re-implement standard features such as L2/L3 forwarding, ACLs, and multicast routing. Incremental programming offers a middle ground: users extend the vendor's program with a small amount of custom code, while the vendor retains responsibility for the correctness and support of the base program.

Achieving this vision requires addressing several challenges. First, incremental code must not break existing, well-tested base functionality. Second, it must not compromise the interface between the data plane and the control plane, since NXOS depends on that interface for correct operation. Third, vendors typically do not want to expose their proprietary P4 code to customers. In contrast, customers likewise should not be forced to study the base code to add a few lines of functionality. Finally, hardware resources such as TCAM, SRAM, and ALUs must be safely shared between base and incremental programs. daPIPE addresses these challenges by constraining what extensions can do: incremental programmers are permitted to add new headers, parser states, tables, and actions, but they cannot modify existing ones. These constraints ensure isolation, preserve vendor intellectual property, and maintain compatibility with the preloaded operating system.

The architecture of daPIPE reflects these constraints. It provides a graphical development environment that guides incremental programmers through writing new headers, parsers, tables, and actions, while exposing only controlled "views" of the base program. For example, the Header View reveals metadata and headers available for reuse, but not the vendor's actual code. The Parser View allows new parser states to be attached to hook points automatically extracted from the base parser, making it possible to merge independent parsing graphs

without direct access to source code. The Table and Action Views allow new tables to be defined and safely combined with existing ones, ensuring that base tables remain untouched. When the incremental code is compiled, daPIPE merges it with the base program inside a secure build environment using Barefoot's official software development kit (SDE). The resulting binary is uploaded to the switch, but error messages visible to the programmer never reveal the base code. Control plane support is similarly isolated: daPIPE exposes only the interfaces for the newly added functionality via a Thrift server, while existing base program APIs remain under NXOS control.

The system was evaluated on a real use case from the broadcast media industry: timestamp switching for multicast RTP flows. This feature requires rewriting the destination IP address based on RTP timestamp values, a capability not supported by any commercial switch at the time. With daPIPE, the incremental programmer could implement only the timestamp switching logic, while reusing the vendor's existing multicast forwarding implementation and NXOS control logic. This demonstrated the practical value of incremental programming: customers could add specialized features with minimal effort, without re-implementing the entire data plane or compromising vendor support guarantees.

While daPIPE effectively demonstrates incremental programming, it also highlights essential limitations. Because base code cannot be modified, some use cases cannot be supported—for example, changing the size or keys of existing tables, or altering algorithms already implemented by the vendor. Resource allocation is left to the backend compiler, which may fail if the incremental program requires more memory or ALUs than are available after the base program's allocation. Debugging also becomes more complex: since vendors and customers have visibility into different parts of the merged program, identifying whether a bug lies in the base or incremental code requires new forms of tooling and cooperative support models. These limitations suggest that incremental programming introduces a new set of engineering challenges distinct from those of writing monolithic P4 programs.

For this thesis, daPIPE is particularly relevant as it marks the first serious attempt to merge vendor-supplied and customer-supplied P4 code in production environments. Its design strongly emphasizes reliability and intellectual property protection, achieved through additive-only extensions and controlled visibility. However, its rigidity also limits expressiveness and reusability. In contrast, the approach explored in this thesis seeks to provide modularity and extensibility more directly within P4 itself. daPIPE demonstrates that incremental programming is both feasible and valuable, but also that without deeper compiler support, extensibility remains heavily constrained.

2.2.2 µP4

One of the most influential attempts to address modularity in data plane programming is µP4 [14]. The authors identify three obstacles in the current P4 model: programs are written in a monolithic style with globally shared headers and metadata; parsers, controls, and deparsers rely on heterogeneous sub-languages that lack a uniform interface; and programs are tightly coupled to specific architectures such as V1Model or PSA, which undermines portability. These challenges make it extremely difficult to isolate protocol functionality, compose independent programs, or reuse existing code across targets.

To overcome these limitations, µP4 introduces a logical architecture, called µPA, and a compiler, µP4C, that together enable modular and target-agnostic data plane programming. Each µP4 module is a self-contained unit that processes a portion of the packet and exchanges information through well-defined logical buffers rather than global metadata. The framework defines explicit interfaces—Unicast, Multicast, and Orchestration—that specify how modules can invoke one another, ensuring that functionality can be combined without breaking encapsulation. This design is conceptually inspired by the Click modular router, but adapted to the constraints of P4 data planes.

The compiler plays a central role in realizing this vision. µP4C transforms programs in three phases: a frontend that translates µP4 into an intermediate representation, a midend that homogenizes all packet-processing blocks into match-action tables, and a backend that maps the resulting program to specific hardware targets. In particular, the homogenization step is crucial: by compiling parsers and deparsers into MATs, the system creates a uniform abstract machine that supports seamless transfer of control between independent modules. This approach enables complex forms of composition, such as building a modular router by combining Ethernet and IPv4/IPv6 modules, or incrementally extending the router with SRv6 support. µP4 also demonstrates the ability to compose higher-level network functions such as firewall and NAT, realizing operators like sequential composition, override, and even A/B testing, all within the data plane itself.

Although µP4 demonstrates that modular programming for data planes is feasible, it comes with important limitations. Despite keeping syntax changes relatively small, µP4 still requires developers to adopt a language variant and learn new logical externs, creating a learning curve and reducing backward compatibility with existing P4 programs. Moreover, resource overheads are non-negligible: transforming parsers into MATs increases the number of pipeline stages and stresses PHV containers on Tofino, with overheads sometimes reaching 1.5× compared to equivalent monolithic programs. The prototype also lacks support for multicast/orchestration interfaces, recursion, and stateful processing, and thus cannot yet

handle the full range of realistic data plane applications.

For this thesis, µP4 is particularly relevant because it represents a compiler-driven attempt to decouple modularity from hardware architecture. However, its main drawback is that it forces programmers to abandon plain P4 and rewrite functionality in a new variant, which undermines reuse of the large body of existing P4 code. By contrast, the approach taken here aims to preserve modularity within P4 itself, either through compiler-managed reuse or lightweight language extensions that remain compatible with the standard ecosystem. In this sense, µP4 illustrates both the promise of modular data plane programming and the pitfalls of solutions that break compatibility with the broader P4 community.

2.2.3 Lyra

Another significant attempt to simplify modularity and portability in data plane programming is Lyra [15]. Lyra was motivated by the fact that modern data center networks (DCNs) are inherently heterogeneous, often combining programmable ASICs from multiple vendors such as Broadcom's Trident-4, Barefoot's Tofino, and Cisco's Silicon One. Writing P4 or Network Programming Language (NPL) programs for these devices directly is comparable to the assembly language era in software: each program is tightly coupled to vendor-specific details, has to be tuned separately for each ASIC, and becomes nearly impossible to maintain when scaled across a network. The authors argue that the fundamental missing piece is a high-level, cross-platform language for the data plane that can guarantee portability, extensibility, and composition.

Lyra introduces such a high-level language by offering a one-big-pipeline (OBP) abstraction. Instead of focusing on chip-level constructs like tables or registers, Lyra allows programmers to describe algorithms using intuitive semantics such as if-else statements, arithmetic operations, and function calls. These algorithms are grouped into a single pipeline, which can span multiple switches and ASIC types. Crucially, Lyra also introduces the notion of algorithm scope, enabling operators to specify where particular functionality should be deployed (e.g., a stateful load balancer only on ToR switches). This abstraction removes the burden of reasoning about hardware resource limits or pipeline constraints, which are then handled by the compiler.

The Lyra compiler translates these high-level programs into runnable P4 or NPL code. Its frontend generates a context-aware intermediate representation (IR) by expanding functions, flattening conditionals into predicates, and converting programs into static single-assignment form. The backend then performs synthesis and constraint solving. Unlike approaches based on integer linear programming, Lyra encodes resource allocation and placement as an Satisfi-

ability Modulo Theories (Solver) (SMT) problem that accounts for instruction dependencies, topology constraints, and heterogeneous ASIC resource models. The solver then produces a placement strategy that splits or distributes algorithms across switches as needed. This allows Lyra to automatically generate chip-specific implementations for an entire network deployment while preserving the high-level program's logical semantics.

The evaluation of Lyra shows substantial improvements in resource usage and code simplicity. Across several case studies, Lyra-generated programs consumed up to 87.5% fewer hardware resources and required up to 78% fewer lines of code compared to human-written P4 and NPL implementations. For example, Lyra could deploy in-band network telemetry (INT) and a stateful L4 load balancer across heterogeneous ToR and aggregation switches in a large-scale DCN without requiring programmers to manually coordinate table sizes, pipeline stages, or language differences between P4 and NPL.

Nevertheless, Lyra also has limitations. Because it is a new language, it requires programmers to learn a fresh syntax and programming model, which reduces backward compatibility with existing P4 programs. While Lyra hides low-level details, it still exposes global, internal, and external variables that may not map naturally to all architectures, and splitting algorithms across devices requires packet header modifications to pass intermediate state. Furthermore, while the compiler achieves strong results with SMT-based synthesis, the scalability of this approach to very large programs or dynamic runtime adaptation remains uncertain.

For this thesis, Lyra is relevant because it demonstrates the value of a compiler- and language-based abstraction layer to address portability and composition in programmable data planes. However, the key distinction is that Lyra abandons native P4 in favor of an entirely new language, whereas the approach proposed here seeks to extend or reuse P4 itself to achieve modularity. This difference is crucial: Lyra proves that high-level abstractions can resolve the portability crisis in heterogeneous ASICs, but it does so at the cost of compatibility with the broader P4 ecosystem. In contrast, the work in this thesis pursues modularity without requiring a departure from the P4 language, thereby offering a path toward adoption that builds directly on existing tools and codebases.

2.2.4 P4 Weaver

P4 Weaver [16] was designed specifically to address the problem of incremental and modular programming in P4, particularly in scenarios where a vendor provides a base switch program and customers or third parties wish to extend it with new functionality. In practice, this reflects the most common deployment model: vendors ship switches with preloaded programs implementing standard features such as L2/L3 forwarding, ACLs, or tunneling,

while customers may want to add specialized features, for example new tunneling protocols or telemetry headers. Extending such vendor code directly is error-prone and risks breaking essential features or violating control plane contracts. Moreover, vendors often wish to protect the intellectual property of their base programs, making it impossible or undesirable to expose the source code.

P4 Weaver introduces a principled annotation system coupled with a source-to-source compiler to address these challenges. Vendors annotate their base code to specify what extensions are permissible, using constructs such as @Header, @State, and @Table. These annotations control visibility (read, write, or name-only access to headers), indicate where parser states can be extended, and define hook points before or after table applications. On the customer side, extension programs are written in standard P4, augmented with annotations such as @Parser, @Control, and @Deparser that bind new functionality to the annotated hook points in the base program. The Weaver compiler then merges the two annotated Abstract Syntax Tree (AST)s into a unified P4 program, enforcing the vendor's constraints and ensuring syntactic and semantic consistency.

A key design principle of P4 Weaver is that extensions must be additive only. Customers can add new headers, parser states, tables, actions, or deparser logic, but cannot remove or modify existing elements of the vendor code. This conservative choice guarantees that base functionality and control plane APIs remain intact, and that customers cannot accidentally break critical forwarding behavior. To enforce this, Weaver also performs static checks to prevent unauthorized access to restricted headers or invalid control-flow paths. Potential naming conflicts are avoided by automatically aliasing extension identifiers.

The architecture of P4 Weaver further reflects the need for code protection. It adopts a client–server model, in which the vendor's base code is stored on a secure server, invisible to customers. Customers interact with a programming assistant integrated into an IDE (the prototype used VSCode), which exposes only the annotated interfaces of the base program. The IDE supports autocompletion, local syntax checking, and displays vendor-defined extension points. Extension code written by the customer is sent to the server, where the Weaver merges it with the base code, invokes the vendor compiler (e.g., Barefoot's Tofino compiler), and returns a binary image to the customer for deployment. This model allows modular development while safeguarding vendor intellectual property.

The effectiveness of P4 Weaver was demonstrated through several case studies, including support for Real-time Transport Protocol (RTP) timestamp switching, the General Packet Radio Service (GPRS) tunneling protocol (GTP) with Equal-Cost Multi-Path (ECMP) load balancing, and in-band network telemetry (INT). Each of these features required adding new

headers, parser logic, tables, and deparser actions, and P4 Weaver allowed them to be integrated into a vendor base program without manual intervention. Importantly, the overhead introduced by the weaving process was negligible compared to the total P4 compilation time, showing that the approach is practical in deployment.

Despite these advances, P4 Weaver has limitations. By disallowing modifications to base code, it cannot support scenarios where incremental programming requires altering table keys, resizing tables, or removing unused protocols. Resource allocation remains a challenge, since small extensions can change table dependencies and resource consumption in ways not exposed by Weaver. The system also focuses solely on the data plane; extending the control plane in a similarly modular fashion is left for future work. Finally, while using annotations leverages P4's existing extension points, it still requires vendor participation to annotate base programs appropriately, and adoption may be uneven across the ecosystem.

For this thesis, P4 Weaver is highly relevant because it demonstrates that incremental modularity in P4 can be achieved without creating a new language. Instead, it builds directly on P4 through annotations and source-to-source weaving. However, its conservative additive model and reliance on vendor annotations highlight key trade-offs: Weaver prioritizes reliability and IP protection but restricts flexibility. In contrast, the approach pursued here seeks to offer modularity more directly at the compiler or language level, with constructs like inheritance and extension, aiming for both code reuse and backward compatibility without requiring vendor-controlled annotations.

2.2.5 P4 Ansible

P4 Ansible [17] is a prototype framework that extends P4-16 with inheritance-inspired constructs, informally called P4++, to support incremental programming in vendor–customer settings. The core idea is that a vendor supplies a base P4 program, while customers can write their own extensions in P4++. A Python-based shell then merges the two into a single executable program. To make this possible, P4-Ansible introduces new keywords such as override and super, enabling customers to patch existing structs, parser states, controls, or packages rather than rewriting them from scratch. For instance, a customer can override a parser state in the vendor's code, add new states, and redirect transitions, while still calling the original logic through the super keyword.

In principle, this approach promises modular collaboration: the vendor maintains ownership of the base program, and the customer can extend it without breaking existing functionality. However, in practice this model requires that the customer has at least partial visibility into the vendor's code. Customers must know the names and interfaces of the parser states,

controls, or structs they intend to override; without this information, it is impossible to write meaningful extensions. Thus, the vendor must provide either a documented specification or a skeleton of the base program exposing the override points. This makes P4-Ansible less of a black-box solution compared to annotation-based systems like P4 Weaver, which explicitly reveal extension points through vendor-provided annotations and hide the rest of the code.

Moreover, the prototype is incomplete. Parser merging is supported, but fine-grained control over tables and actions is not yet implemented, and deparsers cannot be automatically merged due to header ordering constraints. As a result, P4-Ansible demonstrates a promising but partial vision: it shows that inheritance-like constructs can simplify parser and control extension, but it also highlights the difficulty of supporting vendor–customer modularity without careful exposure of base program details. For this thesis, the significance of P4-Ansible lies in its attempt to embed object-oriented semantics directly into P4. However, its reliance on non-standard keywords and partial visibility requirements limit its practicality in production environments.

2.3 Summary of the Literature, Limitations, and Motivations

Achieving modularity in the P4 programming language remains a fundamental challenge. Although several research efforts have sought to enhance the flexibility and reusability of P4 programs, a common limitation across existing approaches is the lack of integration with the standard P4 language and compiler infrastructure. This section has critically reviewed prior work, identified the limitations of existing approaches, and established the motivation and research direction pursued in this thesis.

2.3.1 Importance of Addressing Modularity in P4 Within the Language and Compile

To understand the importance of native modularity in P4, we can draw parallels with mature programming languages such as C++ and Java. In both languages, modularity and object-oriented programming are handled directly through the language syntax and enforced by the compiler. This direct integration has led to robust, maintainable, and scalable software ecosystems.

Similarly, P4 requires a compiler-native approach to modularity to support advanced development and deployment practices in the data plane. Handling modularity outside the P4 compiler—through external tools, transpilers, or annotation systems—introduces several challenges:

- **Maintenance and Extensibility:** Tools or languages that are not natively designed in the P4 language may pose challenges in maintaining and extending the codebase that handles modularity. Additional layers of tools, such as a transpiler, increase complexity and make it more difficult to manage, debug, and extend the code.
- **Backward Compatibility with preexisting p4 codes:** when integrating with an existing P4 codebase, we should ensure that the integration is seamless and does not introduce inconsistencies or bugs.
- **Learning Curve:** A learning curve might be associated with using external tools, which may impose learning some new syntax.
- **Performance Considerations:** Using external tools can sometimes introduce performance overheads, especially if a transpiled code is less efficient than a native P4 code.
- **Community and Support:** The P4 community prefers to support the development modularity of the P4 by using the P4 language and compilers.

Among the many proposals found in the literature, five works stand out for their direct focus on modularity in P4: **µP4**, **Lyra**, **daPIPE**, **P4 Weaver**, and **P4 Ansible**. Each offers unique mechanisms, but they all share common shortcomings that motivate this thesis.

µP4 proposes a logical architecture (µPA) and a compiler (µP4C) that homogenize all P4 constructs into match–action tables, enabling modular composition of data plane modules. Although powerful, µP4 requires programmers to adopt a new syntax and abandon existing P4 code, making backward compatibility a major limitation.

Lyra introduces a high-level language with a one-big-pipeline abstraction and a solver-based compiler to deploy data plane algorithms across heterogeneous ASICs. While Lyra demonstrates portability and scalability, it completely replaces P4 with a new language, increasing the learning curve and breaking compatibility with existing P4 ecosystems.

daPIPE enables incremental programming in vendor–customer scenarios by constraining customer extensions to additive code that does not modify vendor programs. It successfully isolates resources and preserves control-plane contracts, but requires GUI-driven development and prevents fine-grained modular reuse, limiting expressiveness.

P4 Weaver advances incremental programming by using annotations on vendor programs to expose safe extension points. Customers can extend parsers, controls, and tables through

these annotations while vendors protect their intellectual property. However, Weaver is limited to additive extensions and relies on vendor-prepared annotations, leaving no mechanism for compiler-native modularity.

P4 Ansible takes a different path by introducing inheritance-inspired keywords such as `override` and `super`, allowing customers to patch existing parser states, controls, and actions. While it represents an early attempt to embed object-oriented semantics in P4, it remains incomplete, not fully integrated with the compiler backend, and requires customers to know the names and interfaces of vendor code elements, making it impractical for real deployment.

Also, there are some related works that explore modular approaches to network programmability—one presenting a modular P4 design for flexible 5G fronthaul protocol processing and precise timestamping to enhance synchronization accuracy [18], another proposing Lucid 2.0, a safe and modular packet pipeline programming language that guarantees pipeline-safety through a type system supporting polymorphism and hierarchical abstractions [19], and a third introducing Timepiece, a modular control-plane verification framework based on temporal invariants for scalable reasoning across large network architectures [20].

2.3.2 Motivation and Problem Statement

The survey of prior work reveals a consistent trend: while many approaches attempt to provide modularity for P4 programs, they do so outside the language and compiler itself. Programmers are asked to rely on external annotations, new syntaxes, transpilers, or virtualization mechanisms. These techniques inevitably introduce additional complexity, hinder backward compatibility, and limit adoption within the broader P4 community.

In contrast, mainstream programming languages such as C++ and Java integrate modularity directly into the language and enforce it through the compiler. This integration has been critical to code scalability, reliability, and reuse in large software systems. For P4 to achieve similar maturity in the data plane domain, modularity must likewise be handled natively by the P4 compiler infrastructure.

The central problem, therefore, is that *no existing solution provides compiler-native modularity in P4 while preserving backward compatibility and minimizing disruption to current practice.* This problem motivates two guiding research questions:

1. To what extent can modularity be achieved within the existing P4 language and compiler infrastructure, without introducing new syntax?

2. When language extensions are strictly necessary, what is the minimal and well-scoped

set of new constructs that can enable full modularity while preserving backward compatibility?

Addressing these questions is essential to bridging the gap between external modularity mechanisms and a sustainable, compiler-native approach. By doing so, P4 can evolve towards a robust, maintainable, and scalable programming model comparable to established general-purpose languages.

Table 2.1 **Feature Comparison of P4 Modularity Solutions**

P4 Solutions	Automatic Merging	Backward Compatibility	Vendor-customer Compatibility	No New Syntax or Annotation	Integration with P4 IR	Inheritance
µP4 [14]	No	No	N/A	No	No	No
Lyra [15]	No	N/A	N/A	No	No	No
P4Weaver [16]	No	Yes	Yes	No	No	No
P4Ansible [17]	No	Yes	Yes*	No	No	Yes**
daPIPE [13]	No	Yes	Yes	No	No	No

As summarized in Table 2.1, existing solutions vary significantly in their capabilities. Most approaches fail to provide automatic merging, lack integration with the P4 compiler's intermediate representation (IR), and do not maintain backward compatibility. Many rely on new syntaxes or annotations, limiting their adoption within the P4 community. Furthermore, inheritance support is either absent or implemented in a way that is not compatible with the core P4 compiler framework. These limitations explain why, despite meaningful contributions from prior works, none have achieved comprehensive, compiler-native modularity in P4. In the * and ** entries of the table, in its open-source implementation, P4Ansible provides incomplete inheritance support for core P4 constructs and relies on Bison rather than the P4 intermediate representation.

CHAPTER 3 ARTICLE 1: P4MUSE: ENABLING MODULAR P4 PROGRAMMING VIA COMPILER-MANAGED CODE MERGING WITHOUT SYNTAX MODIFICATIONS

This chapter is a reproduction of an article published in IEEE Access on July 3, 2025. Rahmati, M., Boyer, F.-R., Pontikakis, B., David, J.-P., and Savaria, Y., 2025. P4Muse: Enabling Modular P4 Programming via Compiler-Managed Code Merging Without Syntax Modifications. IEEE Access, vol. 13, pp. 124138–124157.

This article presents the first main contribution of this thesis: the design and implementation of P4Muse, a novel compiler-assisted approach that enables modular programming in P4 without introducing new syntax or annotations. P4Muse extends the open-source P4C compiler with custom passes that automatically merge P4 programs, facilitating protocol reuse and modular composition across data plane components. By supporting seamless integration of multiple P4 functionalities, P4Muse addresses the limitations of existing techniques, such as trans-compilation and virtualization, which often hinder extensibility and backward compatibility. This work demonstrates P4Muse through six case studies across three classes of use case: code reusability, data plane pipeline composition, and vendor-customer integration. The results validate P4Muse's effectiveness on the V1Model architecture using the BMv2 software switch.

3.1 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.

3.2 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 [4, 5], 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 [6, 7] and NPL [21]. 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 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 [13], P4 Weaver [16], P4-Ansible [17], µP4 [14], and Lyra [15], 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 [22] or SRv6 [23], 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 [12] 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 3.2.1 and the related work using p4 modularity in subsection 3.2.2. The goals and challenges of implementing P4 modularity are described in section 3.3. The framework of P4Muse, including its compiler and interface, is reviewed in section 3.4. The methodology and levels of modularity are presented in section 3.5 and section 3.6, respectively. We present classes of use cases in section 3.7, followed by case studies and evaluation in section 3.8 and section 3.9. The challenges of modularity and the current limitations of P4 are discussed in section 3.10, while the limitations of our work and directions for future research are covered in section 3.11. Finally, the paper concludes in section 3.12.

3.2.1 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.

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 [24] 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 [25, 26], 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 [27], 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.

Data Plane Composition and Virtualization

Data plane composition and virtualization research efforts have explored modularity through virtualization approaches [28].

HyPer4 [29] 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 [30] and later HyperVDP [31] 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 [32] 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 [33]. P4VBox [34], unlike HyPer4 and HyperVDP, reengineers parallel lookup modules using Hardware Description Language (HDL) to support multiple programs simultaneously on the data plane.

P4Visor [33, 35] 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 [36, 37] 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.

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 [38], 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.

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.

3.2.2 Related Works using P4 Modularity

Service Function Chain (SFC) Frameworks

Service function chain (SFC) frameworks [39–42] facilitate the flexible composition of network 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.

Runtime Programmability

Research on runtime programmability [43–47] 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.

3.3 Goals and Challenges for Implementing P4 Modularity

3.3.1 Design Goals of P4Muse

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 3.11, we will discuss potential future enhancements for referencing these objects.

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.

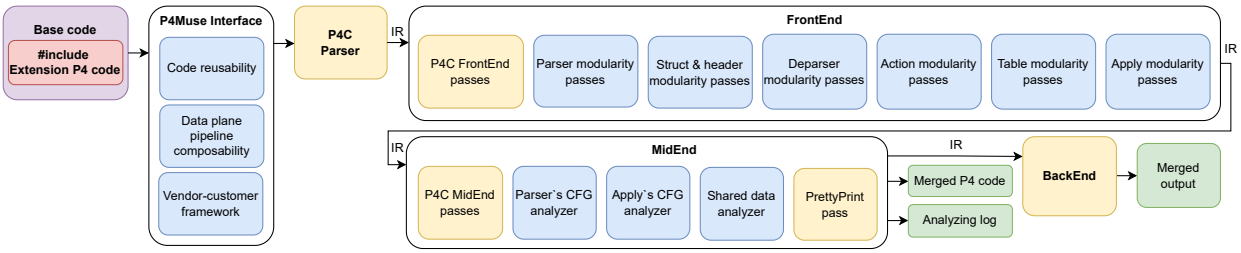


Figure 3.1 Overview of the P4Muse compiler showing its modular interface options and compiler stages.

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.

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.

3.3.2 Challenges in Adapting P4 for Modularity

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.

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.

3.4 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 3.1 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.

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.

P4Muse Compiler

Figure 3.1 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.

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

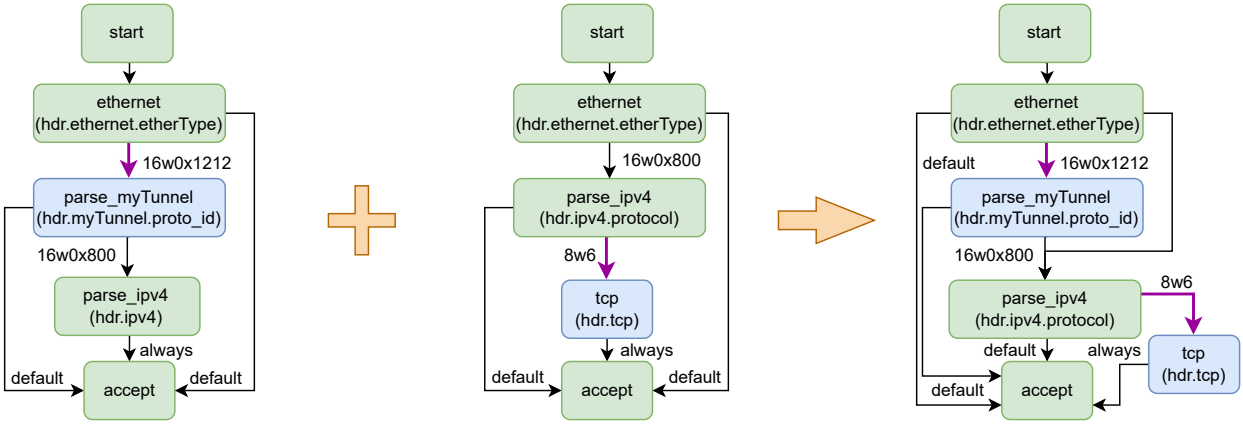


Figure 3.3 Parser graphs of an advanced tunnel, a firewall, and the corresponding merged P4 codes. Note that the firewall's acceptance or rejection of packets occurs in the match-action unit, not in the parser stage.

3.5 Methodology for Implementing the Proposed P4 Modularity

3.5.1 Methodology of Parser

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.

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 3.2, 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 3.2, which shows the codes for ‘ExtensionParserFirewall‘, ‘BaseParserAdvanced-Tunnel‘, 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.

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

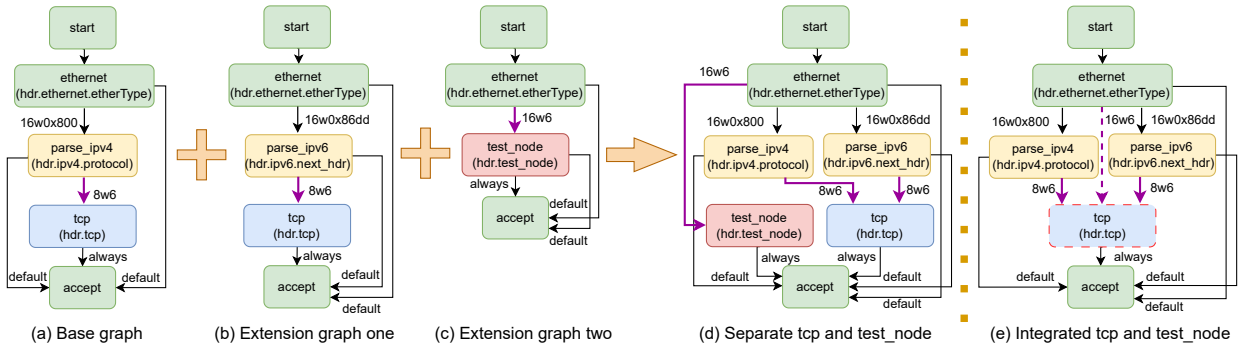


Figure 3.4 Comparison between integrated and separate parsing strategies with different graphs.

enable comparison between objects in the extension and base codes.

Unlike prior approaches such as PRIME and its improved version [36, 37], 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) [48]. 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 3.3 presents the parser graphs corresponding to Figure 3.2. Key nodes include ‘start,‘ ‘parse_ethernet,‘ ‘parse_myTunnel‘ with transition value "16w0x1212," ‘parse_ipv4‘ with transition value "16w0x800," and terminal states ‘accept‘ and ‘reject‘ for the base code, as well as ‘start,‘ ‘parse_ethernet,‘ ‘parse_ipv4‘ with transition value "16w0x800," ‘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.

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 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 3.4, 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 3.4. 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 3.4(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 3.2, `parse_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 3.2, 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 3.2, 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.

Algorithm 1: Parser Comparison Algorithm

```
Input: baseObjTree, extObjTree
Output: Updated extObjTree
Function compareParser(baseObjTree, extObjTree):
    baseStates ← map of (state.transitionVal → state) foreach state in baseObjTree;
    foreach extState in extObjTree do
        extTransition ← extState.transitionVal;
        if extTransition in baseStates then
            if compareHeader (baseStates[extTransition], extState) then
                compareChildTransitions (baseStates[extTransition], extState);
        else
            // Situation 3:  Different Nodes
            extState.requiresMerging ← true;
Function compareChildTransitions (baseState, extState):
    if childTransitionVals(baseState) ≠ childTransitionVals(extState) then
        // Situation 2:  Mergeable Nodes
        extState.childTransitionsRequireMerging ← true;
        foreach transition in extState.transitions not in baseState.transitions do
            transition.requiresMerging ← true;
    // Else:  Situation 1:  Equivalent Nodes
Function childTransitionVals(state):
    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
extSetAddState ← set of extObj for each extObj in extObjTree if
  (extObj.requiresMerging);
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)) ;
processedAddState ← false;
processedMergeTransition map of (baseObj.transitionVal → false) (for each
  baseObj in baseObjTree);
Function preorder(IR::parserState node by reference):
    // Adding whole states
    if (not processedAddState) and (getTransitionVal not in
      extSetMergeTransition) then
        parserStateVector ← IR::vector of node;
        if extObj in extSetAddState then
            parserStateVector.append(extObj);
        processedAddState ← true;
        return parserStateVector;
    // Merging child transitions
    if not processedMergeTransition[getTransitionVal ] then
        if getTransitionVal in extSetMergeTransition then
            selectCaseVector ← IR::vector of existingCase for each existingCase in
              node.selectExpression.selectCases;
            selectCaseVector.append(extObj.transition) for each extObj.transition in
              extObj.transitions;
            node.selectExpression ← new IR::selectExpression(
              node.selectExpression.srcInfo, node.selectExpression.select,
              selectCaseVector);
        processedMergeTransition[getTransitionVal ] ← true;
    return node;
```

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:

- **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.

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 3.5, 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 3.5e as blue arrow). However, if graph 'd' were the base, the default transition would instead be considered as "start_0" (shown on Figure 3.5e as red arrow). Additionally, "start_0", which represents the inlined sub-parser's accept state, shares the same transition value (16w0x86DD) 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.

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.

3.5.2 Methodology for the Struct and Header Data Type

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 3.6, for example, the "packet.extract(hdr.myTunnel)" call in the "parse_myTunnel"

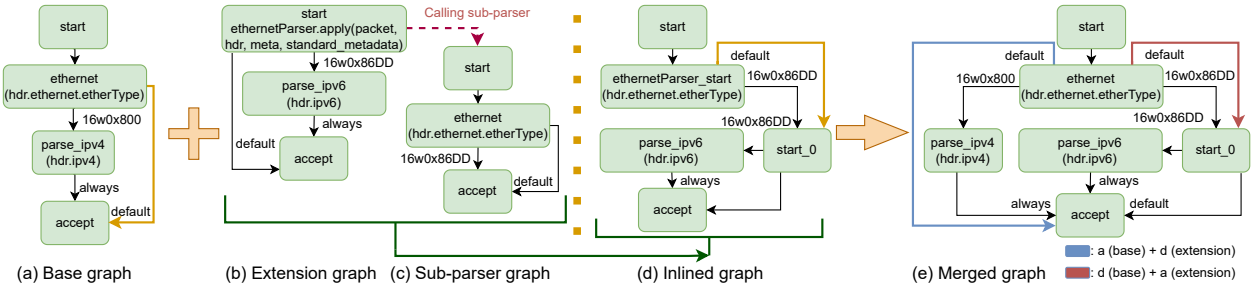


Figure 3.5 Parser graphs transformation, including sub-parser inlining and final merging.

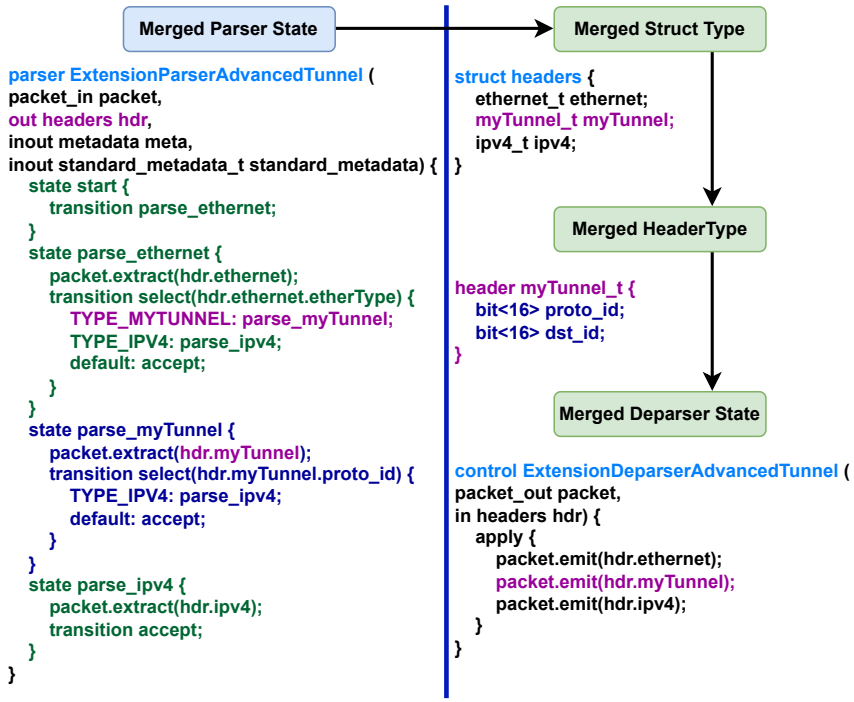


Figure 3.6 Parser, struct, header, and deparser blocks of advanced tunnel P4 code.

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.

Name Standardization for Structs and Headers

As shown in Figure 3.2, 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 3.7, 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.

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.

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 3.7, are present.

3.5.3 Methodology for the Deparser

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 3.6, 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.

Standardizing Deparser Names

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

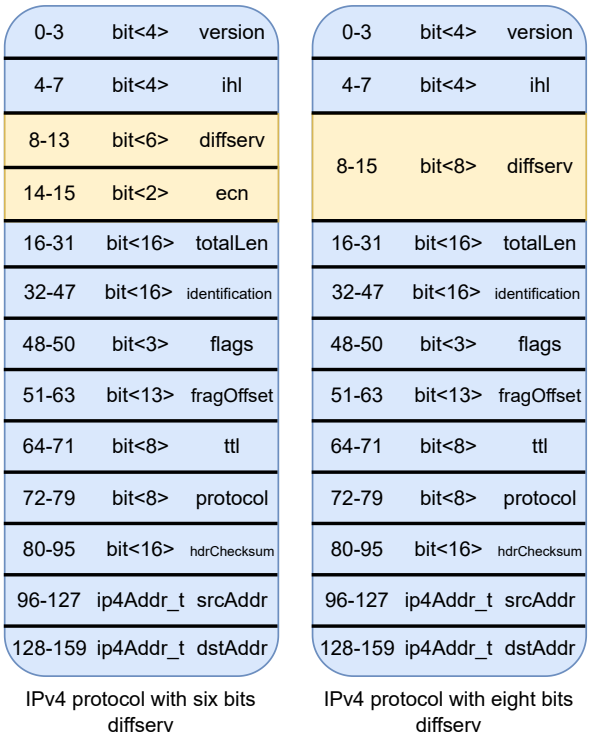


Figure 3.7 Comparison of IPv4 header formats with two kinds of DiffServ field lengths

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.

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.

3.5.4 Methodology for the Control

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.

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.

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.

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.

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);
```

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.

- **`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.

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;
```

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.

- **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.

3.5.5 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.

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.

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.

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.

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.

3.6 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.

3.6.1 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.

3.6.2 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.

3.6.3 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 3.2 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.

3.7 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.

3.7.1 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.

3.7.2 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 3.8, 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.

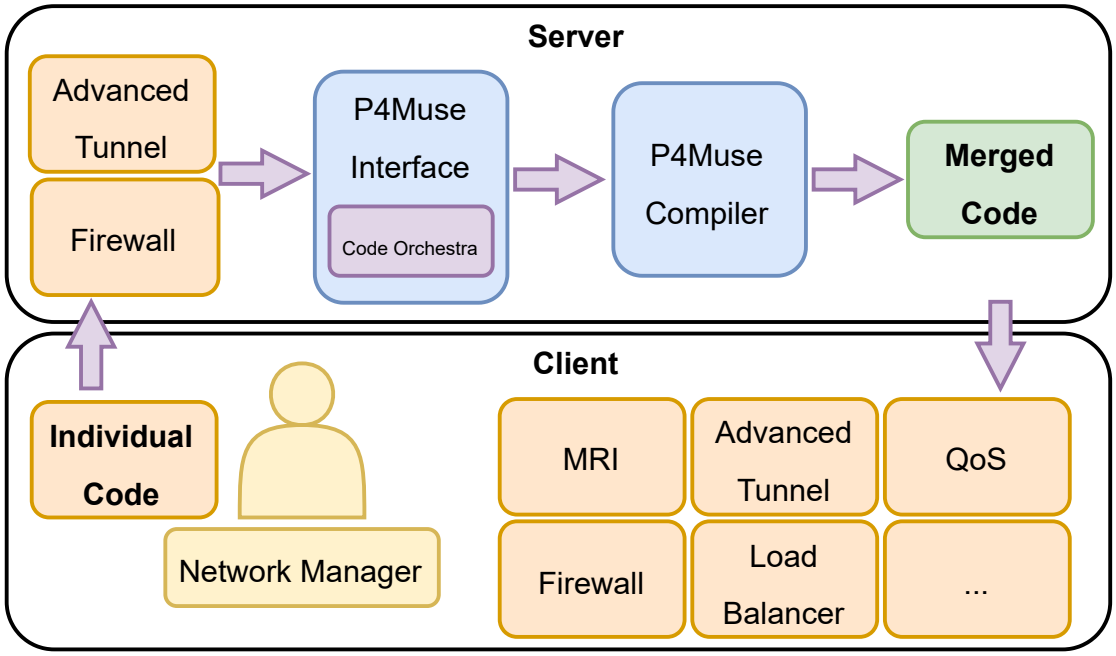


Figure 3.8 Client-server framework for data plane pipeline composability

3.7.3 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 [16] used for P4Muse without a new annotation system, illustrated in Figure 3.9, 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.

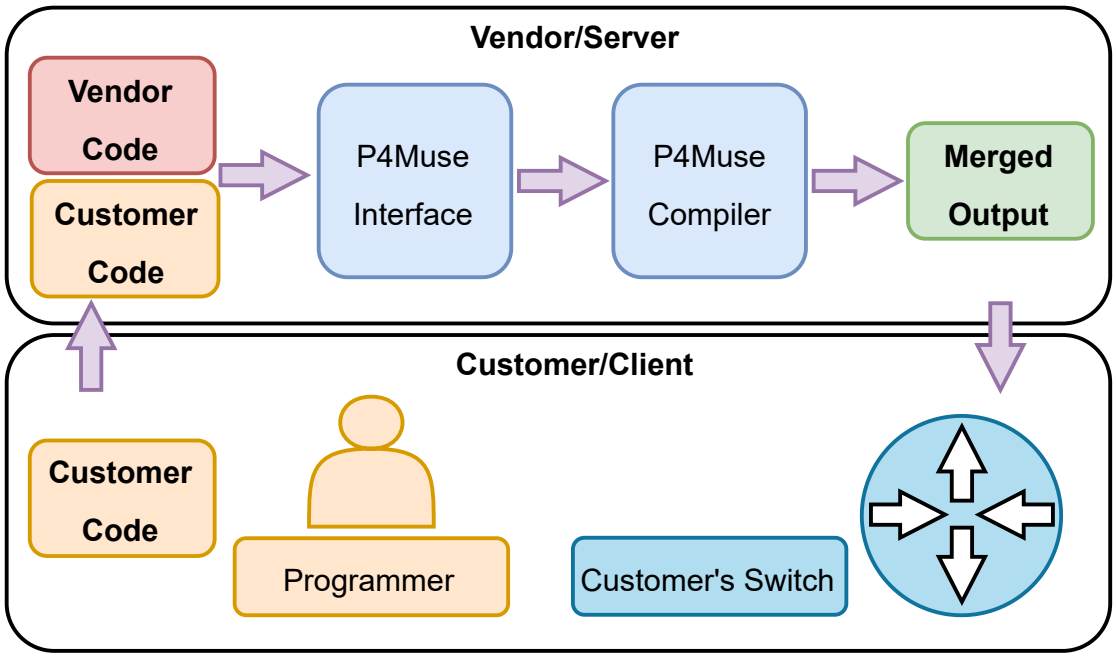


Figure 3.9 Client-server framework for vendor-customer collaboration

3.8 Case Studies of P4Muse Modularity

To illustrate the benefits and effectiveness of P4Muse, we present six case studies based on the P4 tutorials [2] and the Next-Gen SDN tutorial [49], exploring how P4Muse enhances modularity at both the protocol and application levels, improves security and performance,

and demonstrates its extensibility across three different system configurations.

3.8.1 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.

3.8.2 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.

3.8.3 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.

3.8.4 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 functionalities and address performance bottlenecks through modular design.

3.8.5 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.

3.8.6 SRv6 and NDP: Complex Application Integration

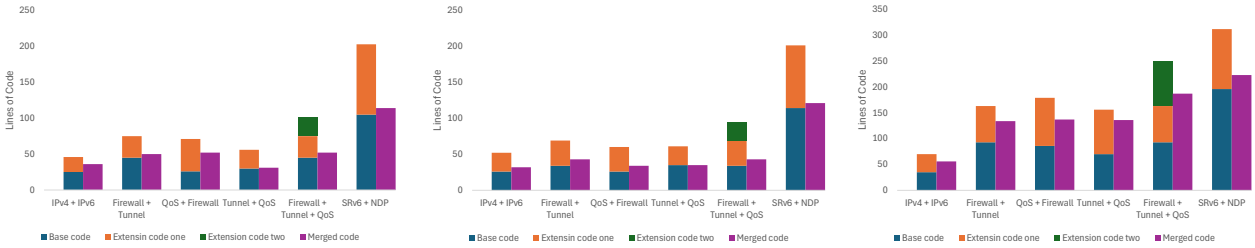
A major objective of P4 modularity is the ability to merge complex applications. In this case study, we explore combining advanced applications—SRv6 (Segment Routing over IPv6 data plane) and NDP (Neighbor Discovery Protocol). This integration highlights the potential of the modular framework to support 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.

3.9 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.

3.9.1 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 3.10a, we compare the code lines in the header and struct sections between the merged code and the combined base and extension code. Figure 3.10b shows a similar comparison for the parser and deparser sections. In contrast, Figure 3.10c reports the number of code lines in the ingress and egress sections, highlighting the code reduction achieved through merging.



(a) Header and struct components

(b) Parser and deparser components

(c) Ingress and egress components

Figure 3.10 Comparison of lines of code across components in Base, Extension, and Merged Codes.

3.9.2 Performance Analysis of P4Muse

To assess the performance impact of P4Muse, we analyzed the throughput and latency of the merged code compared to the separate base and extension codes across six case studies. In Figure 3.11, the latency of the merged code is compared with that of the base and extension codes combined. Similarly, Figure 3.12 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 3.11, 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 3.12 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.

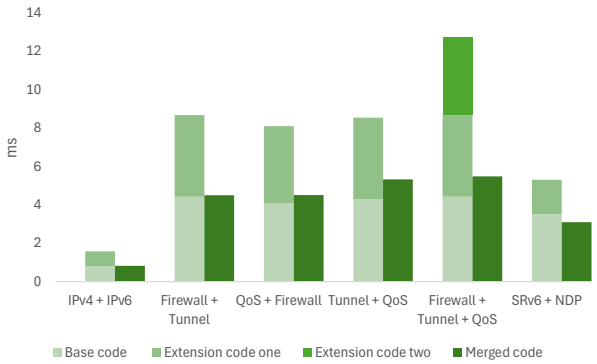


Figure 3.11 Comparison of Latency Between Base, Extension, and Merged Codes

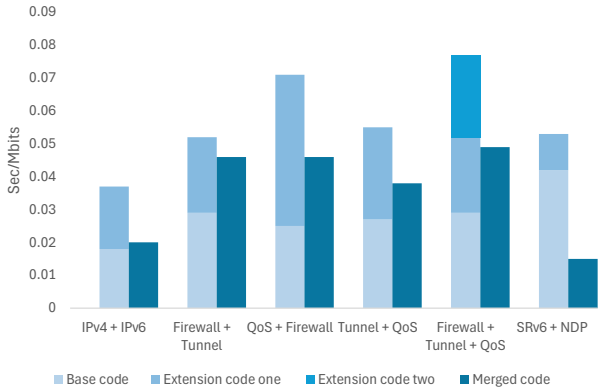


Figure 3.12 Comparison of Reciprocal of Throughput Between Base, Extension, and Merged Codes

3.9.3 Comparison with Related Work

Table 3.1 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.

3.10 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 com-

Table 3.1 **Feature Comparison of P4 Modularity Solutions**

P4 Solutions	Automatic Merging	Backward Compatibility	Vendor-Customer Compatibility	No New Syntax & Annotation
µP4 [14]	No	No	N/A	No
Lyra [15]	No	N/A	N/A	No
P4Weaver [16]	No	Yes	Yes	No
P4Ansible [17]	No	Yes	Yes	No
daPIPE [13]	No	Yes	Yes	No
P4Muse	Yes	Yes	Yes	Yes

ponents. Addressing these limitations could guide future research and development efforts within the P4 community to enable built-in modular and object-oriented support.

3.10.1 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.

3.10.2 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.

3.10.3 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.

3.10.4 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.

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

3.11 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.

3.11.1 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 3.13, 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 3.6, 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.

3.12 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.

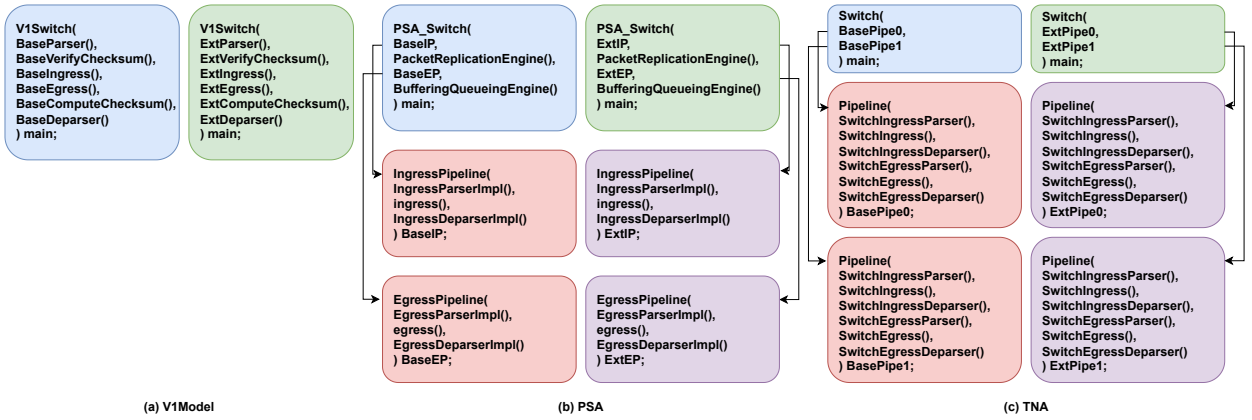


Figure 3.13 Modular comparison workflow across V1Model, PSA, and TNA architectures.

CHAPTER 4 ARTICLE 2: P4O2: ENABLING OBJECT-ORIENTED-INSPIRED MODULARITY IN P4

This chapter is a reproduction of an article submitted to IEEE Access on February 10, 2025. Rahmati, M., Boyer, F.-R., Pontikakis, B., David, J.-P., and Savaria, Y. P4O2: Enabling Object-Oriented-Inspired Modularity in P4. Submitted to IEEE Access.

This article presents the second main contribution of this thesis: the design and implementation of P4O2, a lightweight and backward-compatible extension to the P4 language that introduces structured modularity inspired by object-oriented programming (OOP) principles. P4O2 extends the open-source P4C compiler with new language constructs that enable incremental extension and reuse of key components such as structs, parsers, and controls, fostering deep composability in data plane programs. By allowing modular design directly at the language level, P4O2 addresses the limitations of monolithic P4 development, enabling scalable construction of protocol libraries and application-specific components without sacrificing performance or compatibility. P4O2 is demonstrated through three case studies using the V1Model architecture, showcasing modular implementations of protocol stacks and network applications. All evaluations are performed on the BMv2 software switch, validating the practicality and effectiveness of P4O2 in real-world programmable networking scenarios.

4.1 Abstract

Dataplane programming languages such as P4 offer flexible and high-performance packet processing capabilities, enabling the expression of diverse behaviors in Software-Defined Network (SDN) environments. However, traditional P4 programs are typically monolithic, which limits modularity, code reuse, maintainability, and ease of debugging. This paper presents P4 Object-Oriented-Inspired (P4O2), a lightweight and intuitive superset of the P4 language that implements structured modularity through object-oriented programming (OOP) principles. The P4O2 framework allows developers to build upon essential components, including structs, parsers, and controls, through incremental extension and reuse, thereby promoting deep, composable modularity and facilitating testing and debugging. P4O2 integrates seamlessly with the open-source P4C compiler and supports all existing P4 programs without disrupting legacy code. We evaluate P4O2 through three V1Model architecture–based case studies, demonstrating how it enables modular protocol stacks and applications. Experiments on the BMv2 software switch and V1Model architecture confirm that P4O2 works effectively.

4.2 Introduction

P4 [6] is a programming language for network device data planes, including switches, routers, and smartNICs [1, 50]. It enables developers to specify packet parsing, match-action table processing, and packet reassembly at a basic programming level. Beyond these fundamental capabilities, P4 provides direct support for complex, high-performance packet processing tasks in the data plane, making it highly valuable in modern networking environments such as SDN [4, 51] 5G infrastructures, and Network Function Virtualization (NFV). By using P4, network operators can design custom protocols and processing logic, allowing them to quickly innovate and adapt to evolving requirements through software modifications rather than hardware upgrades.

However, P4's flexibility and power do not include the built-in modularity support needed to manage complexity in large-scale or evolving network programs. The monolithic nature of traditional P4 code makes it difficult to reuse components, extend protocol implementations, and organize code into clear, composable units. This lack of modularity becomes a critical challenge when implementing layered protocol stacks.

The absence of modularity forces developers to duplicate code or perform manual integration, which results in code redundancy, higher maintenance costs, and higher error probabilities. Adding modularity to P4 would enable better separation of concerns, code reuse, and improved collaboration, resulting in more scalable and maintainable P4 programs.

The best solution to enable modularity is to follow an object-oriented design philosophy, which provides a natural and structured way to organize and extend code. **Object-oriented programming (OOP)** encourages building software from reusable and extensible components, making it easier to manage complexity and scale development. However, since P4 is a domain-specific language (DSL) focused on packet processing, it benefits most from a lightweight adaptation of OOP that introduces only the minimal constructs necessary to support modularity. To do this, we implement P4O2, an object-oriented-inspired extension of P4 that enables modular development through selective inheritance and reusable components, while preserving P4's simplicity.

The P4 ecosystem has received multiple previous attempts at modularity through daPIPE [13], P4 Weaver [16], P4-Ansible [17], µP4 [14], Lyra [15], and P4Muse [52]. These approaches either depend on external tools, lack native syntax, or fail to support inheritance at the IR level.

P4O2 (P4 Object-Oriented-Inspired) extension addresses these limitations by providing a lightweight backward-compatible extension to the P4 language, which integrates modular

programming principles including inheritance and composability, directly into the language syntax and compiler. P4O2 allows developers to create P4 code that is cleaner, reusable, and extensible, which leads to improved productivity and maintainability across different network applications.

This paper makes the following key contributions:

- **Language Extension for Inheritance:** The P4O2 language extension, based on object-oriented principles, allows direct inheritance implementation within P4 syntax.
- **Compiler Support for Inheritance:** The P4 compiler [12] infrastructure receives an extension which enables inheritance across all language components, including `structs`, `parsers`, and `control` directly and `header` and `deparser` indirectly. The compiler performs inheritance relationship validation, reference resolution, and merges reusable logic across program components.
- **Backward Compatibility and Lightweight Design:** The P4O2 system maintains complete backward compatibility with existing P4 programs. Adopting P4O2 allows developers to transition gradually without rewriting existing code, enabling easy integration into current projects and tool chains. The minimal syntactic additions in P4O2 to standard P4 result in a low learning curve while maintaining a familiar interface for P4 developers.
- **Case Studies and Evaluation:** The practicality and expressiveness of P4O2 is demonstrated through three case studies which implement protocol stacks including IPv4, tunneling, firewall, QoS, and SRv6 [53] using BMv2 software switch [54] and V1Model architecture [10].

The remainder of this paper is organized as follows. We summarize the related work in section 4.3 . The goals and challenges of implementing P4 modularity are described in section 4.4. The framework of P4O2, including its compiler and syntax, is presented in section 4.5. The methodology is presented in section 4.6 and followed by a parser modularity style in section 4.7. We present case studies in section 4.8 and results in section 4.9. Finally, the paper concludes in section 4.10.

4.3 Background of the proposed P4O2

The research community has made various attempts to address the modularity limitations of P4. ClickP4 [24] and related works [25–27] propose early steps toward modularity in

P4 programming, enabling developers to compose P4 programs from reusable components. However, these approaches require manual integration of modules and do not support the reuse of existing P4 code as-is. As a result, they lack the flexibility and composability expected in modern programming environments, highlighting the need for more automated and reusable modular P4 solutions.

HyPer4 [28, 29] introduced the first P4-hypervisor and a versatile table design, paving the way for follow-up work such as HyperV and HyperVDP [30,31], which optimized parsing and action processing using a description header. Other approaches, such as P4Bricks [32] and P4VBox [34], enabled multi-program composition through configuration merging or HDL-based designs. P4Visor [33, 35] focused on compiler-level merging of P4 programs but is limited to $P4_{14}$ and lacks full modular support. PRIME [36, 37] advanced composition techniques but required additional tables and recirculation logic, introducing hardware overhead. Collectively, these efforts demonstrate significant progress in virtualizing and composing P4 programs but highlight persistent challenges in modularity, efficiency, and compatibility with $P4_{16}$.

Recent approaches such as µP4 and Lyra aim to improve modularity and portability in data plane programming by introducing intermediate abstractions or unified pipeline models. P4All [38] further contributes with an adaptive data structure that enables code reuse across varying hardware platforms. However, these methods often introduce new syntactic frameworks and lack backward compatibility with existing P4 code, limiting their adoption.

daPIPE and P4 Weaver advance incremental programming in P4 by enabling structured vendor and customer code integration while preserving intellectual property. daPIPE provides a GUI for editable regions, whereas P4 Weaver enhances this with annotation-based code isolation and sequencing. P4Ansible continues in this direction by extending the Bison parser of the P4 compiler with new syntax (override, super, default), enabling the reuse of core language constructs such as parser states, controls, and headers. However, it lacks support for processing these features within the P4 intermediate representation (IR) and remains an incomplete effort.

Service Function Chain (SFC) frameworks [39–42] demonstrate the value of modularity in P4 by enabling dynamic and flexible composition of network functions. These approaches highlight the need for standardized, modular support within the P4 language to implement SFCs efficiently, ensuring programmability without incurring additional performance overhead. FlexNF proposes a flexible orchestration framework that enables modular service chaining and per-flow on-demand network function switching on P4-based programmable data planes [55]. Their work reinforces the critical role of modular design in scalable and

high-performance network architectures.

In addition, research on runtime programmability [43–47] highlights the architectural and language-level constraints that hinder dynamic, in-service updates, such as the addition or removal of protocols and network functions. Complementary to these studies, recent work on runtime verification automatically detects, localizes, and even corrects software bugs in P4 programs through reinforcement learning–guided fuzzing and dynamic analysis [56]. According to Vass et al. [57], optimally compiling P4 programs with resource constraints is inherently complex. However, they also state that even though no polynomial-time approximation can solve the problem, P4 compilation is approximable in linear time with a small constant bound, even for the most complex, nearly real-life models. Cao et al. [58] proposed a framework for compiling P4 programs into VHDL templates for FPGA-based network processors, introducing a match-action-oriented hardware architecture and evaluation library that improves throughput and reduces latency compared to existing P4FPGA approaches. Taken together, these studies underscore the critical role of modularity in the P4 language—not only for improving runtime adaptability and verification, but also for reducing the complexity of compilation and accelerating deployment across diverse hardware targets in programmable data planes.

The current solutions present trade-offs between performance, compatibility, and usability. The current tools either modify P4 syntax standards, slow down performance through additional processing steps, or require complex manual setup. Existing limitations demonstrate the ongoing requirement for solutions that enable code modularity with the P4 language and compiler infrastructure while maintaining compatibility with current development methods.

4.4 Goals and challenges of the Proposed P4O2

4.4.1 Goals of the Proposed P4O2

Enable Inheritance with Minimal Overhead

A core goal of P4O2 is to support inheritance-based modularity with minimal overhead without compromising the language's simplicity. This is achieved by extending P4 with lightweight and intuitive syntax elements—such as `extend`, `:`, and `.`—that allow developers to define and reuse components like parser states, control blocks, and structs in a modular fashion.

Support Fine-Grained Reusability and Extensibility

P4O2 aims to enable fine-grained control over modularity, allowing developers to incrementally compose and extend individual components such as parser states or match-action tables. This supports the development of reusable protocol libraries and application-specific extensions, reducing code duplication and easing integration across multiple network applications. Although our prior work, P4Muse, provided modularity mechanisms at the table level, it lacked support for reusing subcomponents within a table. P4O2 addresses this limitation by introducing inheritance mechanisms that support modular reuse at the granularity of individual parser states and tables, enabling more expressive and extensible P4 designs with minimal overhead.

Preserve Backward Compatibility and Toolchain Integration

To ensure seamless adoption, P4O2 is designed to be fully backward-compatible with existing $P4_{16}$ programs and the standard P4C compiler. All modular extensions are optional and non-intrusive, allowing developers to incrementally adopt P4O2 features without rewriting legacy code or modifying existing toolchains.

4.4.2 Challenges of the Proposed P4O2

Monolithic P4 Programs

The P4 language follows a monolithic design that lacks modular constructs commonly found in object-oriented languages. It also does not provide a built-in mechanism to maintain semantic links between different code components. For instance, there is no explicit connection between the definition of an IPv4 header and the parser state that extracts it.

Inheritance Inconsistency

Inheritance in P4O2 introduces challenges when the same component is reused through multiple inheritance paths, leading to the well-known *diamond problem* [59]. Additionally, when applying an inherited control or parser, the compiler may reference the original version instead of the extended one, resulting in outdated behavior being applied.

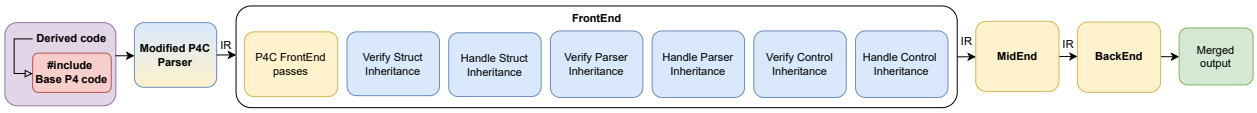


Figure 4.1 Overview of the P4O2 compiler showing its compiler stages.

4.5 Design of P4O2

P4O2 Syntax

P4O2 introduces a lightweight and backward-compatible syntax extension to the standard $P4_{16}$ language to support inheritance-based programming constructs. These extensions are inspired by classical object-oriented programming principles. They are designed to maintain the familiar look and feel of P4 while enabling composability and reusability through inheritance of protocol logic and application.

The core syntactic changes include:

- **Struct Inheritance:** Developers can define new header or metadata structures that extend existing ones using a ‘`:`’ notation in the struct of header or metadata, enabling incremental extension of protocol fields.
- **Parser Extension and Inheritance:** Parser states in derived parsers can extend base states using the `extend` keyword. This allows the reuse or augmentation of state transitions without duplicating the complete parser logic.
- **Table Extension and Inheritance:** Controls can inherit from one or more base controls. Tables within derived controls can be extended using `extend`, allowing the reuse or augmentation of base tables’ key, action, and size specifications.
- **Multiple Inheritance:** P4O2 supports multiple inheritance in parsers and controls, enabling the composition of functionalities across protocols and applications (e.g., combining tunneling and firewalling logic).

These syntactic constructs are intentionally minimal and non-intrusive. They allow existing P4 programs to remain valid while enabling new programs to adopt modular features incrementally.

P4O2 Compiler

Figure 4.1 illustrates the internal architecture of the P4O2 compiler and the key stages involved in processing modular P4 programs. The compilation process begins with the inclusion of derived code into a base P4 program using the standard `#include` directive. A modified version of the P4C front-end parser, extended with Yacc (Yet Another Compiler Compiler)/Bison grammar [60] and Lex (lexical analyzer) to handle the new keywords (`extend`, `:`, and `.`), then parses the resulting combined source.

P4O2 introduces several custom compiler passes in the *frontend* stage to support modular constructs:

- **Struct Inheritance:** The compiler first verifies that inherited fields across structs do not introduce naming conflicts or duplication. Once validated, it merges the fields from base structs into derived structs within the intermediate representation (IR).
- **Parser Inheritance:** Inherited parsers and their states are checked for correctness, ensuring that transitions are consistent and target states exist. The derived parser then incorporates base parser logic by combining transitions and adding missing states as needed.
- **Table Inheritance:** The compiler validates that inherited tables maintain compatible key and action definitions. It then integrates base table elements into the derived tables, allowing for extension or selective override of table properties and behavior.

After modular transformations are applied, the updated IR is passed to the unmodified P4C midend and backend stages, which handle target-specific lowering and code generation. The result is a fully merged P4 program that preserves the semantics of both the parent and child components and is compatible with standard backend toolchains (e.g., BMv2 [54]).

4.6 Methodology for Implementing the Proposed P4O2

We propose a methodology to enable inheritance in the P4 programming language by incorporating principles inspired by object-oriented programming [1]. This approach introduces inheritance constructs, enabling developers to define and reuse base components for key language elements such as structs, parsers, and control blocks. By extending the syntax and semantics of P4, we facilitate the effective use of inheritance syntax, allowing for improved

[1] P4O2 code and test cases discussed in the paper will be posted at https://github.com/PolyMTL-P4/P4O2 upon paper acceptance.

code organization, flexibility, and maintainability. This inheritance-based design empowers developers to extend base constructs according to specific requirements, enhancing the expressiveness and scalability of P4 programs.

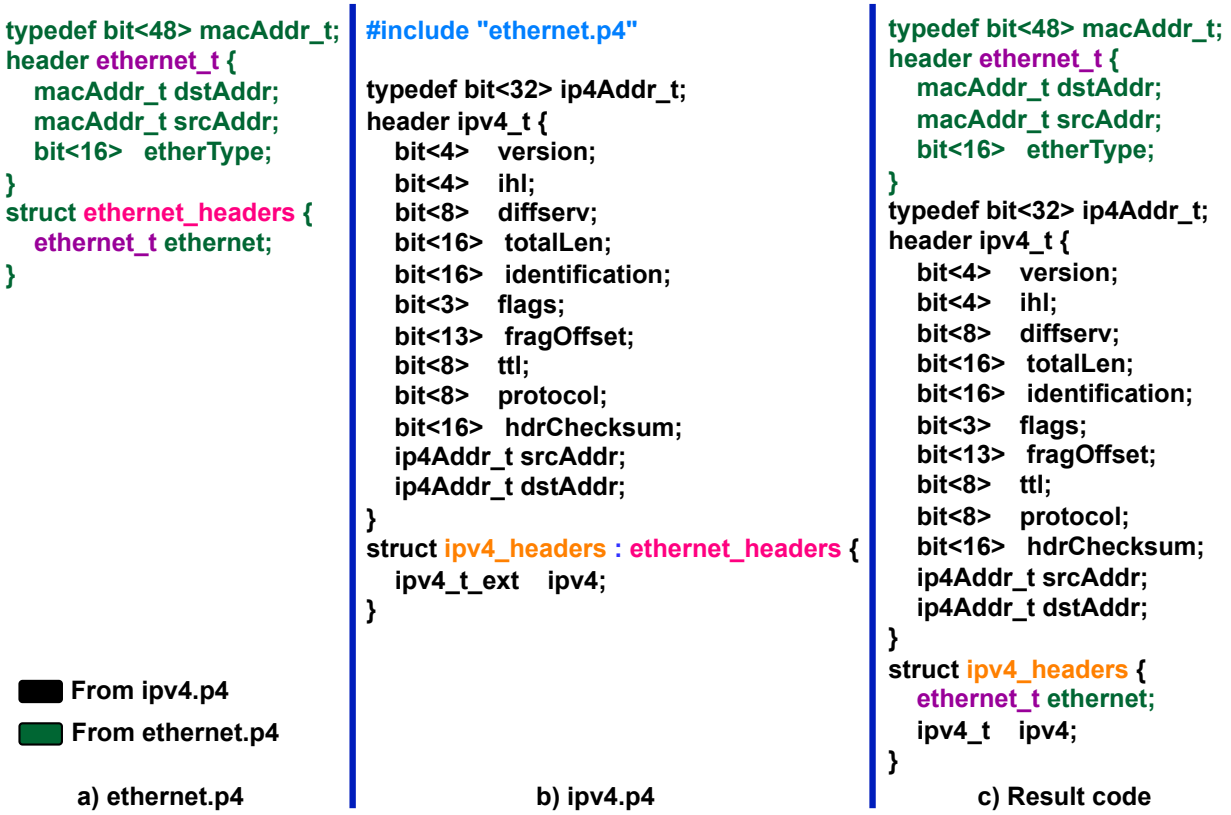


Figure 4.2 Struct and header blocks of Ethernet, IPv4, and the result P4 codes

4.6.1 Enabling Inheritance in the Struct Data Type

We enhanced the P4 language by modifying its YACC/Bison grammar and Lex rules to enable inheritance in the struct data type. These modifications introduce a new syntax that utilizes the ':' character, followed by the name of a base struct. For example, as demonstrated in Figure 4.2, the syntax enables a new struct such as `ipv4_headers` in `ipv4.p4` to extend an existing struct like `ethernet_headers` from `ethernet.p4` using the ':' character. As illustrated in the resulting code, the inheritance pass extends `ipv4_headers` by incorporating the fields of `ethernet_headers`, including the `ethernet_t ethernet`. The `include` directive is used in the standard way to bring in code from other files, allowing access to definitions such as the Ethernet header and macAddr without requiring their redefinition.

Multiple inheritance in the Struct Data Type

Multiple inheritance is supported in the struct data type when a derived struct inherits from two or more base structs. This is done by using the ':' character followed by the name of the first base struct, with additional base structs separated by commas. For example, in Figure 4.3, `struct advTunnel_firewall_headers` inherits from both `advTunnel_headers` and `tcp_headers` using the following syntax: 'struct advTunnel_firewall_headers : advTunnel_headers, tcp_headers {}'.

Both structs from `tcp.p4` and `advTunnel.p4` inherit from the struct `ipv4_headers` defined in `ipv4.p4`. This could lead to ambiguity in the 'advTunnel_firewall_headers' struct in `advTunnel_firewall.p4` when determining which instance of `ipv4_t ipv4` should be used. This scenario represents the classic diamond problem in multiple inheritance. To resolve this, we use the version of the conflicting field from the first parent that defines it, based on the left-to-right order in the inheritance list. Because all types are known at compile time, and we do not have dynamic typing, this does not add any overhead like virtual base in C++. Additionally, a warning is generated to inform the user of the conflict.

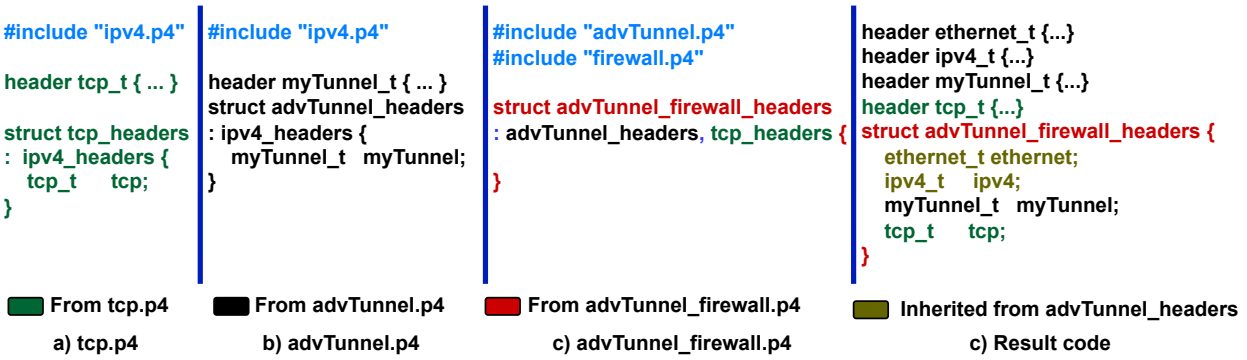


Figure 4.3 Struct and header blocks of TCP, Advanced Tunnel, Advanced Tunnel and Firewall, and the result P4 code

The Header Inheritance

We can reuse header definitions indirectly by using the ':' character to define a new struct, such as `ipv4_headers : ethernet_headers`, which extends a base struct and includes its headers. This allows us to reuse existing headers inside the new struct through inheritance. For example, as demonstrated in Figure 4.2, this syntax allows the `struct ipv4_headers` to reuse the existing `header ethernet_t`.

Figure 4.4 Parser blocks of Ethernet, IPv4, and the result P4 codes

4.6.2 Verification of Struct and Header Data Types

Through inheritance from one or more parent structs, the verification pass first checks whether the specified parent struct(s) exist. It then verifies the existence of any headers used within those parent structs. For example, it verifies the existence of `struct ethernet_headers` referenced in `ipv4_headers : ethernet_headers`, and also checks the existence of the `header ethernet_t` used within `struct ethernet_headers`.

4.6.3 Enabling Inheritance in the Parser

Figure 4.4 demonstrates how parser modularity is enabled by introducing inheritance syntax and selective state extension. We enable parser inheritance by allowing a parser, such as `IPv4Parser`, to extend a base parser using the ':' symbol, as shown in Figure 4.4b with `parser IPv4Parser : EthernetParser`. Additionally, we introduce the 'extend' keyword to allow a derived parser to extend specific states from the base parser. For example, the `parse_ethernet` state in `ipv4.p4` is extended to add a new transition to `parse_ipv4` when the `etherType` indicates IPv4. In Figure 4.4a, `EthernetParser` defines the original parsing logic, including the `parse_ethernet` state. In Figure 4.4b, `IPv4Parser` inherits this logic and selectively extends it. The inheritance pass for parser modularity handles both explicitly extended states and those inherited implicitly. It automatically includes states from the parent parser that are not explicitly extended by the child parser, such as the `start` state. For extended states, it also merges additional logic like header extraction (e.g., `packet.extract<ethernet_t>(hdr.ethernet)`) and transitions (e.g., `accept`). Figure 4.4c

shows the resulting code after compilation, where the IPv4 parser includes both Ethernet and IPv4 parsing behavior due to this inheritance mechanism.

Multiple inheritance in the parser

A parser exhibits multiple inheritance when it inherits from two or more parent parsers. This is expressed using the ':' symbol, followed by the name of the first parent parser, then additional parent parsers separated by commas. For example, in Figure 4.5, `advTunnel_firewall.p4` inherits from both `AdvTunnelParser` and `TCPParser`.

`AdvTunnelParser` reuses `parse_ipv4` from
`Ipv4Parser` and `TCPParser`, also `AdvTunnelParser` extends `parse_ipv4` from `Ipv4Parser`. Also, there is no ambiguity problem in this case because `AdvTunnelParser` does not extend `parse_ipv4` itself. However, if both parent parsers extend the same parser state (e.g., `parse_ipv4`), a conflict may arise. In such cases, we resolve the ambiguity by selecting the version from the first base parser (in left-to-right order) that defines the conflicting state. Also, if the programmer wants another way, he can explicitly say the state is from which base parser in 'child parser state 'extend' base parser '.' base parser state'. . A warning is also generated to inform the user of the ambiguity.

The reason we have 'child parser state 'extend' base parser '.' base parser state' in syntax is that the compiler can determine which parser state is extended from which parser state in the base parser.

Inheritance hierarchy for the parser

Figure 4.5 illustrates the inheritance hierarchy, with *Advanced Tunnel* and *Firewall* at the top. The `AdvTunnel_TCP` parser reuses both `AdvTunnel` and `TCP` parsers. In turn, `TCP` and `AdvTunnel` both rely on `IPv4` parser, which itself is built upon `Ethernet` parser. Additionally, the *Firewall* application directly reuses the `TCP` parser, while the *QoS* application builds directly upon the `IPv4` parser. So this illustrates the parser hierarchy among the *Advanced Tunnel*, *Firewall*, and *QoS* applications. The corresponding parser graph is depicted at the bottom of each hierarchy level, shown by a blue rectangle, where an oval represents each parser state. For clarity and space, standard states such as `reject` are omitted from the parser graphs. Each parser graph is shown in a way that reflects its inheritance syntax. For example, in the graph at the `IPv4` level, the `IPv4` parser reuses the `Ethernet` state—represented by a purple oval as extended one—since it is already defined at the Ethernet level. Additionally, the `IPv4` state includes a transition to the `accept` state as an inherited state, which is

represented by a red arrow.

In the graph on the right, we present the resulting parser graph generated by combining the inherited components from all levels. This is the same graph we would obtain if we manually wrote a modular parser by hand. The `Eth` and `accept` states, along with their transitions, are inherited directly from the parser graph defined at the Ethernet level. The `IPv4` parser, inherited at the IPv4 hierarchy level, introduces a new `IPv4` state with a transition to the existing `accept` state defined at the Ethernet level. At the Advanced Tunnel level, the parser extends the `Eth` state by adding a new transition to `parse_myTunnel`. Meanwhile, the TCP parser—built on top of the IPv4 parser—extends the `IPv4` state with a transition to the `TCP` state.

The result graph on the right correctly reflects this multi-level inheritance: each state and transition appears as defined or extended in its corresponding hierarchy level. Thus, the final graph is a composition of all the parser graphs on the left, validating the correctness of hierarchical inheritance and confirming that each reuse and extension has been properly applied.

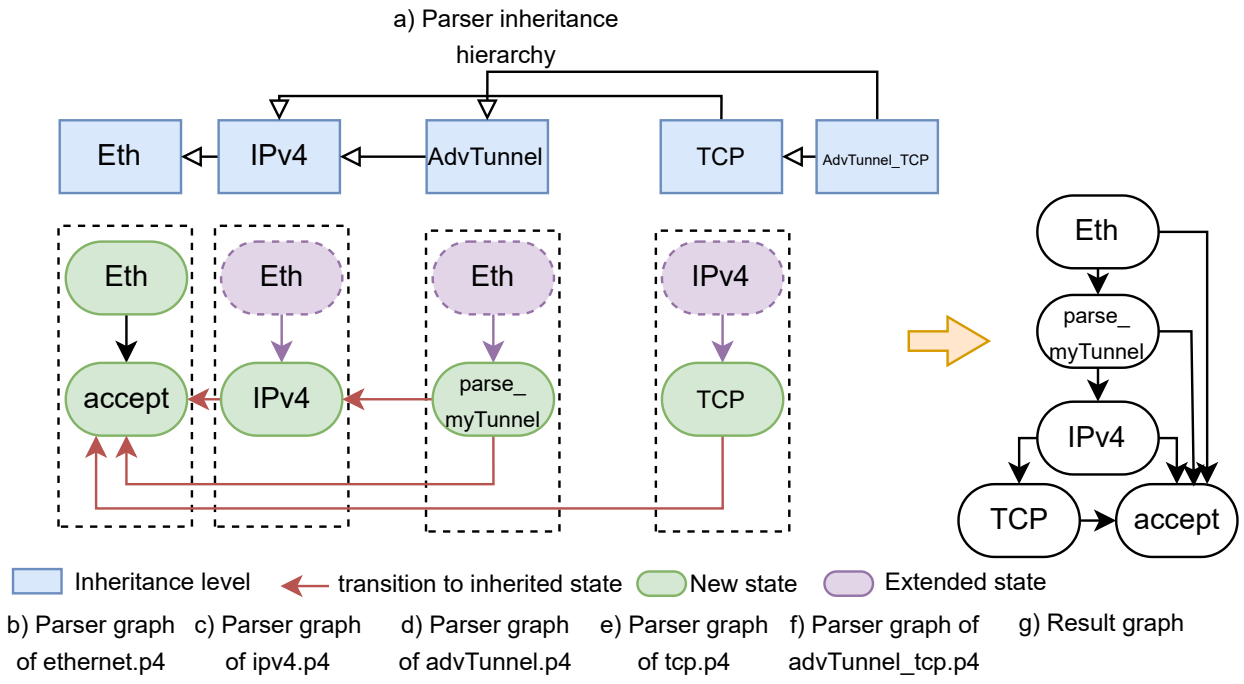


Figure 4.5 Hierarchical Inheritance and Parser Graph of Advanced Tunnel, Firewall, and QoS: Reuses the Parsers of Advanced Tunnel and TCP

4.6.4 Verification of Parser

The parser verification pass ensures that any referenced base parser is defined and accessible. It also verifies that all inherited parser states exist within the specified base parser. In Figure 4.4, the verification process first ensures the existence of the `EthernetParser` parser in `ethernet.p4`. It then checks that the `parse_ethernet` state is defined within the `EthernetParser` parser.

4.6.5 The Deparser Inheritance

The deparser can also benefit from inheritance in structs. When a struct inherits from one or more base structs, the deparser can emit the entire derived struct in a single `packet.emit(hdr)` call. According to the P4 specification (Section 16.1), emitting a struct automatically emits all of its member headers in declaration order, including those inherited from base structs. Therefore, it is unnecessary to explicitly emit headers from base structs. It should be noted that in the P4 language, headers not marked as valid are not emitted.

For example, in `ipv4.p4`, the `Ipv4Deparser` deparser emits `hdr`, which is of type `ipv4_headers`. Since `ipv4_headers` inherits from `ethernet_headers`, this single emit call automatically emits both the Ethernet and IPv4 headers. This makes the deparser code simpler and more modular without needing to duplicate logic from base deparsers.

This inheritance-aware behavior extends naturally to more complex header hierarchies. For instance, in `tcp.p4`, the `tcp_headers` struct inherits from `ipv4_headers`, which in turn inherits from `ethernet_headers`. Thus, the `TcpDeparser` emits all three headers (Ethernet, IPv4, and TCP) via a single `emit(hdr)` call. Similarly, in the `udp_tcp.p4` example, `udp_tcp_headers` inherits from both `udp_headers` and `tcp_headers`, which both reuse `ipv4_headers`, creating a potential diamond inheritance. However, due to the compiler's inheritance resolution, the common base headers like `ipv4` and `ethernet` are only emitted once, preserving correctness.

4.6.6 Reusing Control Logic via Apply Invocation

P4 supports modular and reusable control logic through a mechanism described in the P4 specification [7] as *Invoking Controls* (Section 13.4). This mechanism allows one control block to call the functionality of another, similar to invoking a subroutine. To achieve this, the base control must be instantiated, and its behavior is executed by calling its `apply` method.

In P4O2, we leverage this mechanism to reuse an existing `control` block with change. As

Figure 4.6 Control blocks of IPv4, Advanced Tunnel and the result P4 codes

illustrated in Figure 4.6, the control block `AdvTunnelIngress` (Advanced Tunnel Ingress) in `advTunnel.p4` reuses the logic of `Ipv4Ingress` defined in `ipv4.p4`. This reuse typically requires instantiating the control using 'IPv4Ingress() ipv4_ingress;' and invoking it with 'ipv4_ingress.apply(hdr, meta, standard_metadata);'. However, when inheritance is used, we allow to invoke it directly without instantiation using 'IPv4Ingress.apply(hdr, meta, standard_metadata);' for better clarity and simplicity.

- **Action**: In regular control invocation, actions defined in the base control (e.g., `ipv4_forward`) are also invoked in the new context, with their names prefixed by the instance (e.g., `ipv4_ingress_ipv4_forward`) to ensure modularity and avoid name collisions. However, when using inheritance, the actions are accessed without a prefix (e.g., `ipv4_forward`), offering better clarity and simplicity.

- **Table**: The tables defined in the base control (e.g., `ipv4_lpm` in `Ipv4Ingress`) are reused in the invoking control. These tables are uniquely identified in the generated code using the instance prefix (e.g., `ipv4_ingress_ipv4_lpm`) in regular control invocation. However, when using inheritance, the tables retain their original names (e.g., `ipv4_lpm`) for better readability and modular clarity.

- **Apply block**: In standard control invocation, the `apply` block of the instantiated control is executed. If the `apply` block contains conditional logic, that logic is preserved during the invocation. For example, the call to the instance `ipv4_ingress.apply(...)` results in the conditional check `hdr.ipv4.isValid()` and the corresponding table call 'ipv4_ingress_ipv4_lpm()'. However, when using inheritance, the `apply` block is reused directly without instantiation, preserving conditions and applying tables without prefixes, as shown in Figure 4.6c. This approach maintains original names and enhances clarity and modular structure.

4.6.7 Enabling Inheritance in the Table

In P4O2, table inheritance is enabled using the ':' syntax after the child control, followed by the name of the base control block. Additionally, similar to the parser inheritance syntax, the child table uses the `extend` keyword followed by the base control name, a dot, and the name of the base table.

For example, in Figure 4.6b, the table `ipv4_lpm` is extended by writing `extend IPv4Ingress .ipv4_lpm` after the new table's declaration. This allows the new table to inherit the structure and behavior of the base table.

When the base control is invoked by using inheritance (as explained in subsection 4.6.6), the inheritance pass accounts for the table and action identifiers introduced during invocation.

The inherited table retains its key fields, actions, and size, while the child table can define additional actions. In this example, actions from the `myTunnel_ingress` table are merged with the inherited actions from 'ipv4_lpm', supporting clean and modular extension of table behavior.

P4O2 can retain extended tables alongside their original ones and invoke them in different apply blocks as needed. This approach enables reusing a table instead of defining multiple ones when an extended table is referenced only once, but the table is invoked multiple times. Since P4 does not support loops, a table cannot be applied more than once in the pipeline. Therefore, the compiler keeps the extended table under a different name while preserving the original table, allowing the programmer to correctly invoke both when required.

Multiple Inheritance in the table

When a control block inherits from two or more parent control blocks, P4O2 supports multiple inheritance. This capability also extends to table inheritance. In other words, a child table can inherit from tables defined in any of its parent control blocks. This allows programmers

to selectively inherit tables from multiple sources, enabling flexible and modular composition of dataplane logic. Each inherited table contributes its keys, actions, and parameters to the extended table in the child control.

4.6.8 Verification of Table

A verification pass is applied when table inheritance ensures correctness and consistency. First, it verifies the existence of the base control referenced in the inheritance declaration (i.e., after the ':' in the child control). Then, it checks whether the base control specified in the `extend` statement of the child table matches one of the inherited base controls. Finally, it verifies that the referenced base table exists within the specified base control. For example, in Figure 4.6, the verification process first ensures that `IPv4Ingress` appears in the list of base controls inherited by the child control. It then confirms that `IPv4Ingress` is defined in `ipv4.p4`, and finally checks for the existence of the `ipv4_lpm` table within `IPv4Ingress`.

4.7 Parser Modularity Style: Composed vs. Independent Inheritance

Higher-layer protocols such as TCP are typically parsed after lower-layer protocols like IPv4 or IPv6 in layered network protocol stacks. Traditional monolithic P4 parsers reflect this structure by encoding all transitions within a single sequential parser. In contrast, P4O2 introduces inheritance and selective state extension, allowing each protocol to be defined independently and later linked together to form complete parser chains.

Figure 4.7 compares two approaches to implementing TCP parsing using P4O2: composed and independent inheritance. For instance, a `TCPParser` may inherit from `IPv4Parser` and/or `IPv6Parser`, and extend their `parse_ipv4` and/or `parse_ipv6` states to insert transitions into its own logic. Unlike standard P4, which embeds all protocol logic in a single parser, P4O2 enables modular reuse through two inheritance styles, each offering a different way to organize protocol parsing.

Composed Inheritance

In composed inheritance, the TCP parser extends the parsing states of specific predecessor protocols, such as `parse_ipv4` and `parse_ipv6`, within the same file. As shown in Figure 4.7a, this creates a version of `TCPParser` that directly depends on both `IPv4Parser` and `IPv6Parser`. While this approach maintains the expected parsing order, it limits reusability: incorporating TCP parsing into a new context (e.g., with a different protocol) requires redefining or duplicating parts of the TCP logic.

Independent Inheritance

In independent inheritance, each protocol is defined in isolation. As shown in Figure 4.7b, the `TCPParser` contains only the TCP-specific logic, and another parser, such as `SRv6_TCPParser`, inherits from both IP parsers and the TCP parser. It then extends the IP states to introduce transitions into the TCP parser. This approach avoids duplication and allows the same TCP logic to be reused across different protocol compositions without modification.

```
// This is tcp.p4
#include "ipv4.p4"
#include "ipv6.p4"
parser TCPParser(...) : IPv4Parser, IPv6Parser {
   state parse_ipv4 extend IPv4Parser.parse_ipv4 {
      transition select(hdr.ipv4.protocol) {
         8w6: parse_tcp;
      }
   }
   state parse_ipv6 extend IPv6Parser.parse_ipv6 {
      transition select(hdr.ipv6.next_hdr) {
         8w6: parse_tcp;
      }
   }
   state parse_tcp {
      packet.extract<tcp_t>(hdr.tcp);
      transition accept;
   }
}
```

a) TCPParser in composed Inheritance style

```
// This is tcp.p4
parser TCPParser(...) {
   state parse_tcp {
      packet.extract<tcp_t>(hdr.tcp);
      transition accept;
   }
}
// This is srv6_tcp.p4
#include "ipv4.p4"
#include "ipv6.p4"
#include "tcp.p4"
parser SRv6_TCPParser(...) :
IPv4Parser, IPv6Parser, TCPParser {
   state parse_ipv4 extend IPv4Parser.parse_ipv4 {
      transition select(hdr.ipv4.protocol) {
         8w6: parse_tcp;
      }
   }
   state parse_ipv6 extend IPv6Parser.parse_ipv6 {
      transition select(hdr.ipv6.next_hdr) {
         8w6: parse_tcp;
      }
   }
}
```

b) TCPParser in Independent Inheritance style

Figure 4.7 Comparison of Composed and Independent Inheritance Styles for TCP Parsing

4.8 Case studies of proposed P4O2

4.8.1 Firewall and Advanced Tunnel

This case study demonstrates how P4O2 enables modular composition of two distinct functionalities: a firewall (based on TCP/IPv4 filtering using Bloom filters) and an advanced tunneling mechanism (based on encapsulating IPv4 packets with a custom tunnel header). We show how the modular implementation using inheritance improves code reuse and maintainability compared to traditional monolithic approaches.

Modular Implementation using P4O2

In the modular P4O2 version, we split the functionality into reusable and composable modules:

- **ethernet.p4**, **ipv4.p4**, and **tcp.p4** define reusable packet headers, metadata, and basic parser/control blocks for Ethernet, IPv4, and TCP.
- **firewall.p4** reuses `Ipv4Ingress` by invoking control and reusing 'ipv4_lpm' table and 'drop' and 'ipv4_forward' actions.
- **advTunnel.p4** also extends `Ipv4Ingress`, adding tunneling behavior while reusing IPv4 parsing and forwarding.
- **firewall_advTunnel.p4** composes both 'FirewallIngress and 'AdvTunnelIngress' using multiple inheritance and invoking their controls. The resulting 'FirewallAdvTunnelIngress' control merges both functionalities into a single, coherent apply block.

This composition avoids code duplication and leverages base behaviors such as the `ipv4_lpm` table, which is extended in the tunneling application.

Monolithic Implementation Baseline

To evaluate the impact of modularity introduced by P4O2, we implemented the same functionalities using traditional monolithic P4 programs based on the P4 tutorial [2]. These monolithic versions follow the standard P4 design practice where all components—header definitions, parser states, controls, tables, and actions—are integrated into two files without reuse or composition.

Comparison of Functional Behavior

While the individual behaviors of the modular and monolithic implementations are functionally equivalent in isolation (e.g., both correctly perform IPv4 forwarding, tunneling, and TCP-based filtering), they differ significantly in composability. In the monolithic approach, the firewall and advanced tunnel functionalities must be implemented and executed as separate P4 programs, making it impossible to run them together on a single switch. In contrast, P4O2 allows these functionalities to be composed into a single P4 program (`firewall_advTunnel.p4`) by including and reusing logic from both `firewall.p4` and `advTunnel.p4`. This modular integration enables unified packet processing pipelines that are not achievable with traditional monolithic designs.

4.8.2 Firewall, Advanced Tunnel, and Quality of Service

This case study extends the previous composition of firewall and advanced tunneling functionalities by introducing a third functionality: Quality of Service (QoS). This demonstrates how P4O2 supports modular layering of orthogonal features while reusing previously developed modules without modification.

Modular Implementation using P4O2

Building on the modular design from the previous case study, we introduce a new module for QoS while reusing the existing firewall and tunneling logic:

- **ethernet.p4**, **ipv4.p4**, **tcp.p4**, **firewall.p4**, and **advTunnel.p4** are reused exactly as defined in the prior case.
- **qos.p4** introduces a new module for traffic classification and priority-based queuing based on packet headers.
- **firewall_advTunnel_qos.p4** composes all three modules using multiple inheritance and control invocation by using inheritance.

By reusing existing modules and only adding new logic in the `qos` and `firewall_advTunnel_qos` modules, this implementation avoids redundancy and enables modular integration across functionalities.

Monolithic Implementation Baseline

To provide a fair comparison, we implemented monolithic versions of the firewalling, tunneling, and quality of service functionalities.

4.8.3 SRv6 and NDP

This case study demonstrates how P4O2 supports modularly implementing complex applications, SRv6 (Segment Routing over IPv6) and NDP (Neighbor Discovery Protocol). These protocols represent two distinct aspects of IPv6 networking: SRv6 introduces flexible source routing for traffic engineering, while NDP handles critical functionality such as address resolution and router discovery.

Modular Implementation using P4O2

In our P4O2-based design, we decompose the SRv6 and NDP logic into composable modules that reuse core IPv6 parsing and metadata handling:

- **ethernet.p4** and **controllerIO.p4** define base Ethernet and control IO headers, metadata, and parsers, forming the common foundation for all modules.
- **ipv4.p4** and **ipv6.p4** extend these foundations by introducing IPv4 and IPv6 headers, parsers, and metadata. Both reuse the control-IO interface from `controllerIO.p4`.
- **udp.p4** and **tcp.p4** build on the IPv4 and IPv6 modules, adding transport-layer headers and parser states. They extend `IPv4Parser` and `IPv6Parser` to handle protocol identification and metadata population for L4 ports.
- **icmpv4.p4** and **icmpv6.p4** define ICMP-specific headers, metadata fields, and parser states. `ICMPv4Parser` extends `IPv4Parser`, while `ICMPv6Parser` extends `IPv6Parser`, extracting ICMP headers and keeping ICMP type codes.
- **ndp.p4** extends `IPv6Parser` to implement Neighbor Discovery Protocol parsing and control logic, leveraging the same metadata used by ICMPv6.
- **srv6.p4** extends `IPv6Parser` with Segment Routing over IPv6 (SRv6) header definitions and parser transitions, processing packets based on the IPv6 `next_hdr` field.
- **acl.p4** implements an Access Control List module as a control block that operates on the shared Ethernet and IP metadata, enabling packet filtering and CPU redirection.
- **srv6_ndp.p4** composes all different modules using multiple inheritance and control invocation by using inheritance.

All modules reuse a common local metadata structure. Instead of defining new types for each protocol, the NDP and SRv6 modules extend the existing `ipv4_local_metadata_t`, which includes fields such as IP protocol and L4 ports. This reuse is valid because both NDP and SRv6 operate over IPv6 and require similar metadata for control decisions.

The SRv6 and NDP modules can be composed into a single program by simply importing their respective parsers and controls. Parser transitions naturally flow from Ethernet to IPv6, then into SRv6 or NDP based on protocol fields. This layered parsing model, enabled by inheritance, results in clean and maintainable code where each protocol module focuses only on its unique behavior.

Monolithic Implementation Baseline

In the monolithic baseline, we implemented SRv6 and NDP functionality as standalone P4 programs based on the next-generation SDN tutorial [49]. Each monolithic version includes its own definition of Ethernet, IPv6, and other headers, duplicated parsing logic, and duplicated control blocks. The result is two isolated programs—one for SRv6 forwarding and one for NDP processing—that cannot be composed without manual code integration.

4.9 Results Obtained With P4O2

All analyses in this section are conducted on three representative modular programs: *Firewall + Tunnel*, *Firewall + Tunnel + QoS*, and *SRv6 + NDP*.

4.9.1 Reusability Analysis of P4O2

To assess P4O2's modularity benefits, we compare the line of code (LoC) in the header and struct, parser and deparser, and ingress and egress components.

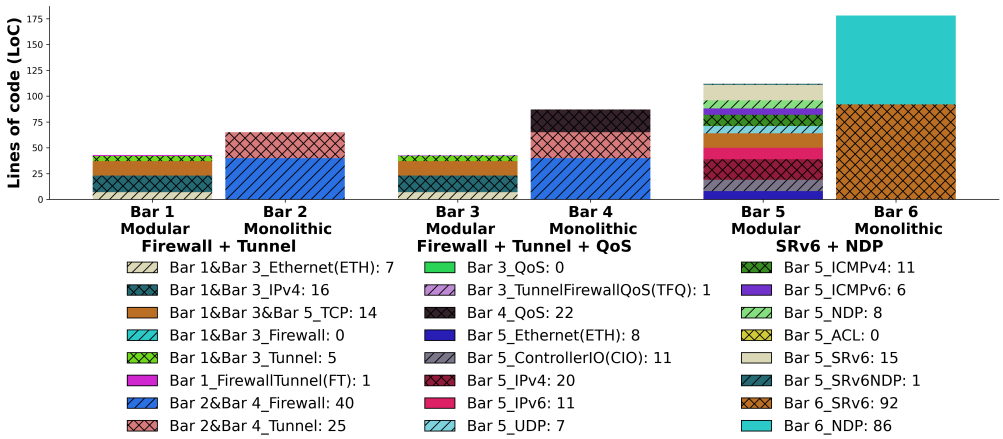
As shown in Figure 4.8a, line counts for header and struct components reveal clear modular advantages. The *Firewall + Tunnel* program exhibits 65 lines in the monolithic version, compared to just 43 lines in the modular version, reflecting significant reuse of definitions across modules.

Similarly, Figure 4.8b reveals that modular implementations sometimes involve parser and deparser code due to separation into multiple smaller components. For example, the *Firewall + Tunnel* program has 48 lines in the parser and deparser of the modular version, while its monolithic counterpart requires only 43 lines. However, this slight increase reflects improved separation of concerns and modular reuse.

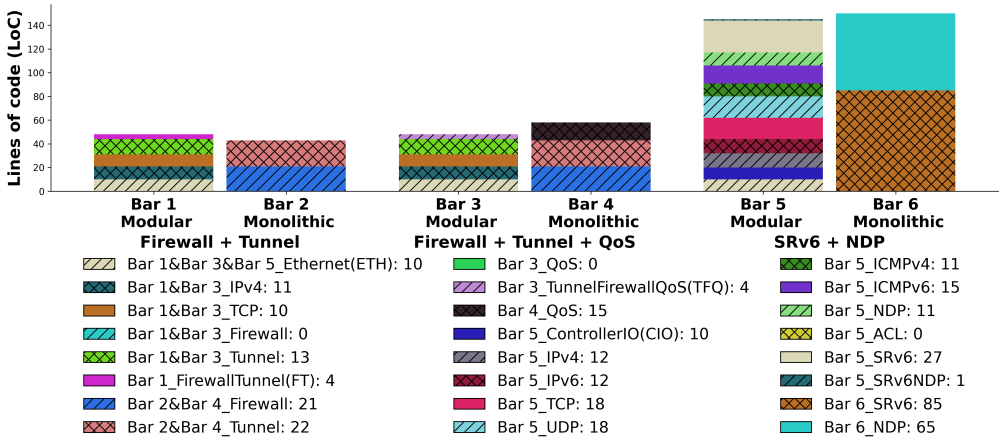
Finally, Figure 4.8c shows the code size in ingress and egress blocks. In the *Firewall + Tunnel + QoS* program, the monolithic implementation requires 163 lines. In contrast, the modular version reduces this to 139 lines, including just 5 lines in the reusable shared module `TFQ`, thanks to abstracted control logic and shared processing stages.

4.9.2 Modularity Analysis of P4O2

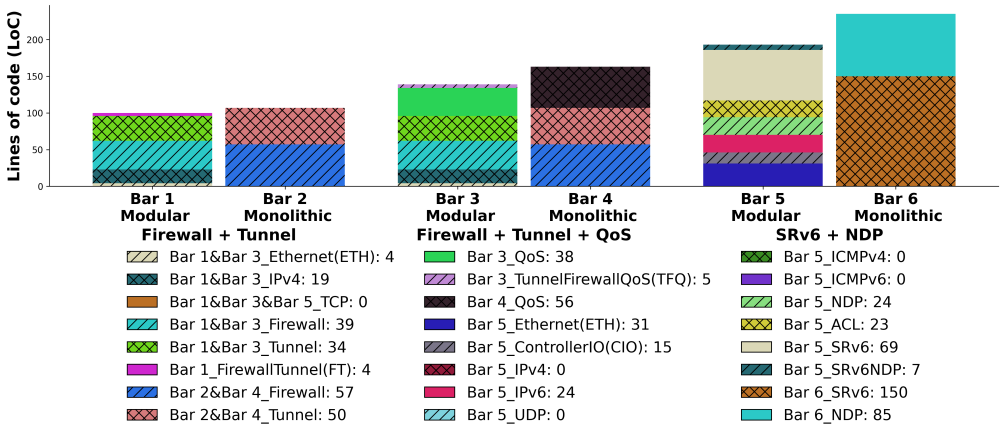
To further assess P4O2's modularity benefits, we analyze how functionalities are decomposed into reusable modules across the header and struct, parser and deparser, and ingress and egress components.



(a) Header and struct components



(b) Parser and deparser components



(c) Ingress and egress components

Figure 4.8 Comparison of lines of code (LoC) across components in modular and monolithic P4 programs.

For instance, in the *Firewall + Tunnel* program, the modular version separates protocol-specific headers into 6 dedicated components, compared to just 2 combined blocks in the monolithic version. Similarly, the *SRv6 + NDP* case includes 12 modular header components, whereas the monolithic version merges them into just 2 large definitions.

In Figure 4.8b, the modular approach cleanly separates parsing logic for each protocol. The *Firewall + Tunnel + QoS* program uses 6 distinct parser modules, each handling a specific protocol and a shared deparser component—totaling 6 modules. In contrast, the monolithic design groups all logic into 3 larger components, making reuse across applications more difficult. The modular structure facilitates independent reuse of each parser in other programs and promotes better maintainability.

For Figure 4.8c, modularity promotes separation of concerns and easier policy composition. For example, the *SRv6 + NDP* program in the modular version (Bar 5) uses 7 distinct modules in the ingress and egress blocks—representing clearly separated functionalities for Ethernet handling, IPv6 forwarding, SRv6 behavior, neighbor discovery, ACLs, controller interaction, and combined SRv6+NDP logic. In contrast, the monolithic version (Bar 6) compresses this logic into 2 larger files, reducing clarity and flexibility. Across all programs, the modular ingress and egress design leads to better scalability and easier integration of new features.

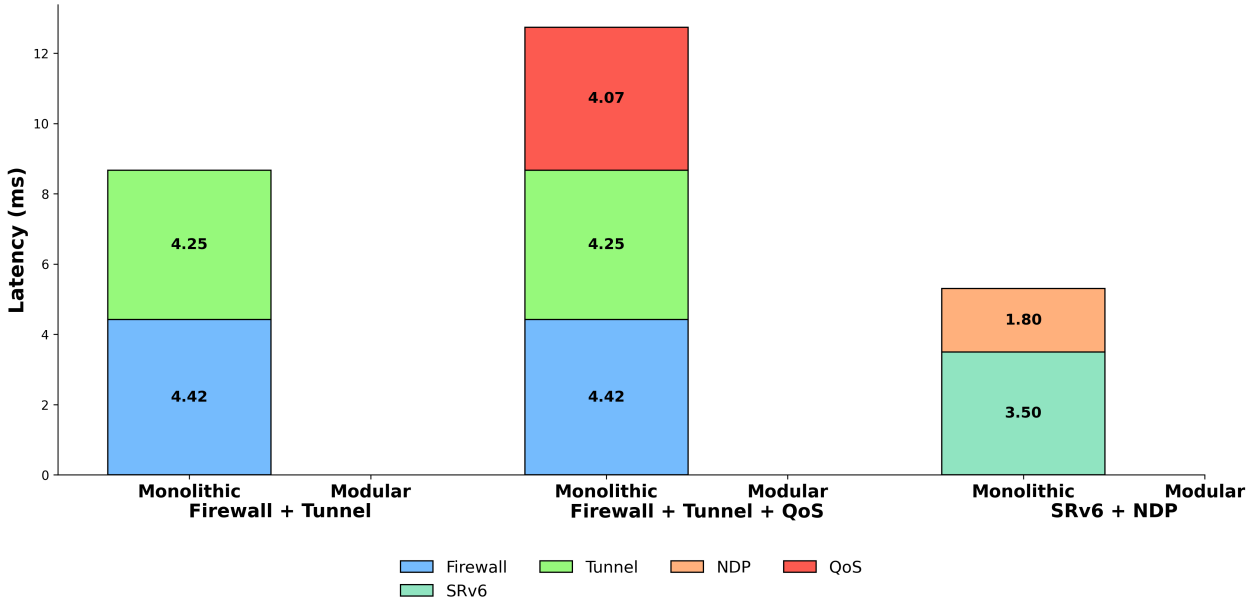
4.9.3 Performance Analysis of P4O2

To evaluate the performance of P4O2, we analyzed the Latency and Reciprocal of throughput of our three case studies. Figure 4.9a presents the end-to-end latency measurements for each program, while Figure 4.9b shows the reciprocal of throughput.

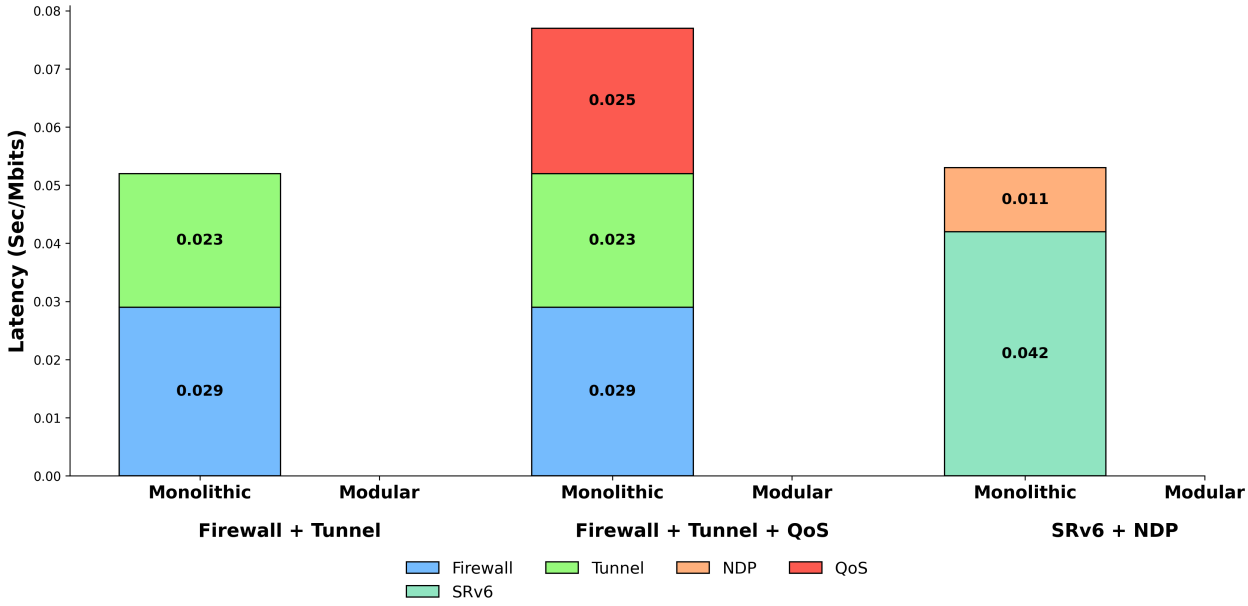
These results demonstrate that despite the modular decomposition of functionalities across multiple reusable components, P4O2 programs maintain competitive performance. As seen in Figure 4.9a, the Latency remains within acceptable bounds, even for more complex compositions such as *Firewall + Tunnel + QoS*, which includes deeper processing pipelines. This indicates that modularization in P4O2 does not introduce processing overhead.

Similarly, the Reciprocal of throughput results in Figure 4.9b confirm that modular programs in P4O2 achieve high forwarding performance. The *SRv6 + NDP* example, which represents a feature-rich program with extensive header processing and control logic, shows that P4O2 can support advanced use cases without sacrificing efficiency.

Overall, the performance analysis confirms that P4O2 enables scalable and maintainable data plane development without compromising runtime performance.



(a) Latency of modular and monolithic programs



(b) Reciprocal of throughput for modular and monolithic programs

Figure 4.9 Comparison of latency and reciprocal of throughput in modular and monolithic P4 programs.

4.9.4 Comparison with Related Work

Table 4.1 compares P4O2 to six related works—µP4, Lyra, P4Weaver, P4Ansible, daPIPE, and P4Muse—based on key features such as inheritance, backward compatibility, and integration with the P4 intermediate representation (IR). Unlike P4O2, these approaches lack at least one of these essential features. In particular, they often do not directly leverage the P4 IR or compiler infrastructure and typically do not maintain backward compatibility with existing P4 programs. In particular, P4Ansible does not provide complete inheritance support for the main P4 constructs in its open-source project, and it is implemented using Bison rather than the P4 IR.

4.10 Conclusions

P4O2 introduced in the present paper is a lightweight, backward-compatible extension to the P4 language, enabling modular and reusable dataplane programming through inheritance and selective extension. By allowing parsers, structs, and tables to be defined independently and reused across programs, P4O2 significantly improves code organization, maintainability, and scalability. Through case studies such as tunneling, firewall, QoS, SRv6, and NDP, we demonstrated how P4O2 simplifies protocols and applications composition while reducing code duplication.

The results further confirm that P4O2 delivers tangible reusability, modularity, and performance benefits. Our reusability analysis shows consistent reductions in the number of lines of code for headers, structs, and control logic, highlighting the efficiency of defining modules once and reusing them across multiple programs. The modularity analysis demonstrates that decomposing parsers, ingress, and egress pipelines into smaller components improves separation of concerns and facilitates scalable extensions, such as combining SRv6 with NDP. Importantly, our performance evaluation shows that these advantages come at no runtime cost, even for complex, feature-rich use cases. These findings demonstrate that P4O2 enables

Table 4.1 **Features Comparison in relation with P4 Modularity Solutions**

P4 Solutions	Inheritance	Backward Compatibility	Integration with P4 IR
µP4 [14]	No	No	No
Lyra [15]	No	N/A	No
P4Weaver [16]	No	Yes	No
P4Ansible [17]	Yes*	Yes	No**
daPIPE [13]	No	Yes	No
P4Muse [52]	No	Yes	Yes
P4O2	Yes	Yes	Yes

developers to write cleaner, more maintainable programs while ensuring that the data plane remains efficient and production-ready.

CHAPTER 5 GENERAL DISCUSSION AND CONTRIBUTIONS

5.1 Addressing Research Objectives

In today's programmable data plane devices, flexible, high-performance packet processing capacity has advanced significantly, with most pipeline components now fully programmable. However, despite this progress, the absence of principled modularity within the P4 language and its compiler remains a key obstacle to developing scalable, reusable, and maintainable data plane programs. Achieving modularity in P4 is challenging due to its monolithic architecture, lack of explicit modular constructs, and lack of standardized mechanisms for connecting and reusing protocol components.

To address these challenges, this thesis undertook a comprehensive review of existing modularity solutions in the P4 ecosystem and related domains, examining both academic and industrial approaches. This survey clearly identified the requirements for modular programming in data plane environments, including the need for composable protocol libraries, backward compatibility, and a minimal learning curve. The analysis also revealed critical limitations and trade-offs in prior solutions—such as reliance on external preprocessors, lack of compiler-native integration, or incompatibility with legacy code—which have hindered their adoption in real-world programmable networks.

Against this backdrop, this thesis introduces two complementary modularity frameworks for P4: (1) a compiler-managed approach (P4Muse) that achieves modular composition without any changes to the P4 language syntax, and (2) a lightweight, object-oriented-inspired language extension (P4O2) that enables structured inheritance and explicit module reuse via minimal, backward-compatible syntax additions. These contributions directly address the objectives outlined in the thesis introduction, and are realized through a combination of new compiler passes, extended syntax, and empirical validation on representative network applications. This chapter discusses in detail the main findings and contributions presented in chapter 3 and chapter 4, which are aligned with the specific objectives outlined in chapter 1, Introduction.

Chapter 3 introduced P4Muse, a novel compiler extension designed to address the lack of modularity in P4 programming. P4Muse implements an object-based framework for modular composition and code reuse across all core P4 program sections, including parsers, headers, structs, controls, and deparsers. We presented new compiler passes and algorithms that enable automatic merging of P4 modules without requiring changes to the P4 syntax or ex-

isting codebases. The key primitives introduced, such as standardized name mapping, state and header comparison, and functional verification, facilitate seamless integration of base and extension codes, overcoming challenges like naming inconsistencies and tightly coupled logic. Through comprehensive evaluation across several classes of use cases, including code reusability, data-plane pipeline composability, and vendor-customer collaboration, P4Muse demonstrated its effectiveness in enabling modular development, reducing code redundancy, and maintaining backward compatibility. Our results show that P4Muse not only streamlines incremental development and collaborative composition but also supports complex program integration that previous approaches struggled to achieve within the constraints of the standard P4 language.

Chapter 4 introduced P4O2, an object-oriented-inspired extension to the P4 language designed to enable true modularity and code reuse in data plane programming. We presented the P4O2 intuitive syntax, including inheritance for structs, parsers, and tables, and described their integration into the P4C compiler through specialized compiler passes for modular transformations and verification. P4O2's lightweight syntax and backward compatibility allow developers to incrementally adopt modular features without rewriting legacy code. Through a series of case studies—covering advanced tunneling, firewalling, QoS, and SRv6—we demonstrated how P4O2 enables flexible composition of protocol modules, reducing code duplication and improving maintainability compared to monolithic P4 programs. Our experimental evaluation on BMv2 showed that these modular programs maintain competitive performance and enable scalable, maintainable development for complex network functions, surpassing the capabilities of prior modularity solutions for P4.

5.2 Comparison with Existing Literature

With respect to existing approaches to achieve modularity in P4 programming, prior solutions exhibit a range of strengths and weaknesses, as detailed in chapter 2 and summarized in Table2.1. Most importantly, these trade-offs center around aspects such as ease of integration with standard P4 compilers, backward compatibility, and the granularity of modular composition. For example, µP4 and Lyra provide new abstractions for data plane modularity, but require either new syntax or separate intermediate languages, limiting their direct applicability and compatibility with legacy P4 code. Approaches such as P4Weaver and daPIPE support incremental development and vendor-customer separation but depend on external annotation or partial compilers rather than integration at the language or IR level. Virtualization-based solutions enable multi-program composition but at the cost of added complexity and, often, reduced performance. In response to these limitations, this thesis in-

troduces P4Muse and P4O2, two complementary solutions designed to combine the benefits of automatic, fine-grained modularity and backward-compatible language extensibility, while avoiding the key drawbacks of previous methods. The following section presents a comparative analysis, evaluating P4Muse and P4O2 against existing literature regarding compiler integration, ease of use, modular expressiveness, and support for maintainable, scalable P4 development.

5.3 Implications and Significance

The significance of this thesis is underscored by two main contributions: the introduction of principled modularity into P4 programming via compiler-managed composition (P4Muse) and the definition of a minimal, object-oriented-inspired extension to the P4 language (P4O2). First, we presented P4Muse, a modularity framework that brings automated, fine-grained code composition and reuse directly into the P4 compilation process. This advancement fulfills a critical gap in the P4 ecosystem, supporting maintainable and scalable data plane program development—capabilities integration wiht P4 IR and backward compatibility that were previously unattainable within the confines of the standard P4 language.

Throughout the development of P4Muse and P4O2, we rigorously analyzed the unique constraints and requirements imposed by programmable network data planes. Key considerations included the need for backward compatibility, seamless integration with legacy code, and minimal overhead in both code management and runtime performance. Our solutions had to accommodate the evolving needs of P4 developers, enabling them to compose, extend, and maintain complex protocol logic without introducing breaking changes or excessive complexity. Additionally, we recognized the necessity for precise mechanisms to resolve conflicts during code merging and inheritance, especially in large-scale and collaborative settings.

As a result, this work does not merely propose new modularity constructs for P4 but also provides a set of design principles and technical solutions that can guide the evolution of modularity support in future versions of the P4 language and compiler. We believe these insights should inform ongoing and future initiatives to enhance code organization, reuse, and extensibility in data plane programming languages.

Moreover, the expressive power of the proposed modularity frameworks was demonstrated through a series of representative case studies, ranging from protocol library reuse and multi-function pipeline composition to complex scenarios such as vendor-customer code integration and advanced service chaining. These studies revealed the distinct requirements of modular composition and helped identify the essential features needed for a truly modular P4 pro-

gramming environment. By validating P4Muse and P4O2 on these diverse applications, we have shown that our approaches enable the development of scalable, maintainable, and robust data plane programs without compromising the performance or semantics of the underlying network functions.

To our knowledge, this is the first comprehensive effort to bring native, fine-grained modularity to both the P4 language and its reference compiler. By building this foundation, we provide the P4 community and broader network programming field with a set of practical tools, case studies, and lessons learned, which can accelerate further innovation and adoption of modular practices in programmable data planes. We anticipate these advances will empower researchers and practitioners alike to design, extend, and maintain increasingly complex data plane applications with greater confidence, flexibility, and efficiency.

The evaluation metrics adopted in this thesis were chosen to jointly capture the software engineering and performance implications of introducing modularity into P4 programs. Lines of code (LOC) were used to quantify improvements in maintainability and code reuse, as reductions in LOC directly reflect decreased code duplication and improved structural organization enabled by modular composition. In addition, the number of modules was used as a complementary metric to characterize the granularity and expressiveness of modular decomposition, providing insight into how complex data plane functionality can be structured into reusable and composable components. To validate that these modularity benefits do not come at the expense of runtime efficiency, latency, and reciprocal throughput were employed as performance metrics and evaluated for both modular and individual implementations. Together, these metrics enable a balanced evaluation that demonstrates how P4Muse and P4O2 improve programmability, scalability, and maintainability while preserving the performance characteristics of the underlying data plane.

5.4 Limitations

5.4.1 Inherited Limitations

Several limitations in our proposed modularity frameworks, P4Muse and P4O2, are inherited from the current P4 language design and broader paradigms in programmable data plane development.

Lack of Native Modularity in P4: Since P4 was originally designed without modular constructs, both P4Muse and P4O2 must operate within the constraints of the monolithic language and IR structure. As a result, certain aspects of modularity—such as explicit module boundaries, well-defined interfaces between protocol elements, and automatic linkage

of related components—remain constrained by the language's original architecture.

Dependence on Compiler Extensions: While our approaches are realized as extensions to the P4C compiler, full modularity support requires these extensions to be integrated and maintained alongside ongoing developments in the main P4C codebase. This reliance on external or patched compiler versions can present adoption barriers and may delay widespread community support until official upstreaming occurs.

5.4.2 Specific Limitations of P4Muse and P4O2

Static Modularity Only: Both P4Muse and P4O2 currently support modularity at compile time; they do not address dynamic or runtime modularity, such as hot-swapping modules or live program updates found in some emerging data plane architectures. Future network applications may demand more flexible mechanisms for updating or extending data plane functionality at runtime.

Evaluation Limited to Reference Architectures: Our empirical validation has been performed primarily on the BMv2 software switch using the V1Model architecture. While this is representative of many research and early deployment settings, additional work will be needed to adapt and evaluate these modularity solutions for alternative and emerging P4 architectures, such as PSA or TNA, and for deployment on high-performance hardware targets.

Tooling and Ecosystem Integration: While the core functionality is implemented in the compiler, broader adoption will benefit from enhanced developer tooling—such as IDE support, modularity-aware debugging, and visualization—and closer integration with testing and verification pipelines.

CHAPTER 6 CONCLUSION

In this thesis, we introduced two complementary modularity frameworks for P4 programming: P4Muse, a compiler-managed modularity solution requiring no changes to the P4 syntax, and P4O2, a minimal object-oriented-inspired extension enabling explicit inheritance and code reuse within the P4 language. The primary objective of this doctoral research was to bridge the gap in current data plane development by providing scalable, maintainable, and reusable programming constructs—thereby aligning P4 with modern software engineering practices and the principles of Software-Defined Networking (SDN).

The research began with a comprehensive survey of the state of modularity in P4 and related domains, identifying critical shortcomings in existing solutions such as reliance on external preprocessors, lack of compiler-native integration, and incompatibility with legacy code. Building on these insights, we designed and implemented P4Muse, introducing new compiler passes for the automatic merging of protocol modules, pipeline components, and vendor-customer codebases without requiring any changes to the P4 language or disruption of existing workflows.

Recognizing the potential for even greater modularity and code reuse, the research progressed to developing P4O2, which adds lightweight inheritance constructs for structs, parsers, and controls. This extension was carefully integrated into the P4C compiler, ensuring full backward compatibility while empowering developers to compose complex, layered network functions in a modular and incremental fashion.

Through comprehensive empirical evaluation using a range of representative case studies, we demonstrated that P4Muse and P4O2 significantly enhance the maintainability, extensibility, and scalability of P4 programs—enabling use cases and collaborative workflows previously unattainable in monolithic P4 codebases. These contributions pave the way for a new generation of modular, robust, and production-ready P4 applications.

6.1 Summary of Works

This thesis is organized to present the original research contributions to modularity in P4 programming across each chapter. Chapter 3 introduced P4Muse, a novel compiler-managed modularity framework that enables automatic merging and composition of P4 program modules without requiring changes to the P4 language syntax. P4Muse was designed to overcome the inherent monolithic structure of P4 programs, supporting code reuse, pipeline

composition, and vendor-customer collaboration by leveraging new compiler passes for protocol, parser, control, and table integration. The effectiveness of P4Muse was demonstrated through several representative case studies, including modular protocol library reuse and multi-function pipeline composition, where it was shown to reduce code duplication and improve maintainability while preserving the performance and semantics of the composed programs.

Building upon this foundation, Chapter 4 introduced P4O2, an object-oriented-inspired extension to the P4 language. P4O2 introduces lightweight syntax for inheritance and selective extension of structs, parsers, and controls, enabling developers to construct layered protocol stacks and reusable network function modules. The integration of these features into the P4C compiler was described in detail, along with handling multiple inheritance and conflict resolution. Through a diverse set of case studies—including advanced tunneling, firewalling, quality of service (QoS), and SRv6—P4O2 demonstrated the ability to support flexible, scalable, and maintainable data plane application development, far surpassing the compositional capabilities of conventional monolithic P4 code.

To facilitate adoption and ensure practical impact, both P4Muse and P4O2 were evaluated using the BMv2 software switch and the V1Model architecture, with results showing that modular and inheritance-based designs can be realized with no measurable penalty in performance or resource usage. Together, these contributions provide the P4 community with a comprehensive framework for modular programming, enabling efficient code organization, collaboration, and future extensibility in programmable data plane environments. This lays a strong foundation for ongoing research into advanced modularity features and their integration into emerging network programming languages and architectures.

6.2 Summary of the Contributions

This thesis has made two key contributions to the advancement of modularity in P4 programming:

- It introduced **P4Muse**, a compiler-managed modularity framework for P4, enabling automatic merging and composition of protocol modules, controls, and pipeline components without requiring changes to the P4 language or manual code intervention.
- It proposed and realized **P4O2**, a lightweight, object-oriented-inspired extension to the P4 language, providing inheritance and selective extension constructs for structs, parsers, and controls, thereby facilitating modular design, code reuse, and scalable protocol stack development.

6.2.1 Publications

During the course of this doctoral research, several publications have resulted from the work presented in this thesis:

1. Rahmati, M., Boyer, F.R., Pontikakis, B., David, J.P., Savaria, Y., "Modular Code Parser for the P4 Language," *Proceedings of the P4 Workshop 2023*

2. Rahmati, M., Boyer, F.R., Pontikakis, B., David, J.P., Savaria, Y., "P4Muse: Enabling Modular P4 Programming via Compiler-Managed Code Merging Without Syntax Modifications," *IEEE Access*, Vol. 13, 2025, pp. 124138–124157.

3. Rahmati, M., Boyer, F.R., Pontikakis, B., David, J.P., Savaria, Y., "P4O2: Enabling Object-Oriented-Inspired Modularity in P4," [Submitted to IEEE Access], February 2025.

6.3 Future Research

In this section, we outline potential avenues for future research and development that can further advance modularity in P4 programming, building upon the frameworks introduced in this thesis.

Dynamic and Runtime Modularity: While P4Muse and P4O2 currently enable modularity at compile time, future research should explore mechanisms to support dynamic or runtime modularity in P4. This includes hot-swapping modules, live updates to protocol stacks, and the ability to dynamically compose or extend data plane programs without requiring recompilation or redeployment. Such capabilities would significantly enhance the adaptability and lifecycle management of programmable networks.

Standardization and Ecosystem Integration: For broad adoption, it is essential to pursue the standardization of modularity constructs and their integration into the mainline P4C compiler and associated toolchains. Efforts should focus on developing language specifications, APIs, and best practices for modular programming in P4, ensuring that the benefits of modularity are accessible to the entire P4 community and compatible with a wide variety of targets and platforms.

Enhanced Tooling and Developer Support: The development of user-friendly tools, such as modularity-aware integrated development environments (IDEs), visual module graph editors, automated code analysis and verification tools, and advanced debugging utilities, will

be crucial to supporting modular P4 development at scale. Such tools can simplify module management, conflict resolution, and testing in complex projects.

Extending to New Architectures and Hardware Targets: Future work should investigate the application and adaptation of P4Muse and P4O2 to emerging P4 architectures such as PSA and TNA, as well as their deployment on high-performance ASICs and FPGA platforms. Understanding these environments' practical constraints and opportunities will help realize modular P4 programming in production-scale deployments.

Advanced Conflict Resolution and Policy-Driven Composition: As modular programming becomes more prevalent, sophisticated strategies for resolving naming, inheritance, and behavioral conflicts between modules will become necessary. Future research should focus on user-directed conflict resolution policies, semantic merging techniques, and support for more complex inheritance hierarchies.

Formal Verification and Correctness Guarantees: Ensuring semantic correctness and safety in modular P4 programs is essential, especially as complexity grows. Further work can focus on integrating formal verification tools to check module composition, inheritance, and behavioral preservation - providing stronger correctness guarantees and reliability for mission-critical deployments.

Integration Across Control Planes: Seamless integration of modularity constructs not only within the data plane, but also across the control plane remains a vital direction. This would enable more flexible service chaining, incremental network upgrades, and improved automation in programmable networks.

In summary, the future work described here aims to facilitate the broad adoption, scalability, and impact of modular P4 programming. By advancing in these directions, we anticipate further improvements in code maintainability, flexibility, and innovation across the programmable networking landscape.

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," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023. doi: 10.1016/j.jnca.2022.103561

[2] "P4 tutorial," https://github.com/knetsolutions/p4-tutorials, accessed: 2023-03-08.

[3] A. Liatifis, P. Sarigiannidis, V. Argyriou, and T. Lagkas, "Advancing SDN from OpenFlow to P4: A survey," *ACM Comput. Surv.*, vol. 55, no. 9, jan 2023. doi: 10.1145/3556973

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69–74, Mar. 2008. doi: 10.1145/1355734.1355746

[5] Open Networking Foundation, "OpenFlow switch specification," *Open Networking Foundation Specification*, vol. 1, no. 5.1.0, pp. 1–320, Mar. 2015. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[6] 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," vol. 44, p. 87–95, 2014. doi: 10.1145/2656877.2656890

[7] P4.org Architecture Working Group, "P4_16 language specification," *P4.org Specification*, vol. 1, no. 1.2.5, pp. 1–244, Nov. 2023. [Online]. Available: https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.html

[8] P4 Language Consortium, "p4c-behavioral," https://github.com/p4lang/p4c-behavioral, 2023, gitHub repository.

[9] The P4 Language Consortium, "behavioral-model (BMv2): The P4 reference switch," https://github.com/p4lang/behavioral-model, 2023, gitHub repository.

[10] "The p4 language consortium. 2019. v1model.p4 - architecture for simple_switch," https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4, accessed: 2023-03-08.

[11] “"the p4 language consortium. p4-16 portable switch architecture (psa).",” https://p4.org/p4-spec/docs/PSA-v1.1.0.html, accessed: 2023-03-08.

[12] The P4 Language Consortium, “P416 reference compiler,” https://github.com/p4lang/p4c, 2019, accessed: 2025-10-01.

[13] M. Baldi, “daPIPE a data plane incremental programming environment,” in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–6. doi: 10.1109/ANCS.2019.8901893

[14] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, “Composing dataplane programs with µP4,” in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 329–343. doi: 10.1145/3387514.3405872

[15] 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 *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 435–450. doi: 10.1145/3387514.3405879

[16] A. Fattaholmanan, M. Baldi, A. Carzaniga, and R. Soulé, “P4 Weaver: Supporting modular and incremental programming in P4,” in *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, ser. SOSR ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 54–65. doi: 10.1145/3482898.3483353

[17] “"p4-ansible.",” https://mnkcg.com/products/p4-ansible/, accessed: 2023-03-08.

[18] A. Nojavan, B. Pontikakis, F.-R. Boyer, and Y. Savaria, “5g fronthaul in modular p4: ecpri protocol processing and precise bmv2 timestamps for ptp-1588,” *IEEE Access*, vol. 13, pp. 22 710–22 727, 2025. doi: 10.1109/ACCESS.2025.3536362

[19] D. Loehr and D. Walker, “Safe, modular packet pipeline programming,” vol. 6, no. POPL, Jan. 2022. doi: 10.1145/3498699. [Online]. Available: https://doi.org/10.1145/3498699

[20] T. Alberdingk Thijm, R. Beckett, A. Gupta, and D. Walker, “Modular control plane verification via temporal invariants,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. doi: 10.1145/3591222. [Online]. Available: https://doi.org/10.1145/3591222

[21] GitHub, “NPL-spec,” San Francisco, CA, USA, 2021, accessed: 2021-02-04. [Online]. Available: https://github.com/nplang/NPL-Spec

[22] The P4 Language Consortium, “switch.p4 program,” Stanford, CA, USA, 2013. [Online]. Available: https://github.com/p4lang/switch/

[23] “"the srv6 network programming",” https://github.com/netgroup/p4-srv6/blob/master/p4src/include/parser.p4, accessed: 2023-03-08.

[24] Y. Zhou and J. Bi, “ClickP4: Towards modular programming of P4,” in *Proceedings of the SIGCOMM Posters and Demos*, ser. SIGCOMM Posters and Demos ’17. New York, NY, USA: Association for Computing Machinery, 08 2017, pp. 100–102. doi: 10.1145/3123878.3132000

[25] 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 *2021 17th International Conference on Network and Service Management (CNSM)*, 2021, pp. 375–379. doi: 10.23919/CNSM52442.2021.9615508

[26] 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 *Proceedings of the 3rd P4 Workshop in Europe*, ser. EuroP4’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 59–61. doi: 10.1145/3426744.3431330

[27] P. D. Bol, R. Lunardi, B. de França, and W. Cordeiro, “Modular switch deployment in programmable forwarding planes with switch (de)composer,” p. 30–32, 2021. doi: 10.1145/3472716.3472856

[28] S. Han, S. Jang, H. Choi, H. Lee, and S. Pack, “Virtualization in programmable data plane: A survey and open challenges,” *IEEE Open Journal of the Communications Society*, vol. 1, pp. 527–534, Jun. 2020. doi: 10.1109/OJCOMS.2020.2990182

[29] D. Hancock and J. van der Merwe, “HyPer4: Using P4 to virtualize the programmable data plane,” in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 35–49. doi: 10.1145/2999572.2999607

[30] 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 *Proceedings of the 26th International Conference on Computer Communication and Networks (ICCCN ’17)*. Vancouver, BC, Canada: IEEE, Jul. 2017, pp. 1–9. doi: 10.1109/ICCCN.2017.8038396

[31] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, 2019. doi: 10.1109/JSAC.2019.2894308

[32] H. Soni, T. Turletti, and W. Dabbous, "P4Bricks: Enabling multiprocessing using linker-based network data plane architecture," 2018.

[33] P. Zheng, T. Benson, and C. Hu, "P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 98–111. doi: 10.1145/3281411.3281436

[34] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "P4VBox: Enabling P4-based switch virtualization," *IEEE Communications Letters*, vol. 24, no. 1, pp. 146–149, 2020. doi: 10.1109/LCOMM.2019.2953031

[35] P. Zheng, T. A. Benson, and C. Hu, "Building and testing modular programs for programmable data planes," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, pp. 1432–1447, 2020. doi: 10.1109/JSAC.2020.2986693

[36] R. Parizotto, L. Castanheira, F. Bonetti, A. Santos, and A. Schaeffer-Filho, "PRIME: Programming in-network modular extensions," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–9. doi: 10.1109/NOMS47738.2020.9110355

[37] ——, "Consistent composition and modular data plane programming," *IEEE Communications Magazine*, vol. 59, no. 6, pp. 60–65, 2021. doi: 10.1109/MCOM.001.2000904

[38] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, "Modular switch programming under resource constraints," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 193–207. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/hogan

[39] 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. 160 982–160 997, 2019. doi: 10.1109/ACCESS.2019.2950446

[40] J. Ma, S. Xie, and J. Zhao, "P4SFC: Service function chain offloading with programmable switches," in *Proceedings of the IEEE 39th International Performance Computing and Communications Conference (IPCCC)*. Austin, TX, USA: IEEE, Nov. 2020, pp. 1–6. doi: 10.1109/IPCCC50635.2020.9391530

[41] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "Compiling service function chains via fine-grained composition in the programmable data plane," *IEEE Transactions on Services Computing*, vol. 16, no. 4, pp. 2490–2502, 2023. doi: 10.1109/TSC.2023.3242072

[42] 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 *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*. Santa Clara, CA, USA: USENIX Association, Apr. 2024, pp. 477–490.

[43] T. Wang, X. Yang, G. Antichi, A. Sivaraman, and A. Panda, "Isolation mechanisms for high-speed packet-processing pipelines," in *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*. Renton, WA, USA: USENIX Association, Apr. 2022, pp. 1289–1305.

[44] 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 *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, ser. HotNets '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 69–76. doi: 10.1145/3484266.3487367

[45] 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 *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, ser. HotNets '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 91–98. doi: 10.1145/3484266.3487377

[46] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 651–665. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/xing

[47] 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 *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM '23)*. New York, NY, USA: Association for Computing Machinery, Sep. 2023, pp. 1028–1042. doi: 10.1145/3603269.3604882

[48] Internet Assigned Numbers Authority (IANA), "Protocol numbers," https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml, 2025, accessed: 2025-10-01.

[49] Open Networking Foundation, "The next-gen SDN tutorial," https://github.com/opennetworkinglab/ngsdn-tutorial, 2019, accessed: 2025-10-01.

[50] B. Goswami, M. Kulkarni, and J. Paulose, "A survey on P4 challenges in software defined networks: P4 programming," *IEEE Access*, vol. 11, pp. 54 373–54 387, 2023. doi: 10.1109/ACCESS.2023.3275756

[51] Open Networking Foundation, "OpenFlow switch specifications," https://www.opennetworking.org/sdn-resources/onf-specifications/openflow, 2015, accessed: 2025-10-01.

[52] M. Rahmati, F.-R. Boyer, B. Pontikakis, J. Pierre David, and Y. Savaria, "P4Muse: Enabling modular P4 programming via compiler-managed code merging without syntax modifications," *IEEE Access*, vol. 13, pp. 124 138–124 157, 2025. doi: 10.1109/ACCESS.2025.3589353

[53] Internet Engineering Task Force (IETF), "SRv6 network programming," 2020, internet-Draft, draft-ietf-spring-srv6-network-programming-15. [Online]. Available: https://tools.ietf.org/html/draft-ietf-spring-srv6-network-programming-15

[54] P4 Language Consortium, "Behavioral model (bmv2): The p4 reference switch," https://github.com/p4lang/behavioral-model, 2022, accessed: 2025-10-01.

[55] J. Xiao, X. Zuo, Q. Li, D. Zhao, H. Zhao, Y. Jiang, J. Sun, B. Chen, Y. Liang, and J. Li, "FlexNF: Flexible network function orchestration for scalable on-path service chain serving," *IEEE/ACM Transactions on Networking*, vol. 32, no. 3, pp. 2026–2041, 2024. doi: 10.1109/TNET.2023.3334237

[56] A. Shukla, K. Hudemann, Z. Vági, L. Hügerich, G. Smaragdakis, A. Hecker, S. Schmid, and A. Feldmann, "Runtime verification for programmable switches," *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1822–1837, 2023. doi: 10.1109/TNET.2023.3234931

[57] B. Vass, E. R. Bérczi-Kovács, Á. Fraknói, C. Raiciu, and G. Rétvári, "Charting the complexity landscape of compiling packet programs to reconfigurable switches," *IEEE/ACM Transactions on Networking*, vol. 32, no. 5, pp. 4519–4534, 2024. doi: 10.1109/TNET.2024.3424337

[58] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen, and C. Zhang, "P4 to FPGA–a fast approach for generating efficient network processors," *IEEE Access*, vol. 8, pp. 23 440–23 456, 2020. doi: 10.1109/ACCESS.2020.2970683

[59] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley, 2013.

[60] S. C. Johnson, "Yacc: Yet another compiler-compiler," AT&T Bell Laboratories, Murray Hill, NJ, USA, Computing Science Technical Report CSTR-32, Jul. 1975. [Online]. Available: https://www.epaperpress.com/lexandyacc/download/yacc.pdf