Verifiable Traffic Control with Compact Data Structures in the Data Plane

Mengying Pan

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Advisers: Jennifer Rexford, Andrew W. Appel

September 2025

Abstract

As network traffic grows in volume and diversity, operators increasingly require real-time, in-network control over packets. Tasks such as rate-limiting large flows, identifying heavy downloaders, and dropping unsolicited packets demand stateful processing at line rate—capabilities absent in traditional fixed-function switches. The advent of programmable data planes has made such control feasible, allowing custom stateful logic to run directly in the network.

However, programming traffic control remains challenging. Data planes impose strict memory constraints, making conventional data structures impractical. Instead, operators rely on approximate data structures like Bloom filters and hash tables, trading accuracy for efficiency. Selecting and configuring these structures requires expertise, as poor choices can lead to excessive errors or wasted resources.

To address this, we developed Network Approximate Programming (NAP), a high-level language centered on a versatile approximate dictionary abstraction. This abstraction captures a broad range of compact data structures while allowing programmers to specify the types of errors an application can tolerate. The NAP compiler automatically selects and configures the appropriate structure to optimize hardware utilization and compiles to P4, the native programming language for data planes—significantly simplifying development.

Ensuring correctness of approximate data structures in P4 poses additional challenges. While research has advanced new streaming algorithms, little attention has been paid to adapting them systematically to data-plane constraints and refreshing them at runtime. Moreover, these practical concerns may introduce subtle bugs. To address these concerns, we developed general synthesis and verification frameworks. We deployed an approximate sliding-window Bloom filter on Intel Tofino and verified its correctness, demonstrating how to build P4 data structures with correctness guarantees.

Finally, P4 itself presents foundational challenges: its low-level design, ambiguous specification, and lack of formal semantics make stateful programming error-prone. To provide a rigorous foundation, we developed a formal semantics for P4 based on its two-phase evaluation model. Our mechanized formalization precisely defines language constructs, including stateful externs on Intel Tofino. By adhering to P4's intended behavior, our work uncovered previously undocumented ambiguities and contributed improvements to the specification.

Together, these efforts form a cohesive framework for enabling robust network control in programmable data planes.

Acknowledgments

I am deeply grateful to my advisor, Jennifer Rexford, for her mentorship, patience, and unwavering support throughout my graduate studies. I had the privilege of working with her from my master's program through my PhD, and her guidance has shaped not only this dissertation but also my growth as a researcher. Jen's insight, generosity, and care have continually inspired me: she embodies what it means to choose wisely, pursue passion, and meet each person with the understanding they need. Her example will continue to guide me long after this PhD journey.

I would also like to thank my committee members—David Walker, Maria Apostolaki, Andrew Appel, and Hyojoon Kim—for their invaluable feedback and perspectives that strengthened this dissertation. I am especially grateful to Andrew for his mentorship and collaboration on programming languages, which have been formative in my work, and to Hyojoon for his encouragement and guidance across multiple projects.

I have been fortunate to collaborate with many wonderful colleagues during my PhD. Robert MacDavid guided me through my first steps in the world of programmable switches and taught me how to stay positive through the uncertainties of research. Danny Chen's sharp instincts and countless energizing discussions pushed our projects forward. I am also thankful to Qinshi Wang, Shengyi Wang, and Lennart Beringer, whose expertise in programming languages made the P4 semantics and verification work possible; their patience and generosity in sharing knowledge were invaluable. To my labmates—Mary Hogan, Yufei Zheng, Sophia Yoo, and Sata Sengupta—thank you for your friendship, encouragement, and inspiration, which made research not only productive but joyful. Beyond Princeton, I am grateful to Yingjie Bi and Hamid Bazzaz for their mentorship and collaboration during my internship, which broadened my perspective and enriched this work.

Most importantly, I thank my family for their endless love and encouragement. To my grandparents, extended family, and especially my parents, thank you for your sacrifices throughout my education, for traveling across the world to support me, and for always believing in me. To my three feline companions—Jane, Jewel, and Ming—thank you for your quiet company during long days and late nights, and for reminding me that life exists beyond research deadlines. Finally, to Shuo, my strongest support: thank you for your unconditional love, optimism, and brilliance, and for walking beside me through every high and low. I could not have completed this journey without you.

This research was supported in part by the DARPA Peraton Programmable Distributed Defense in Depth for 5G Service, the DARPA Peraton Enterprise ProNet: Programmable Network Telemetry in Campus Networks, and by funding from the Princeton University Provost's Office.

To my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The rapid growth of digital services has led to an unprecedented surge in network traffic, both in volume and variety. Applications such as cloud computing, video streaming, and online gaming generate massive amounts of data that must be transported across networks with minimal delay to ensure seamless performance. At the same time, the wide adoption of mobile devices, Internet of Things (IoT) systems, and hyperscale data centers has introduced highly dynamic and unpredictable traffic patterns. Unlike traditional workloads, which often followed simple and predictable traffic patterns, modern network traffic is characterized by frequent bursts, asymmetric data exchanges, and varying quality-of-service requirements.

These trends place immense pressure on network infrastructure, requiring it not only to handle higher throughput but also to adapt to fast-changing conditions in real time. To meet application-level expectations—such as low latency, fairness, and security—networks need to track flow-specific state across packets, make per-flow decisions on the fly, and respond immediately to changes in traffic behavior. This demands the ability to dynamically track the state of traffic flows and apply fine-grained control to each flow at line rate.

Traditional network devices, primarily designed for simple packet forwarding, lack the flexibility and responsiveness needed to support these capabilities. In response, a new generation of programmable switches has emerged, allowing networks to execute custom, stateful packet processing directly within the data plane. These switches enable operators to define custom processing logic, maintain state at line rate, and dynamically control traffic behavior. However, leveraging these capabilities effectively requires novel programming abstractions and optimization techniques to ensure scalability and correctness.

In the following sections, we examine categories of network control applications (Section 1.1), the limitations of traditional networking approaches (Section 1.2), and the challenges of modernizing networks through programmable architectures (Section 1.3), data structures (Section 1.4), and programming languages (Section 1.5). We conclude with a summary of contributions (Section 1.6) and an outline of the dissertation roadmap (Section 1.7).

1.1 Network Control Applications

As networks continue to evolve, the need for fine-grained real-time control over traffic has grown significantly. Network control applications enable operators to actively monitor the network state, maintain historical context for traffic flows, and make intelligent decisions on packet processing. This ability to update and query state at line rate is essential for enforcing control policies.

Network control applications can be broadly categorized into three main areas:

- **Traffic management and quality of service.** Ensuring efficient traffic flow is critical in modern networks, where applications impose strict requirements on latency and bandwidth. To meet these demands, network operators deploy control applications that dynamically adjust routing decisions based on network

conditions. For example, load balancers [1, 18, 31] distribute traffic across multiple paths to optimize resource utilization, requiring the system to maintain state on past routing decisions to ensure consistency. Active queue management mechanisms [6, 29, 51] track flow sizes and service classes, probabilistically dropping large flows to prevent bottlenecks or deprioritizing less time-sensitive traffic to improve service quality. These control applications rely on real-time state tracking to adapt to traffic patterns without introducing excessive delays.

- **Security and access control.** As network attacks become more sophisticated, real-time threat detection and mitigation are increasingly embedded directly into network infrastructure. Stateful firewalls [28, 52], for example, track active connections to enforce security policies, blocking unauthorized access attempts and filtering malicious traffic based on historical flow data. Similarly, DNS amplification mitigation systems [43, 33] monitor outgoing DNS requests, keeping state on previous DNS requests to identify and discard excessive responses. The ability to store and rapidly update state enables networks to react instantly to evolving threats without waiting for centralized control-plane intervention.

- **Network monitoring.** Real-time visibility into network performance is crucial for diagnosing issues, optimizing resource allocation, and maintaining service reliability. Modern telemetry applications continuously collect and analyze network statistics at line rate, triggering alerts when anomalies or performance degradations are detected. These applications include flow monitoring systems [50, 21] that maintain per-flow statistics, packet sampling mechanisms [13, 17] for in-depth traffic analysis, and event-driven logging [55, 27] for rapid fault detection. By embedding telemetry directly into network devices and maintaining information on traffic patterns, operators can proactively identify and resolve issues, reducing downtime and improving network efficiency.

These network control applications must operate at high speed and scale to support millions of users while maintaining minimal processing overhead. Implementing them directly within the network, rather than relying solely on centralized control or end hosts, can significantly enhance overall efficiency in responding to dynamic conditions.

1.2 Limitations of Traditional Network Devices

Traditional network devices were primarily designed for packet forwarding based on preconfigured rules, offering limited flexibility and programmability. These devices fall into two main categories: fixed-function switches and software-defined networking (SDN). While both have enabled large-scale networking, neither supports real-time, stateful traffic control.

Fixed-function switches: inflexibility and vendor dependence

Fixed-function switches have long been the backbone of network infrastructure, designed for high-speed packet forwarding with minimal processing. They operate using predefined protocols, leaving operators with little control over packet processing beyond basic header-based forwarding.

In early networks, this approach was sufficient, as complex traffic control was offloaded to end hosts or centralized controllers. However, modern control applications increasingly require in-network computation (Section 1.1), making fixed-function switches a limitation. Updating their functionality typically requires hardware modifications or vendor-supplied firmware updates, a slow and rigid process that limits adaptability to evolving traffic patterns and deployment of customized network applications.

Software-defined networking: overhead and scalability constraints

Software-defined networking (SDN) [34] improves programmability by decoupling the data plane from the control plane. A software controller program collects network state, computes forwarding rules, and installs them on the data plane via protocols like OpenFlow. Despite its increased flexibility, SDN still faces several key limitations:

- **Latency**: Stateful processing, such as connection tracking, is offloaded to the controller, introducing an external control loop with inherent delays. These delays can cause inaccuracies in policy enforcement, making some time-sensitive applications infeasible.
- **Scalability**: Relying on a centralized controller to update rules introduces overhead, particularly at high traffic volumes. Processing packets in software is too slow to match modern link speeds, leading to throughput bottlenecks in large-scale deployments.
- **Security risks**: Many security applications, such as denial-of-service (DoS) mitigation, require real-time traffic monitoring. Forwarding packets to the controller for analysis in SDN-based architectures increases the exposure to attacks due to delays in updating rules.

These limitations make SDN generally insufficient for stateful, per-packet decision-making at high speeds.

1.3 Programmable Data Plane

To overcome the limitations of traditional network devices, modern programmable switches [23, 12, 26] adopt a fundamentally different design: they integrate both packet and state processing entirely within the data plane. This enables fine-grained control at line rate, eliminating delays and throughput bottlenecks at their source.

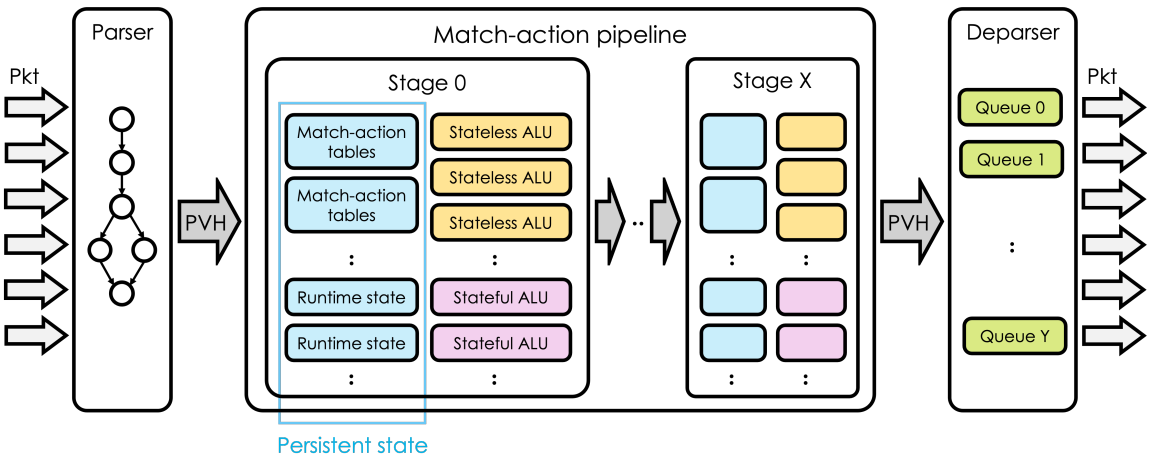


Figure 1.1: Protocol-Independent Switch Architecture.

1.3.1 Data Plane Architecture

High-speed network devices rely on specialized architectures. A widely adopted model is the **Protocol-Independent Switch Architecture (PISA)** [9], shown in Figure 1.1, which organizes packet processing into three core components:

- **Parser**: Extracts structured header fields using a state machine and stores them in a Packet Header Vector (PHV) unique to each packet. Each PHV holds parsed fields (e.g., source IP, TCP port), metadata (e.g., ingress port, arrival timestamp), and intermediate results.
- **Match-action pipeline**: A sequence of processing stages, each performing a limited number of parallel operations on the PHV and accesses to local state. Each stage includes:
  - **Match-action tables**: Match PHV fields to invoke actions based on installed rules.
  - **Stateless ALUs**: Perform actions that apply arithmetic and logic operations to PHV.
  - **Runtime state**: Stores data across packets for stateful processing.

  - **Stateful ALUs**: Perform actions that interact with the runtime state.
  - **Hash units**: Compute functions (e.g., CRC, identity, random) for indexing into runtime state.

- **Deparser**: Reconstructs outgoing packets by serializing modified headers and payloads from the PHV into a bitstream.

Both match-action tables and runtime state hold **persistent state**—information that persists across packets—but they differ significantly. Table rules store mappings from match keys to actions and are managed exclusively by the control plane. Only the control plane can update entries, while packets can only read them in the data plane. In contrast, **runtime state** is directly mutable by packets in the data plane, enabling stateful behavior.

Together, these architectural components allow programmable switches to support a wide range of network control applications directly in the data plane.

1.3.2 Data Plane Constraints

Programmable switches must balance flexibility with performance, imposing strict hardware constraints. Unlike software-based systems, which scale with additional resources, data plane programs must fit within the fixed limits of switch hardware.

Computational constraints

- **Bounded computation per packet**: The match-action pipeline has a fixed number of stages, each with a limited number of stateless ALUs, restricting the total number of operations that can be applied to a packet. Packet recirculation can extend computation, but at the cost of reducing throughput.

- **Shallow per-stage computation**: To ensure consistent packet processing speed, each stage must complete within a fixed number of clock cycles. Conse-

quently, operations within a stage must run independently in parallel—typically one per PHV container—and any dependent operations must be split across stages.

- **Limited intermediate storage**: Each packet has a fixed-size PHV, consisting of a few hundred fixed-width containers. This imposes tight limits on the temporary storage available for computation.

- **Restricted instruction set**: Only primitive operations (e.g., addition, bitwise AND) are supported. Complex operations (e.g., division, modulus, floating-point arithmetic) are unavailable due to performance cost.

Memory constraints

- **Limited on-chip memory**: The data plane has limited static random-access memory (SRAM), shared between match-action tables and runtime state. For example, Intel Tofino [23] provides only tens of megabytes in total, distributed across all stages.

- **Partitioned runtime state**: Runtime state is partitioned across stages and mapped one-to-one to stateful ALUs, with each stateful ALU accessing only its own local runtime state object. As a packet traverses the pipeline, it can access only the runtime state local to the current stage via the corresponding stateful ALU. These constraints ensure isolation and prevent data hazards.

- **Constrained state access**: Each stateful ALU can typically access only a single memory location in its associated runtime state object, with limited computation allowed during access. For example, Intel Tofino supports read-modify-write operations, but the modification step is highly restricted. Like stateless ALUs, all stateful ALUs operate independently in parallel to maintain line-rate processing.

- **No external storage**: To minimize overhead, switches avoid offloading data to external memory, requiring all data structures to reside within the data plane.

These hardware constraints enforce a very different programming mindset for network operators. They need to have a deep understanding of the architectural, computational, and memory constraints in order to take full advantage of the switch programmability.

1.4 Approximation in the Data Plane

Programmable data planes enable powerful in-network processing but operate under strict computational and memory constraints. These limitations make maintaining exact per-packet (or even per-flow) state infeasible, necessitating approximation techniques to efficiently store and process traffic data.

1.4.1 Approximate Data Structures

Many network control applications require taking different actions on individual traffic flows, but the data plane lacks sufficient memory to store per-flow state for potentially millions of flows. For example, an Intel Tofino switch provides only tens of megabytes of SRAM, whereas a layer-4 load balancer may require hundreds of megabytes for precise connection tracking.

This gap between memory availability and application demands forces network operators to turn to approximate data structures, which trade accuracy for efficiency while still enabling essential functionality. These data structures reduce memory consumption by allowing controlled errors, with their accuracy determined by available resources. Some commonly used examples include:

- **Bloom filter** [7]: A probabilistic data structure for set membership that allows false positives but never false negatives.

- **Count-min sketch** [15]: A space-efficient frequency estimation data structure that may overestimate (but never underestimate) counts due to hash collisions.
- **Hash table**: A key-value store that retains frequently accessed entries while evicting others based on capacity constraints.
- **Hash table with fingerprinting**: A variation that stores only fingerprints of keys to reduce space usage, introducing occasional false lookups.

Beyond these classic structures, researchers have developed data structures specifically tailored to the PISA model. Examples include Fridges [54], BeauCoup [11], NitroSketch [30], and CocoSketch [53], all of which demonstrate the feasibility of implementing efficient approximations in hardware-constrained environments.

While approximate data structures help address resource limitations, their effectiveness depends on how well they align with application requirements. This leads to an important question: do control applications tolerate approximation?

1.4.2 Approximate Traffic Control

While approximation is often a necessity, its impact on control applications varies. Some applications require precise state maintenance, while others can tolerate controlled errors without significantly affecting functionality. Many packet-processing applications fall into the latter category, where approximation enables scalable solutions while preserving essential behavior.

A classic example of approximation in action is NetCache [25], which is a key-value store that uses a Count-Min Sketch to estimate frequencies of querying keys. While it may overestimate counts for less popular keys, it remains space-efficient and ensures that frequently accessed items are kept in cache. To resolve queries efficiently, NetCache pairs its Count-Min Sketch with a hash table, which retains values for popular keys while occasionally evicting others.

Another critical example, which we will revisit throughout this thesis, is stateful firewalls [42], which drop unsolicited incoming traffic by tracking recent outgoing connections.

In an ideal implementation, a stateful firewall allows all outgoing packets and the solicited incoming packets. In other words, it permits incoming packets only if they match a key from recent outgoing traffic (e.g., internal/external IP pairs recorded within the last 60 seconds). However, due to limited hardware memory and high traffic rates, exact state tracking is impractical, making approximation necessary.

Importantly, different applications have different tolerances for approximation errors. For example, in an enterprise network, it may be preferable for a stateful firewall to occasionally allow unsolicited packets rather than risk blocking legitimate traffic. This trade-off is acceptable because additional security mechanisms—such as intrusion prevention systems or endpoint security filters—can mitigate the small fraction of unwanted traffic that bypasses the firewall. Conversely, incorrectly blocking legitimate traffic can disrupt critical services, making false negatives far more problematic than false positives.

The choice of approximate data structure plays a crucial role in how well the firewall balances security and performance. Specifically, for an approximate stateful firewall,

- A Bloom filter is a natural fit since it never produces false negatives, ensuring that all recorded connections remain valid. This guarantees that legitimate traffic is never mistakenly blocked.

- A hash table, on the other hand, is not suitable, as valid connections may be evicted, leading to false negatives where legitimate packets are wrongly dropped.

This example highlights a broader challenge in programmable data planes: choosing the right approximation technique and tuning it for optimal performance.

1.4.3 Challenges: Selecting & Sizing Data Structures

The stateful firewall example illustrates a key insight: while approximate data structures enable efficient traffic control under resource constraints, selecting the right data structure and tuning its parameters remain significant challenges.

Challenge 1: selecting the data structure

As more approximate data structures are developed, choosing the most suitable one becomes increasingly difficult. Each structure has its own trade-offs, and selecting the best one requires a deep understanding of application-specific requirements.

For example, while the original NetCache implementation uses a Count-Min Sketch to track popular keys, an alternative approach could employ a basic hash table to count as many keys as possible. Choosing the suitable data structure requires a deep understanding of the available options and their trade-offs, especially when memory resources are limited.

Challenge 2: sizing the data structure

Once a data structure is selected, optimizing its size is another challenge. The goal is to minimize approximation error while staying within hardware constraints. Programmable switches like the Intel Tofino offer extremely limited memory, making precise allocation critical. Overprovisioning wastes resources, while underprovisioning degrades accuracy and performance.

Unlike conventional software environment where memory allocation can be adjusted dynamically, data-plane programs must predefine resource usage at compile time. Current compilers for programmable switches do not efficiently search the implementation space, forcing developers to manually tune parameters through trial and error. This process requires significant expertise, time, and iterative experimentation.

Ultimately, achieving the right balance between accuracy and efficiency remains a fundamental challenge in deploying approximation techniques on programmable data planes.

1.5 P4 Language

Programming the specialized architectures of the data plane requires a language tailored to its unique constraints. General-purpose languages like C, which rely on dynamic memory allocation, are ill-suited for this task. Unlike software programs that execute arbitrary logic, data-plane programs operate within a structured packet-processing pipeline: the packet moves through predefined stages, each with access only to local state and no heap memory. Given this strictly linear execution model, conventional programming constructs, such as pointers, are unnecessary.

To address these constraints while enabling programmable packet processing, P4 [8, 37] was developed as a domain-specific language. P4 is both target-aware and domain-specific, serving two key roles:

- **Specification language** (Section 1.5.1): Hardware vendors define device-specific constraints and capabilities.
- **Programming language** (Section 1.5.2): Developers implement packet-processing logic, including parsing, match-action pipelines, and deparsing.

1.5.1 Specification Language

Each programmable target—whether a high-performance switch like Intel Tofino or a software-based model—provides an architecture file that dictates how P4 programs interact with the hardware. This file serves as a contract between the P4 program and the underlying hardware, specifying:

- **Switch model**: Defines the programmable components that must be declared and instantiated to form a complete switch model:
  - `Parser`: Extracts headers from incoming packets.
  - `Control`: Implements a packet-processing pipeline, modifying header fields, metadata, and runtime state.
  - `Deparser`: Reassembles headers and payload into packets.
- **Target-specific externs**: Provides vendor-defined functions and classes that extend P4's capabilities beyond its core constructs.

For example, the Very Simple Switch (VSS) architecture [37] (Figure 1.2) defines a `package` type called `Switch`, which represents the switch model (lines 14-16). To target this architecture, a P4 program must instantiate a `Switch` named `main` by supplying instances of programmable components whose type signatures match `Parser`, `Pipe`, and `Deparser`. This ensures that the program adheres to the processing model defined by the VSS architecture.

Additionally, the VSS architecture defines a `Checksum16` extern class (lines 20-28) and a `min` extern function (line 30). The imported `core.p4` library also provides extern classes such as `packet_in` and `packet_out`, which are passed into parsers and deparsers to extract packet headers from incoming bitstreams and reassemble headers back into packets, respectively (lines 9, 11). While externs provide flexibility, they also introduce implicit constraints. For instance, `Checksum16` may only be instantiated in the parser and deparser for packet validity checks. Such restrictions are often undocumented in architecture files, forcing developers to consult vendor manuals or infer correct usage through experimentation.

Beyond checksum computations, P4 supports many other externs. As introduced in Section 1.1, runtime state is fundamental to modern network control, yet P4 lacks

```p4
#include <core.p4>

struct standard_metadata_t {
  bit<4> in_port;
  bit<4> out_port;
}

/* Programmable components */
parser Parser<H>(packet_in pkt, out H hdr);
control Pipe<H>(inout H hdr, inout standard_metadata_t md);
control Deparser<H>(packet_out pkt, inout H hdr);

/* Switch model */
package Switch<H>(Parser<H> parser,
                  Pipe<H> pipe,
                  Deparser<H> deparser);

/* Target-specific externs */
// Extern class
extern Checksum16 {
  // Constructor
  Checksum16();
  // Methods
  void clear();                // clear the added data
  void update<T>(in T data);   // add data to checksum
  void remove<T>(in T data);   // remove data from existing checksum
  bit<16> get();               // get checksum for existing data
}
// Extern function
extern T min<T>(in T t1, in T t2);
```

Figure 1.2: A very simplified switch (VSS) architecture `vss.p4`.

native support for it. Instead, architectures may expose runtime state through externs, with certain externs dedicated to storing and accessing such state. This design reflects the reality that state storage and access are highly hardware-dependent, making it impractical to standardize these features in the core P4 language.

Overall, the target-dependent nature of P4 complicates stateful traffic control, as developers must navigate architecture files and vendor-specific documentation to correctly implement applications.

1.5.2 Programming Language

Beyond defining the architecture interface, P4 is also a programming language. Developers must implement the programmable components specified as parameters to the switch model.

One fundamental component is the parser, organized as a state machine with a designated `start` state and two terminal states: `accept` and `reject`. A parser comprises two main elements:

- **Declarations**: constants, variables, parser instances, extern instances, and parser states.
- **Implicit execution flow**: defined by transitions between parser states.

Each parser state specifies a sequence of **statements**—such as calling methods on extern instances, invoking the `apply` method of another parser instance, or transitioning to a new state—that process the packet bitstream into structured headers. Execution of a parser instance always begins at the `start` state and proceeds according to the state transitions.

Figure 1.3 illustrates a typical layer 3 parser targeting the VSS architecture. It extracts Ethernet and IPv4 headers and verifies the IPv4 checksum. The parser transitions to the `accept` state only if all header extractions succeed and the checksum is valid; otherwise, it transitions to `reject`.

Another key component is the control. A control consists of a list of declarations and an `apply` block that defines its execution flow. Declarations fall into four categories:

- **Constants and variables**: Declare immutable or mutable identifiers of base types (e.g., `int`, `struct`), providing local storage within the control. These can be referenced in actions, tables, and the `apply` block, similarly to parameters.

```
#include <vss.p4>
#include <headers.p4>

/* Headers */
struct header_t {
  eth_h ethernet;
  ipv4_h ipv4;
}
/* Parser */
parser IPParser(packet_in p, out header_t h)  {
  Checksum16() ck; // instantiate a checksum
  state start {
    transition parse_ethernet;
  }
  state parse_ethernet {
    p.extract(h.ethernet);
    transition select(h.ethernet.ether_type) {
      ETHERTYPE_IPV4 : parse_ipv4;
      _ : reject;
    }
  }
  state parse_ipv4 {
    p.extract(h.ipv4);
    ck.clear();
    ck.update(h.ipv4);
    verify(ck.get() == 0, error.IPv4ChecksumError);
    transition accept;
  }
}
```

Figure 1.3: An IPv4 packet parser on the VSS architecture.

- **Actions**: Define sequences of statements that manipulate variables, call top-level functions, or invoke methods of extern instances. Actions are directly invocable.

- **Tables**: Define match keys and associate them with actions to execute upon a match. Tables expose an `apply` method that is invoked to perform the match-action mapping.

- **Instantiations**: Instantiate controls or extern classes using constructors. Extern instances expose architecture-defined methods available for invocation; control instances expose an `apply` method that can be invoked to execute their `apply` block.

```
/* Top-level ordinary function */
bit<8> decr(in bit<8> x) {
  return x - 1;
}
/* Control */
control IPForwarder(inout header_t h, inout standard_metadata_t m) {
  /* Declarations */
  bit<32> next_ip_;
  action act_forward(bit<32> next_ip, bit<4> port) {
    next_ip_ = next_ip;              // variables are in scope
    m.out_port = port;
    h.ipv4.ttl = decr(h.ipv4.ttl);
  }
  action act_drop() {
    m.out_port = DROP_PORT;
  }
  table tbl_forward {
    key = { h.ipv4.dst_ip: lpm; }    // parameters are in scope
    actions = { act_forward(); act_drop(); }
    default_action = act_drop();
  }
  ...
  /* Apply block */
  apply {
    if (h.ipv4.ttl <= 1) {
      act_drop();
    } else {
      tbl_forward.apply();
      ...
    }
  }
}
```

Figure 1.4: A simple IP forwarding control targeting the VSS architecture.

The `apply` block defines the control's execution flow by invoking its actions and the methods available on its tables and instances. In addition to these locally declared entities, the `apply` block may also invoke methods of top-level extern instances and call top-level functions—namely, extern functions and ordinary functions.[1] This explicit execution flow contrasts with the parser's implicit transitions between states.

Figure 1.4 presents a simple IP forwarding control targeting the VSS architecture. A top-level ordinary function decrements a value by 1. The control declares a local variable for the next-hop IP address, two actions for forwarding and dropping packets,

[1]P4 supports functions in the conventional sense, which can only be defined at the top level.

and a forwarding table that matches on the destination IP address. The `apply` block checks the packet's TTL, drops packets if the TTL has expired, and otherwise applies the forwarding logic via the table.

Finally, it is worth noting that deparsers, though treated as a distinct programmable component at the architectural level, are implemented as controls in P4. Like regular controls, a deparser consists of declarations and an `apply` block. Its primary role, however, is to reassemble headers into a packet, typically by invoking the `emit` method of the `packet_out` extern.

Through these constructs, P4 provides a specialized yet flexible framework for customizing switch behavior and implementing sophisticated network applications. However, the P4 specification [37] remains lengthy and informal, and its domain-specific constructs often introduce subtleties that complicate correct usage. We discuss these challenges further in Section 1.5.3.

1.5.3 Challenges: P4 Semantics & Adapting Data Structures

In Section 1.4.3, we discuss the complexities of selecting and sizing data structures for traffic control in the data plane. However, choosing the right data structures is only the beginning. Implementing them correctly in P4 requires navigating two additional obstacles: adapting data structures to the target architecture, and reasoning precisely about program behavior in P4, a language that lacks a **formal semantics**. A formal semantics defines, with mathematical rigor, how programs behave—what each construct means and how it executes. Without such a foundation, it becomes difficult to reason about correctness.

Challenge 3: adapting data structures correctly

Beyond selecting the appropriate data structures, P4 programmers must also tailor them to the architectural constraints and specialized state mechanisms of the underlying hardware.

- **Memory access constraints**: Conventional data structures often assume unrestricted access to a single block of memory during method calls, but the data plane distributes memory across stages and imposes strict access constraints (Section 1.3.2). To implement such data structures efficiently, developers must redesign them to respect these limitations.
- **Lack of native time-based eviction**: Many data structures rely on time windows to discard stale information. However, the PISA pipeline architecture lacks native time-based eviction mechanisms, forcing developers to implement workarounds that increase complexity.
- **Implicit hardware constraints**: In addition to documented limitations, hardware targets impose subtle, undocumented constraints that impact performance and correctness (Section 1.5.1). These constraints often require extensive study of vendor documentation and empirical testing.

Adapting data structures for P4 is not a straightforward task. Without deep expertise in both P4 and the target hardware, implementing such structures from scratch is error-prone and difficult to debug.

Challenge 4: lack of P4 formal semantics

Unlike many general-purpose languages, P4 lacks a formal execution model, making its semantics difficult to reason about. Several factors contribute to this issue:

- **Specialized constructs**: While P4 is a low-level language, it introduces unique primitives, such as match-action tables and externs, that differ significantly from conventional imperative programming paradigms. Developers must understand these constructs to write correct and efficient programs.

- **Target-dependent execution**: Stateful externs like registers behave differently across hardware targets. Their exact semantics are often only partially specified in architecture files, with additional details informally documented in comments and vendor manuals. This lack of standardization complicates portability and correctness.

- **Ambiguous specifications**: The P4 specification, written in natural language, is prone to ambiguities and potential inconsistencies. Developers frequently resort to compiler experimentation to infer intended behavior, which introduces uncertainty into the programming process.

These challenges raise a fundamental question: What does a P4 program truly mean? Without a well-defined and rigorous semantics, reasoning about correctness and debugging unexpected behaviors remain significant obstacles.

1.6 Contributions

This dissertation addresses the four key challenges in programming traffic control in the data plane, presented in reverse order—from low-level foundations to high-level abstractions—to reflect their dependency hierarchy:

- **Challenge 4: lack of formal semantics**

  Introduces **a formal semantics for P4** in Chapter 2, improving clarity, correctness, and standardization across different P4 targets.

- **Challenge 3: deployment and verification of data structures**

  Develops **a synthesis framework** for systematically adapting approximate data structures to P4 and **a verification framework** for rigorous correctness verification in Chapter 3.

- **Challenge 2 & 1: selection and sizing of data structures**

  Designs NAP, **a high-level Network Approximate Programming Language** that automates data structure selection and parameter tuning, in Chapter 4, eliminating the need for manual optimization.

Together, these contributions advance the research of verifiable, efficient, and practical data-plane programming, laying a foundation for future innovations in network control applications.

1.6.1 Formal Semantics for P4

P4 programming lacks a well-defined formal semantics, making it difficult to reason about program behavior, ensure correctness, and standardize implementations across targets. The language's reliance on specialized constructs, target-dependent execution, and ambiguous specifications further complicates its semantics.

This dissertation introduces a rigorous two-phase execution model for P4:

- **Instantiation phase**: Models the static resource allocation of the compiler, generating an environment that maps P4 variables into compile-time known values.

- **Execution phase**: Defines how packets are processed and state is modified within the static environment.

This model aligns naturally with the static allocation of P4, offering a clear framework for reasoning about the behavior of the program. For the purpose of demonstration, we have formalized the semantics of stateful externs on Intel Tofino switches.

However, our semantics is readily extendable to other architectures and extern functionalities.

Beyond theoretical clarity, this formalization also has tangible impact: it identified 23 errors or ambiguities in the official P4 standards, 17 of which have been corrected in the latest version of the specifications and the compiler implementations. The execution model serves as a reference for compiler developers and switch vendors, enabling more predictable and reliable P4 program execution.

1.6.2 Verifiable Approximate Data Structures

Even with formal semantics, programming stateful applications in P4 remains challenging. Approximate data structures provide a practical way to manage state under strict memory constraints but introduce two major difficulties:

- **Hardware adaptation**: Mapping the data structures to P4 without violating architectural limitations.
- **Verification**: Ensuring functional correctness despite approximations.

To address the first challenge, we developed a synthesis framework for implementing approximate data structures in P4. We generalize techniques for adapting data structures so that they can operate efficiently within the constrained pipelined architecture of the data plane. The implementation in P4 consists of three key steps:

- **Preprocessing**: Establishes cleaning mechanisms to periodically clear outdated entries from the data structure.
- **State operations**: Operates on the data structure distributed across stages.
- **Postprocessing**: Merges results across stages to generate query responses.

To verify the correctness of an implementation, we also adopted a layered verification framework:

- **Concrete functional model**: A parameterized functional model that closely aligns with implementation of data structures in P4.
- **Abstract functional model**: A high-level functional model that specifies the intended functionality of the data structure.

This layered approach separates concerns across different levels of abstraction. Rather than directly verifying the P4 implementation—a complex and error-prone task—we prove that the P4 program correctly implements the concrete model, which in turn refines the abstract model. This composition yields an end-to-end correctness guarantee: the P4 program satisfies the high-level properties captured by the abstract model.

As a proof of concept, this dissertation presents a verified deployment of an approximate sliding-window Bloom filter, ensuring it satisfies the no-false-negative property. This work bridges formal verification and real-world P4 applications, offering correctness guarantees for stateful network programs.

1.6.3 A High-Level Network Control Language

Despite these advances in P4 semantics and data structure implementations, programming traffic control in P4 remains tedious and error-prone. Developers must manually select data structures and configure size parameters, as these decisions directly affect approximation accuracy and hardware feasibility. P4 compilers offer no support in these tasks, leaving developers to rely on repetitive trial and error.

To address this challenge, this dissertation introduces Network Approximate Programming (NAP), a high-level language that lifts developers above the low-level intricacies of data structure selection, configuration, and implementation, enabling them to write concise network control programs.

The central abstraction in NAP is the **approximate dictionary**, a generic key-value store where the key identifies a flow and the value encodes application-specific state. Packets are inserted under their flow keys to update state and queried to retrieve state for flow-level control decisions. This interface captures a common pattern across many network control tasks and provides a unifying abstraction over diverse approximate data structures. To model practical approximation behavior, NAP characterizes the approximate dictionary along two dimensions:

- **Inclusion dimension**: The key-value mapping may overapproximate or underapproximate the state associated with each key.
- **Temporal dimension**: The dictionary retains state only over a recent time window, rather than supporting arbitrary queries.

With these abstractions, developers express control intent at a high level, while the NAP compiler automatically selects, configures, and implements the underlying data structures. It searches the parameter space to find configurations that minimize approximation error while satisfying the hardware constraints. Although such constraints restrict how data structures can be implemented, they also help narrow the search space, enabling the compiler to solve the resulting constrained optimization problem efficiently—typically in under a second. Building on our synthesis framework, which provides verifiable modular data structure templates, the compiler generates complete P4 programs that are directly deployable on the Intel Tofino.

As a proof of concept, we extend the Lucid language with support [42] for approximate dictionaries and demonstrate NAP's expressiveness through three reusable dictionary classes—`ExistDict`, `CountDict`, and `FoldDict`. These abstractions are used to implement a range of traffic control applications, achieving concise NAP programs that exhibit approximation behavior matching the compiler's analytical predictions.

1.7 Dissertation Organization

This dissertation addresses key challenges in traffic control via three contributions:

- **Chapter 2** introduces a formal semantics for P4 that improves the clarity and correctness of P4 programming. This work was published in [47] and conducted jointly with Qinshi Wang, with artifacts in [44].

- **Chapter 3** presents frameworks for synthesizing and verifying approximate data structures. Sections 3.1 and 3.2, published in [39], are the author's work with assistance from Hyojoon Kim, with artifacts in [40]. Sections 3.3 and 3.4, published in [47], were developed jointly with Qinshi Wang, Shengyi Wang, and Lennart Beringer, with artifacts in [45].

- **Chapter 4** describes a high-level language for automating the selection and configuration of data structures. This work was published in [39] and developed by the author with assistance from Hyojoon Kim, with artifacts in [40].

Together, these contributions enable verifiable, efficient, and practical traffic control in programmable data planes, laying a foundation for future innovations in network control applications.

Chapter 2

P4 Semantics Formalization

P4 is a domain-specific language for programmable packet processing, designed to offer flexible control over how network devices handle packets. To rigorously capture the meaning of P4 programs, this chapter presents a mechanized formalization of the P4 semantics in Coq.

As discussed in Section 1.5, P4 combines high-level programmability with target-specific constraints. Accordingly, we divide its semantics into two components. The **core semantics** define the behavior of the language independent of any hardware, covering constructs such as controls, tables, and actions. The **architecture semantics** describe interactions with target-specific components, such as externs and switch models.

A formal understanding of a programming language requires a precise definition of its **evaluation model** (Section 2.1)—that is, how a program should be interpreted. For P4, this is especially important because execution is split into two distinct phases. First, the program is evaluated at compile time, where all resources are allocated and initialized. Then, at runtime, each packet is processed using these pre-instantiated resources. Our semantics reflects this two-phase model by separating evaluation into the **instantiation phase** (Section 2.2) and the **execution phase** (Section 2.3). This

separation closely matches the evaluation model described in the P4 specification, reflecting how P4 programs are compiled and run in the data plane in practice.

Previous efforts to formalize P4—most notably Petr4 [16]—borrowed techniques from functional programming language semantics, such as function closures and dynamic allocation. While classic, these abstractions diverge from P4's evaluation model and can obscure core semantics. In contrast, our semantics closely follows the specification, offering a direct and transparent account of program behavior. Section 2.4 compares our approach to Petr4, highlighting improvements in clarity, modularity, and fidelity.

To support multiple hardware targets, we define architecture semantics as modular extensions. Section 2.5 demonstrates this by formalizing the switch behavior for the VSS architecture and the extern behavior for the Intel Tofino architecture.

Finally, our formalization process systematically uncovered ambiguities and inconsistencies in the P4 specification. Throughout this effort, we engaged with the P4 Language Design Working Group to help clarify and address these issues. Section 2.6 documents our findings and shows how formal semantics can contribute not only to program reasoning, but also to language design and standardization.

2.1 P4 Evaluation Model

To support line-rate processing, programmable data planes are designed with rigid architectures and limited hardware resources (Section 1.3). In particular, each hardware component must be statically allocated, assigned to at most one P4 entity, and accessed at most once per packet, prohibiting dynamic reallocation at runtime. Reflecting this constraint, the P4 specification [37] defines an abstract model that evaluates programs in two distinct stages:

- **Static instantiation**: At compile time, all instantiations are evaluated with compile-time-known constructor parameters.
- **Dynamic execution**: At runtime, each incoming packet executes invocations using packet-specific runtime parameters to completion.

This two-phase model imposes a fundamental restriction: P4 forbids dynamic instance creation at runtime. Controls, parsers, and extern classes must be instantiated during compilation. As a result, many techniques from general-purpose programming, such as dynamic memory allocation and higher-order functions, are not applicable to P4.

Understanding this two-phase evaluation model is critical for developing a faithful formal semantics for P4. In this section, we illustrate the hierarchical instantiation process through a concrete example targeting the Intel Tofino architecture.

2.1.1 Hierarchical Instantiation on Intel Tofino

In Chapter 1.5, we introduced the VSS architecture to explain core P4 constructs. To illustrate real-world instantiation behaviors, we now turn to the Intel Tofino architecture [24], which provides a richer set of externs, including externs for runtime state.

Figure 2.1 shows a fragment of the Tofino architecture. It defines its switch model as a `Switch` package, parameterized by one or more `Pipeline` instances (line 13), each requiring a parser, control, and deparser instance for ingress and egress packet processing (line 9). The architecture also exposes two extern classes: `Register`, which provides runtime state (lines 16-23), and `RegisterAction`, which enables user-defined state accesses on registers (lines 24-31).

Figure 2.2 shows a P4 program targeting this architecture. The program, `count_forward.p4`, defines two controls: `CountForwarder` (lines 17-32), which

```
# include <core.p4>

/* Programmable components */
parser IngressParserT<H, ..>(packet_in pkt, out H hdr, ..);
control IngressT<H, ..>(inout H hdr, ..);
control IngressDeparserT<H, ..>(packet_out pkt, inout H hdr, ..);

/* Switch model */
package Pipeline<IH, ..>(IngressParserT<IH, ..> ingress_parser,
                         IngressT<IH, ..> ingress,
                         IngressDeparserT<IH, ..> ingress_deparser,
                         ..);
package Switch<..>(Pipeline<..> pipe0, ..);

/* Target-specific extern classes for state and state access */
extern Register<T, I> {
    // Constructor
    Register(bit<32> size);
    Register(bit<32> size, T initial_value);
    // Methods
    T read(in I index);
    void write(in I index, in T value);
}
extern RegisterAction<T, I, U> {
    // Constructor
    RegisterAction(Register<T, I> reg);
    // Methods
    U execute(in I index);
    abstract void apply(inout T value, optional out U rv);
    ..
}
..
```

Figure 2.1: Snippet from the Intel Tofino architecture `tna.p4`.

counts and forwards packets, and `Ingress` (lines 33-46), which dispatches packets to the appropriate counter based on the transport protocol. Inside `Ingress`, two independent instances of `CountForwarder` are created to separately count and forward TCP and UDP traffic. Controls may have optional constructor parameters, such as `pbr` in this example, which determines whether policy-based routing applies to TCP and UDP traffic. Both the register state used for counting packets and the table state used for storing forwarding mappings are independent across the two instances. This setup illustrates how each instantiation produces its own copy of constructor parameters and internal stateful entities: registers, register actions, and tables.

```
#include <tna.p4>
#include <headers.p4>

parser Layer4Parser(..) {                  // call start parser state
  state start {..}
  state parse_ethernet {..}
  ..
}
parser IngressParser(packet_in p, out header_t h, ..) {
  /* Instantiate a parser */
  Layer4Parser() layer4_parser;
  state start {
    layer4_parser.apply(..);               // call parser's apply method
    transition accept;
  }
}
control CountForwarder(..)                 // runtime parameters
                      (bool pbr){          // constructor parameters
  bit<32> num_pkts;
  /* Instantiate extern classes */
  Register<..>(..) reg_cnt;
  RegisterAction<..>(reg_cnt) ra_incr = {..};
  action act_incr {
    num_pkts = ra_incr.execute(0);         // call extern's method
  }
  /* Instantiate a table implicitly */
  table tbl_forward {..}
  apply {
    act_incr();                            // call action
    if (pbr) { tbl_forward.apply(); } // call table's apply method
  }
}
control Ingress(inout header_t h, ..) {
  /* Instantiate controls */
  CountForwarder(true) tcp_ctrl;
  CountForwarder(true) udp_ctrl;
  table tbl_forward {..}
  apply {
    tbl_forward.apply();
    if (h.tcp.isValid()) {
      tcp_ctrl.apply(..);                  // call control's apply method
    } else {
      udp_ctrl.apply(..);                  // call control's apply method
    }
  }
}
control IngressDeparser(packet_out p, inout header_t h, ..) { .. }
.. // egress parser, control and deparser
Pipeline(IngressParser(), Ingress(), IngressDeparser(), ..) pipe;
Switch(pipe) main;
```

Figure 2.2: Instantiations and invocations in `count_forward.p4` on Intel Tofino.

This program reflects a broader principle: in P4, parsers, controls, and extern classes must be explicitly instantiated. These entities behave like classes in object-oriented programming: instances are created using constructors, and their methods are invoked on those instances. Controls and parsers each expose a single `apply` method, which, when called, executes their corresponding execution flow [Section 13.10; Section 14.4 [37]].[1] Tables are unique: although they encapsulate stateful mappings, they are implicitly instantiated when defined, since a table is intended to be used only once by its enclosing control instance. Like controls and parsers, tables expose an `apply` method that triggers the table's match-action evaluation when invoked [Section 6.6.2 [37]]. In contrast, parser states, actions, extern functions, and ordinary functions can be regarded as functions: they lack constructors and are directly invokable.

Throughout this thesis, we use the term **class** to refer to declarations of parsers, controls, and extern classes, and the term **instance** to refer to their instantiated copies. We use the term **function** to refer to actions, tables, parser states, extern functions, and ordinary functions. We collectively refer to methods and functions as **invocables**. Instantiation in P4 is recursive: instantiating a class recursively instantiates all classes and tables declared within it, ensuring isolation between instances. Table 2.1 summarizes the properties of these different kinds of invocables, with further discussion in the referenced sections.

2.1.2 Toward a Two-Phase Semantics

Our formal semantics adopts P4's two-phase evaluation model, distinguishing:

[1] Conceptually, a parser may be regarded as providing an implicit `apply` method that consists of a single statement invoking the `start` parser state.

Entity	Parser	Control	Extern Class	Table	Parser State	Action	Ordinary Function	Extern Function
Type	Class				Function			
Invocable Entity	Method				Self			
Non-Extern Invocable (§2.2.2)	✓	✓		✓	✓	✓	✓	
Procedural Invocable (§2.3.4)	✓	✓			✓	✓	✓	
Scope Path (§2.3.1)	Curr. Inst.	Curr. Inst.	Curr. Inst.	Encl. Inst.	Encl. Inst.	Encl. Inst.	Top level	Top level
Declared In (§1.5.2)	Top-Level	Top-Level	Top-Level	Control	Parser	Control	Top-Level	Top-Level
Instantiated In (§1.5.2)	Parser, Package[a]	Control, Package[a]	Parser, Control, Package[a], Top-Level	Control	—	—	—	—
Invoked In (§1.5.2)	Parser state	Apply block	Parser state, Apply block, Action	Apply block	Parser state	Apply block, Action	Parser state, Apply block, Action, Ordin. Func.	Parser state, Apply block, Action

[a] Packages can be instantiated but not programmed. Controls, parsers, and externs can be instantiated anonymously as constructor parameters to packages [Appendix F, [36]].

Table 2.1: Properties of classes and functions in our semantics after preprocessing.

- **Instantiation phase** (Section 2.2): Initializes persistent state and constructs a static environment from declarations, binding globally unique paths to compile-time-known entities such as code definitions, constants, and extern metadata.
- **Execution phase** (Section 2.3): Models runtime packet processing by evaluating statements along the control flow, with lookups performed against the static environment.

This separation mirrors statically allocated structures in C and module instantiation in hardware description languages like Verilog [38], where reusable modules are replicated and bound to fixed hardware resources at compile time.

To support this separation, we design an intermediate language, **P4light**, that cleanly separates declarations from statements. We generate type-annotated P4light abstract syntax trees (ASTs) from P4 source programs using a front end adapted from Petr4 [16]. The AST is implemented in the Coq proof assistant as a family of inductive types, with distinct types for syntactic entities such as expressions, statements, and declarations.

To prepare the ASTs for the two-phase semantics, we introduce a preprocessing pass that hoists the results of function calls and other side-effectful subexpressions into temporary variables, preserving evaluation order while simplifying statements. Additionally, all control flow is confined to statements by converting the initializers of variable declarations into initialization statements inserted at the start of the control flow.

For example, an assignment declared in a control block:

```
a = f(x + g(y)) + z;
```

is normalized to:

```
t1 = g(y);
t2 = f(x + t1);
a = t2 + z;
```

where the local variable `a` is initialized at the beginning of the `apply` block.

This normalization enforces a strict separation: declarations describe only compile-time-known constructs, while runtime computation is expressed as a flat sequence of statements. This design simplifies the semantics and aligns with P4's operational constraints. The resulting AST provides a formal foundation for specifying the instantiation and execution phases, detailed in the following sections.

2.2 Instantiation Phase

The instantiation phase consists of two critical tasks: recording static information and initializing persistent state. First, it establishes a naming scheme, assigning **globally unique paths** and **locally unique locators** based on the program's namespace hierarchy. Then, it binds the globally unique paths of constants, instances, and invocables to their corresponding compile-time known information, including constant

values, instance references, extern metadata, and code definitions. Collectively, these bindings form the **static environment**. In parallel, the instantiation phase initializes the internal state of extern instances, such as registers, ensuring isolation across instances. Once constructed, the static environment is passed unchanged into the execution phase, providing the complete foundation for evaluating the P4 program, with the initialized state serving as its starting point.

2.2.1 Globally Unique Paths and Locally Unique Locators

To support the instantiation and execution phases, our semantics assigns a globally unique path to every P4 entity and a locator to every name in a P4 program. We distinguish two related naming mechanisms:

- **Globally unique paths**: Fully qualified paths used for entries in static environment and persistent state.
- **Locally unique locators**: Partial paths relative to their enclosing scope used for stack frame addressing during execution.

This subsection focuses on how globally unique paths and locally unique locators are constructed during the instantiation phase. Their usage will be discussed later in Section 2.2.2 and Section 2.3.1.

Globally unique paths

According to the P4 specification [Section 18.3, [37]], the control plane uses fully qualified names to uniquely configure entities such as registers and tables at runtime. We adopt this naming scheme directly: in our semantics, control-plane names serve as the globally unique paths for all P4 entities—classes, instances, invocables, tables, variables, and constants.

Paths are allocated based on the hierarchical structure of P4 namespaces, following these rules:

- **Top-level declarations**: Classes, instances, functions, and constants declared at the top level use their local identifiers as globally unique paths. For example, in Figure 2.2, the package instance created by `Switch(pipe, ..) main` is named `main`, and the control class defined by `control CountForwarder(..)` is named `CountForwarder`.

- **Nameless instantiations**: When an instance is constructed inline as an argument, its globally unique path is derived by appending the parameter name to the enclosing instance's path. In the example above, the `Ingress()` instance passed to the `Pipeline` constructor is named `pipe.ingress`, where `ingress` is the parameter name defined in the architecture (see Figure 2.1, line 9).

- **Nested declarations**: Instances and variables declared within a class are named by concatenating the enclosing instance's path with the local name. Likewise, invocables defined within a class are named by appending the local name to the enclosing class's path. Tables receive their instance paths and class paths in the same way. For example, in Figure 2.2, the `tcp_ctrl` instance declared inside the `Ingress` control is named `pipe.ingress.tcp_ctrl`. Similarly, the table `tbl_forward` inside the `tcp_ctrl` instance has an instance path `pipe.ingress.tcp_ctrl.tbl_forward`, and its apply method is named `CountForwarder.tbl_forward.apply`.

This hierarchical naming scheme guarantees global uniqueness by relying on local uniqueness within each scope. It also preserves structural information: entities created under the same parent share a common prefix, while instances produced by instantiation share a common suffix.

Locally unique locators

To statically distinguish between different bindings of the same identifier across scopes, we annotate every name in the abstract syntax tree with a **locator** after parsing and type checking. Locators compactly and uniquely identify variables within the current namespace, enabling efficient access during execution without dynamic name lookup. Locators come in two forms:

- `LGlobal p`: for names defined in the top-level scope.
- `LInstance p`: for names defined within a class scope.

Here, `p` is a path relative to its current scope. Specifically, for `LInstance p`, `p` refers to the suffix of the globally unique path after removing the enclosing parser, control, or extern instance's prefix. For example, a local variable `x` declared inside a control is annotated as `LInstance x`, while a variable `x` defined inside an action `act` within the same control is annotated as `LInstance act.x`. Since P4 places strict restrictions on where classes and functions may be declared (Table 2.1), the resulting locator hierarchy remains shallow.

```
Definition loc_to_path (this : path) (loc : Locator) : path :=
  match loc with
  | LGlobal p => p
  | LInstance p => this ++ p
  end.
```

Figure 2.3: Converting locators to globally unique paths in Coq.

As shown in Figure 2.3, globally unique paths can be reconstructed from locators by maintaining the current scope during execution. For an `LGlobal p` locator, the globally unique path is simply `p`. For an `LInstance p` locator, it is obtained by appending the relative path to the enclosing scope. For example, in Figure 2.2, the register declared inside `CountForwarder` has the locator `LInstance reg_cnt`; when

accessed within the current path `pipe.ingress.tcp_ctrl`, its globally unique path resolves to `pipe.ingress.tcp_ctrl.reg_cnt`.

Together, globally unique paths and locally unique locators form the naming scheme in our semantics. Paths uniquely identify all P4 entities in the static environment (Section 2.2.2) and the persistent state (Section 2.3.1), while locators serve as static locations in the stack frame (Section 2.3.1).

2.2.2 Static Environment

With globally unique paths in place, the instantiation phase constructs the **static environment**: a mapping from paths to compile-time known information, including references, code definitions, extern metadata, constants, and types. In particular, it captures the code definitions of **non-extern invocables**—that is, invocables programmed without reliance on architecture-specific semantics (i.e., all invocables except extern methods and extern functions). Once constructed, this environment remains unchanged throughout the execution phase. The static environment Γ consists of the following components:

- Γ_{def}: Maps paths of non-extern invocables to their code definitions.
- Γ_{inst}: Maps paths of instances to pairs of class names and instance locations.
- Γ_{ext}: Maps paths of extern instances to their metadata (e.g., register size).
- Γ_{const}: Maps paths of constants to compile-time known constant values, including constructor parameters and constant declarations.
- Γ_{typ}: Maps paths to compile-time known type definitions.
- Γ_{senum}: Maps paths of serializable enum members to their numeric values.

Among these, Γ_{def}, Γ_{inst}, and Γ_{ext} record static information about classes, instances, and functions, while Γ_{const}, Γ_{typ}, and Γ_{senum} record compile-time known values and types.

For example, consider the `tcp_ctrl` instance declared in Figure 2.2. Here, Γ_{const} stores its constructor parameter at `pipe.ingress.tcp_ctrl.pbr`. Γ_{inst} maps `pipe.ingress.tcp_ctrl` to `(CountForwarder, pipe.ingress.tcp_ctrl)`, while Γ_{def} maps `CountForwarder.apply` to the apply block spanning lines 28–31. If an alias to this instance is introduced, such as `copy_ctrl`, then Γ_{inst} contains an additional entry mapping `pipe.ingress.copy_ctrl` to `(CountForwarder, pipe.ingress.tcp_ctrl)`. This ensures references are correctly resolved.

As another example, a register instance in the Intel Tofino architecture is represented in Γ_{ext} as a tuple `(IndexWidth` $\times$ `ElementType` $\times$ `RegisterSize)`. These metadata are supplied as constructor parameters and stored for use during subsequent execution of the extern instance.

By separating code definitions, reference paths, and extern metadata, this design avoids redundancy and supports modular analysis of classes and functions.

2.2.3 Static Instantiation and Initialization

Of the six components of the static environment, Γ_{def}, Γ_{typ}, and Γ_{senum} can be constructed directly by collecting relevant information in a single pass. In particular, code definitions are inserted into Γ_{def} by recursively traversing declarations. Figure 2.4 presents simplified pseudocode for the instantiation process, focusing on how the remaining components—Γ_{inst}, Γ_{const}, and Γ_{ext}—are built. It also shows how the initial persistent state of extern instances, s_{pst}, is populated; we will discuss this further in Section 2.3.1.

In the pseudocode, the `instantiate_prog` procedure iterates over top-level declarations. Controls and parsers are recorded into `decl_env` for later lookup when

```
global Γ_inst, Γ_const, Γ_ext, s_pst, decl_env := []

procedure update(curr_path, local_env_r, local_env_w, decl) :=
    next_path := curr_path ++ (decl.name)
    if decl is an instantiation then
        instantiate(next_path, local_env_r, decl)
    else if decl is a constant then
        v := evaluate(local_env_r, decl)
        local_env_w := local_env_w[decl.name -> v]
        if v is a value then
            Γ_const := Γ_const[next_path -> v]
        else // a constant instance reference
            Γ_inst := Γ_inst[next_path -> v]

procedure instantiate(prev_path, local_env, decl) :=
    class_name := decl.class_name
    curr_path := prev_path ++ (decl.name)
    local_env_init := local_env
    for each param in decl.params
        decl' := to_decl(param)
        update(curr_path, local_env_init, local_env, decl')
    Γ_inst := Γ_inst[curr_path -> (class_name, curr_path)]
    if class_name is an extern then
        args := local_env(decl.params)
        (v_static, v_init) := construct_extern(Γ_ext, s_pst, class_name, args)
        Γ_ext := Γ_ext[curr_path -> v_static]
        s_pst := s_pst[curr_path -> v_init]
    else // class_name is a parser or control
        body := decl_env[class_name]
        for each decl' in body
            update(curr_path, local_env, local_env, decl')

procedure instantiate_prog(prog) :=
    local_env := []
    for each decl in prog
        if decl is a class then
            decl_env := decl_env(decl.name -> decl)
        else
            if decl is an instantiation then
                instantiate(ϵ, local_env, decl)
                v := (decl.class_name, decl.name)
            else
                v := evalutate(local_env, decl)
            local_env := local_env[decl.name -> v]
```

Figure 2.4: Pseudocode for instantiation and initialization.

their instances are encountered (line 37)[2]. Meanwhile, `local_env` tracks compile-time known values for evaluating constants (line 44). For each instantiation declaration, `instantiate` is invoked at the top-level path ϵ (line 40).

The `instantiate` procedure takes an instantiation and the enclosing scope. It computes the current instance's globally unique path by appending its local name to the scope and adds an entry to Γ_{inst} (lines 17–22). For extern instances, a backend-specific `construct_extern` function updates their static configuration in Γ_{ext} and initializes their runtime state in s_{pst} (lines 23–27) (see Section 2.5.2).

Constructor parameters are first converted into declarations via `to_decl` and then processed by `update`, which either evaluates their constant values or recursively instantiates them (lines 19–21). Evaluated values are stored in Γ_{const} (line 11), while references to instances are recorded in Γ_{inst} (line 13). For control and parser instances, their inner declarations are similarly evaluated or instantiated (lines 28–31).

2.3 Execution Phase

Building on the static environment established during the instantiation phase, we now turn to the execution phase, where we formally define the runtime behavior of a P4 program during packet processing. This phase interprets statements using the static environment as a guide and operates over a formal notion of **program state**, which captures all information relevant to execution.

Unlike general-purpose programming languages, P4 is designed with hardware constraints in mind—most notably, the absence of loops and recursion guarantees that all P4 programs terminate. Consequently, we adopt a **big-step operational semantics** to model execution: each P4 construct is evaluated in a single, inductive step from an initial to a final program state.

[2] `decl_env` is an environment mapping class names to closures, where each closure contains the code definition and an environment to resolve free names.

This section introduces the key components of P4 execution semantics:

- Section 2.3.1: formal definition of program state;
- Section 2.3.2: operational rules governing program evaluation;
- Section 2.3.3: nondeterminism in semantics arising from uninitialized bits;
- Section 2.3.4: semantics of invocation, which lie at the heart of the evaluation model, demonstrating how the static environment is utilized during execution.

2.3.1 Program State

Before diving into presenting the operational semantics, we first define the structure over which execution operates: the program state. This state represents a snapshot of all variables and persistent state relevant to a packet's traversal through a P4 program. Here, "program state" refers specifically to the programming language concept, distinct from notions like data-plane state or traffic state.

Formally, a program state is a pair of two maps:

$$\begin{aligned} \text{State} &:= \text{StackFrame} \times \text{PersistentState} \\ \text{StackFrame} &:= \text{Locator} \to \text{StorableValue} \\ \text{PersistentState} &:= \text{Path} \to \text{StateObject} \end{aligned}$$

- StackFrame (s_{local}) maps locators to **storable values**, representing all local variables in the current scope. These variables are specific to each packet, so a fresh StackFrame is created for every packet. Since P4 does not support dynamic pointers, variables are accessed directly via syntactic paths rather than memory addresses. (See Section 2.3.3 for details on "storable values.")

- PersistentState (s_{pst}) maps globally unique paths to **state objects**. These objects represent persistent state shared across packets, so a single PersistentState is maintained for all packets. For example, Tofino registers are modeled as arrays stored in PersistentState, initialized during instantiation (Section 2.2.3) and updated during execution.

Local variables are accessed through `LInstance p` locators in the stack frame. These relative paths remain locally unique within the current scope. To ensure correct scoping and variable lifetimes, the stack frame is either preserved when entering a new scope or refreshed on entry and restored on exit. Specifically, invoking a table, action, or parser state reuses the current stack frame, since these constructs access variables declared in the enclosing parser or control instance and must preserve their updates after returning. Variables declared within these invocables remain stored in the stack frame but are inaccessible outside their scope due to their pre-assigned locators. In contrast, invoking a new parser, control, extern instance, or top-level function allocates a fresh stack frame at the callee's scope, which is discarded upon return (see Section 1.5.2 and Figure 1.4).[3]

Stateful entities such as tables and extern instances need to be uniquely identified across packets and scopes. Therefore, unlike local variables, state objects are always keyed by their globally unique paths in the persistent state. This design ensures consistent access across scopes and packets, mirroring the conventions used in the static environment.

This distinction between transient state and persistent state provides a clean foundation for defining the operational semantics.

[3]A top-level function is scoped at ϵ as all its local variables are annotated with `LGlobal` locators (see Section 2.2.1).

2.3.2 Operational Semantics

Having defined the structure of the program state, we now present the operational semantics for P4 execution. We adopt a big-step operational semantics, evaluating each language construct in a single inductive step. In contrast to small-step semantics, which decomposes execution into fine-grained transitions suited for modeling nontermination, big-step semantics provides a more concise and natural framework for P4. Since P4 programs are guaranteed to terminate, big-step semantics aligns well with the language's execution model.

The semantics is defined with respect to three key inputs: the static environment Γ, the path p of the current scope, and the program state s. We adopt several notational conventions in the semantic rules. Variables—such as expressions *exp*, statements *stmt*, and storable values v—are written in *italics*, while constants are typeset in roman. A semicolon is appended to *stmt* to emphasize its role as a statement, and an overline denotes a list of elements (e.g., $\overline{v_{\mathsf{in}}}$ for a list of input arguments).

Execution is formalized by the following core judgments, which define the semantics of statements, call expressions, and invocable execution:

$$\Gamma, p, s \vdash stmt; \Downarrow (s', sig) \qquad \text{(statement, 16 rules)}$$

$$\Gamma, p, s \vdash exp_{\mathsf{call}}(\overline{exp_{arg}}) \Downarrow (s', sig) \qquad \text{(call-expression, 3 rules)}$$

$$\Gamma, p, s \vdash (def, \overline{v_{\mathsf{in}}}) \Downarrow (s', \overline{v_{\mathsf{out}}}, sig) \qquad \text{(invocable execution, 3 rules)}$$

These judgments capture the high-level relations that govern invocation of functions and methods according to the control flow in a P4 program. Each judgment is defined by concrete inference rules, which describe the evaluation behavior inductively based on the syntactic form being processed.

Evaluation of the core judgments relies on the following six auxiliary judgments, which handle expression evaluation, l-value operations, and invocable resolution:

$$s \vdash lv \Downarrow_{\text{read}} v \qquad \text{(l-value read, 6 rules)}$$

$$s \vdash lv := v \Downarrow_{\text{write}} s' \qquad \text{(l-value write, 9 rules)}$$

$$\Gamma, p, s \vdash exp \Downarrow v \qquad \text{(expression, 18 rules)}$$

$$\Gamma, p, s \vdash exp \Downarrow (lv, sig) \qquad \text{(l-expression, 5 rules)}$$

$$\Gamma, p, s \vdash \overline{(dir, exp_{\mathsf{arg}})} \Downarrow \overline{(v, lv)} \qquad \text{(argument list, 5 rules)}$$

$$\Gamma, p \vdash exp_{\mathsf{call}} \Downarrow_{\text{lookup}} (p_{\mathsf{scope}}, p_{\mathsf{def}}) \qquad \text{(invocable lookup, 5 rules)}$$

These auxiliary judgments provide the underlying mechanisms for interpreting the statement and invocation semantics.

As an illustrative example, the judgment $\Gamma, p, s \vdash stmt; \Downarrow (s', sig)$ reads: "With the static environment Γ, current scope p, and initial state s, execution of statement $stmt$; produces a new state s' and a signal sig." The signal sig captures control flow effects to handle return and exit statements in addition to normal completion. Figure 2.5 shows a selection of inference rules for the statement judgment. In each rule, the premises (above the line) must be satisfied in order to derive the conclusion (below the line).

- E-StmtSeq1: If the first statement completes normally, execution proceeds with the second statement.
- E-StmtSeq2: If the first statement yields an abnormal control signal (e.g., return), the second statement is skipped.
- E-StmtReturn: A return statement evaluates its expression and emits a return signal.

$$\frac{\Gamma, p, s \vdash stmt_1 \Downarrow (s', \mathsf{normal}) \qquad \Gamma, p, s' \vdash stmt_2 \Downarrow (s'', sig)}{\Gamma, p, s \vdash stmt_1; stmt_2; \Downarrow (s'', sig)} \text{ E-StmtSeq1}$$

$$\frac{\Gamma, p, s \vdash stmt_1 \Downarrow (s', sig) \qquad sig \neq \mathsf{normal}}{\Gamma, p, s \vdash stmt_1; stmt_2; \Downarrow (s', sig)} \text{ E-StmtSeq2}$$

$$\frac{\Gamma, p, s \vdash exp \Downarrow v}{\Gamma, p, s \vdash \mathsf{return}\ exp; \Downarrow (s, \mathsf{return}\ v)} \text{ E-StmtReturn}$$

$$\frac{\Gamma, p, s \vdash exp_{\mathsf{call}}(\overline{exp_{\mathsf{arg}}}) \Downarrow (s', sig)}{\Gamma, p, s \vdash exp_{\mathsf{call}}(\overline{exp_{\mathsf{arg}}}); \Downarrow (s', sig)} \text{ E-StmtCall}$$

Figure 2.5: Selected semantic rules for statements.

- E-StmtCall: A call expression is evaluated by delegating to the call-expression judgment (explained in Section 2.3.4).

These rules exemplify the recursive nature of big-step semantics: evaluating a compound construct involves recursively evaluating its components, potentially invoking other core or auxiliary judgments.

2.3.3 Nondeterministic Semantics

One subtle, yet important, aspect of P4 execution is the nondeterminism introduced by uninitialized bits. This arises from hardware constraints and optimization strategies employed by P4 compilers.

In programmable data planes, uninitialized variables often share PHV containers with initialized ones. Since fixed-width ALUs operate on entire containers, uninitialized bits may be unintentionally altered by operations on neighboring fields. While isolating uninitialized bits would avoid this issue, it is generally infeasible: switches like Intel Tofino provide only 200–300 fixed-width containers, and such separation would quickly exhaust temporary storage. As a result, reading from an uninitialized variable yields an unspecified value, and subsequent reads may produce different re-

```
/* Reading an uninitialized variable */
bit<8> x, y, z;
y = x;
z = x; // z may differ from y
/* Partially initializing a variable */
x[6:0] = 0; // Top bit is uninitialized; x may read as 0 or 128
x = x << 1; // Fully initialized; x reads as 0
```

Figure 2.6: Nondeterminism in reading P4 uninitialized variables.

sults (see Figure 2.6). This behavior is acceptable in P4, as well-defined programs are not expected to rely on the contents of uninitialized variables.

To model this formally, we introduce **storable values** whose bits may be 0 or 1 when initialized, or $\perp$ when uninitialized. Rather than marking an entire value as uninitialized, we adopt this bit-level representation to support operations like shifting or slicing that may partially propagate undefined bits (see Figure 2.6). Semantically, when a variable is declared but not initialized, its storable value is filled with $\perp$ bits. When such a variable is evaluated, its unspecified bits are nondeterministically replaced with either 0 or 1.

While the P4 specification did not specify when this replacement occurs, we clarify it based on discussions with the P4 Language Design Working Group. For instance, in our semantics, storable values are determinized before being used as operands in arithmetic expressions, on the right-hand side of assignments, or as input arguments to an invocable. Conversely, storing the evaluation results back into the stack frame converts two-valued bits back to three-valued storable bits.

Our semantics thus formalizes P4's inherent nondeterminism, permiting multiple valid outcomes for the same uninitialized variable. We interpret an abstract judgment $a \Downarrow b$ in our operational semantics as representing one possible outcome, not the only one. This forms the basis for verification under nondeterminism: a program satisfies a property if all possible outcomes satisfy that property. In contrast, other P4 semantics like Petr4, resolve nondeterminism by selecting a single outcome—

e.g., defaulting uninitialized bits to zero using a "havoc" operator. While this is a valid interpretation, it is not sufficient for verifying program correctness under full nondeterminism. For instance, in Figure 2.6, such semantics would force both `y` and `z` to read as 0, masking the fact that uninitialized reads may differ. We elaborate on this distinction in Section 2.4.4.

2.3.4 Invocation Semantics

The heart of P4 execution lies in the semantics of invocations, which encompass calling functions and instances' methods. Invocations serve as the bridge between the control flow and the invocable entities, relying on the static environment generated during instantiation.

The call-expression judgment:

$$\Gamma, p, s \vdash exp_{\mathsf{call}}(\overline{exp_{\mathsf{arg}}}) \Downarrow (s', sig)$$

dispatches execution based on the invoked entity's type. There are three rules that govern this judgment; two are illustrated in Figure 2.7.[4]

The primary distinction between these two rules lies in the scope of the stack frame. As discussed in Section 2.3.1, inner functions can access the caller's local variables, while instances and top-level functions allocate new stack frames. Apart from this, evaluation follows four common steps, corresponding to the four lines in the premises:

1. **Invocable lookup**: The **invocable lookup judgment** determines both the scope of the stack frame and the path of the invocable p_{def}. If the scope is $\star$, the caller's stack frame is reused; otherwise, a new stack frame is allocated. The code definition def is then retrieved from the static environment Γ_{def} using p_{def}.

[4]The omitted rule handles P4 built-in functions and methods such as `isValid()` (Figure 2.2, line 40). The evaluation requires no lookup, and a built-in invocable execution judgment is used.

$$\frac{\begin{array}{c}\mathsf{kind}(exp_{\mathsf{call}}) \neq \mathsf{builtin} \\ (1)\quad \Gamma, p \vdash exp_{\mathsf{call}} \Downarrow_{\mathsf{lookup}} (\star, p_{\mathsf{def}}) \quad def = \Gamma_{\mathsf{def}}(p_{\mathsf{def}}) \\ (2)\quad \overline{dir} = \mathsf{dirs}(exp_{\mathsf{call}}) \quad \Gamma, p, s \vdash \overline{(dir, exp_{\mathsf{arg}})} \Downarrow (\overline{v_{\mathsf{in}}}, \overline{lv}) \\ (3)\quad \Gamma, p, s \vdash (def, \overline{v_{\mathsf{in}}}) \Downarrow (s', \overline{v_{\mathsf{out}}}, sig) \\ (4)\quad s' \vdash \overline{lv := v_{\mathsf{out}}} \Downarrow_{\mathsf{write}} s''\end{array}}{\Gamma, p, s \vdash exp_{\mathsf{call}}(\overline{exp_{\mathsf{arg}}}) \Downarrow (s'', sig)}\ \text{E-ExpCall1}$$

$$\frac{\begin{array}{c}\mathsf{kind}(exp_{\mathsf{call}}) \neq \mathsf{builtin} \\ (1)\quad \Gamma, p \vdash exp_{\mathsf{call}} \Downarrow_{\mathsf{lookup}} (p_{\mathsf{scope}}, p_{\mathsf{def}}) \quad def = \Gamma_{\mathsf{def}}(p_{\mathsf{def}}) \\ (2)\quad \overline{dir} = \mathsf{dirs}(exp_{\mathsf{call}}) \quad \Gamma, p, s \vdash \overline{(dir, exp_{\mathsf{arg}})} \Downarrow (\overline{v_{\mathsf{in}}}, \overline{lv}) \\ (3)\quad \Gamma, p_{\mathsf{scope}}, ([\,], s_{\mathsf{pst}}) \vdash (def, \overline{v_{\mathsf{in}}}) \Downarrow ((\_, s'_{\mathsf{pst}}), \overline{v_{\mathsf{out}}}, sig) \\ (4)\quad (s_{\mathsf{local}}, s'_{\mathsf{pst}}) \vdash \overline{lv := v_{\mathsf{out}}} \Downarrow_{\mathsf{write}} s''\end{array}}{\Gamma, p, (s_{\mathsf{local}}, s_{\mathsf{pst}}) \vdash exp_{\mathsf{call}}(\overline{exp_{\mathsf{arg}}}) \Downarrow (s'', sig)}\ \text{E-ExpCall2}$$

Figure 2.7: Selected semantic rules for call expressions.

2. **Argument evaluation**: The **argument list judgment** evaluates arguments into storable values and l-values according to parameter directions. P4 employs a copy-in/copy-out mechanism: in arguments become determinized storable values that will be copied into parameters in step 3; out arguments become assignable l-values for receiving output values in step 4; inout arguments serve both roles. Arguments are thus split into two lists: $\overline{v}$ for in/inout arguments and $\overline{lv}$ for out/inout arguments.

3. **Invocable execution**: The **invocable execution judgment** evaluates the retrieved code definition def with input arguments, producing a new state, output values, and a return signal.

4. **Parameter copying-out**: The **l-value write judgment** assigns output values back to arguments in the caller's stack frame and, together with the updated persistent state, generates the final program state.

Among these steps, the invocable lookup and execution judgments are particularly critical and hence are detailed below.

Invocable lookup judgment

The invocable lookup judgment queries the static environment to resolve the invocable referenced in a call expression to its stack frame scope and class path. Figure 2.8 presents the semantic rules, using the static environment Γ_{inst} to resolve class names and instance paths. The first two rules handle identifier expressions for function calls, while the remaining three handle member expressions for method calls. All identifiers are pre-annotated with locators ($n@locator$), and $\cdot$ denotes path concatenation.

$$\frac{}{\Gamma, p \vdash n@(\mathsf{glob}\ p_{\mathsf{fun}}) \Downarrow_{\mathsf{lookup}} (\varepsilon, p_{\mathsf{fun}})}\ \text{E-LGlob}$$

$$\frac{\Gamma_{\mathsf{inst}}(p) = (n_{\mathsf{class}}, p)}{\Gamma, p \vdash n@(\mathsf{inst}\ p_{\mathsf{fun}}) \Downarrow_{\mathsf{lookup}} (\star, n_{\mathsf{class}} \cdot p_{\mathsf{fun}})}\ \text{E-LInst}$$

$$\frac{\mathsf{kind}(n_{\mathsf{tbl}}@(\mathsf{inst}\ p_{\mathsf{tbl}})) = \mathsf{table} \qquad \Gamma_{\mathsf{inst}}(p) = (n_{\mathsf{class}}, p)}{\Gamma, p \vdash n_{\mathsf{tbl}}@(\mathsf{inst}\ p_{\mathsf{tbl}}).n_{\mathsf{meth}} \Downarrow_{\mathsf{lookup}} (\star, n_{\mathsf{class}} \cdot p_{\mathsf{tbl}} \cdot n_{\mathsf{meth}})}\ \text{E-LTable}$$

$$\frac{\mathsf{kind}(n_{\mathsf{inst}}@(\mathsf{glob}\ p_{\mathsf{inst}})) \neq \mathsf{table} \qquad \Gamma_{\mathsf{inst}}(p_{\mathsf{inst}}) = (n_{\mathsf{class}}, p'_{\mathsf{inst}})}{\Gamma, p \vdash n_{\mathsf{inst}}@(\mathsf{glob}\ p_{\mathsf{inst}}).n_{\mathsf{meth}} \Downarrow_{\mathsf{lookup}} (p'_{\mathsf{inst}}, n_{\mathsf{class}} \cdot n_{\mathsf{meth}})}\ \text{E-LMemGlob}$$

$$\frac{\mathsf{kind}(n_{\mathsf{inst}}@(\mathsf{inst}\ p_{\mathsf{inst}})) \neq \mathsf{table} \qquad \Gamma_{\mathsf{inst}}(p \cdot p_{\mathsf{inst}}) = (n_{\mathsf{class}}, p'_{\mathsf{inst}})}{\Gamma, p \vdash n_{\mathsf{inst}}@(\mathsf{inst}\ p_{\mathsf{inst}}).n_{\mathsf{meth}} \Downarrow_{\mathsf{lookup}} (p'_{\mathsf{inst}}, n_{\mathsf{class}} \cdot n_{\mathsf{meth}})}\ \text{E-LMemInst}$$

Figure 2.8: Semantic rules for invocable lookup.

- E-LGlob: For top-level extern and ordinary functions, a new stack frame is allocated at the global scope ϵ.
- E-LInst: For local actions and parser states, the caller's stack frame is reused.
- E-LTable: For a table's apply method, the caller's stack frame is reused.
- E-LMemGlob: For methods of global extern instances, a new stack frame is allocated at scope p'_{inst}.
- E-LMemInst: For methods of local control, parser, and extern instances, a new stack frame is allocated at scope p'_{inst}.

In general, these rules first resolve class names and references of new scope using Γ_{inst}. The full class path is then constructed by appending function and method names as needed.

Invocable execution judgment

The invocable execution judgment evaluates the code definition retrieved from the static environment. Semantically, tables are treated differently from other non-extern invocables: while tables declaratively map keys to actions, other non-extern invocables specify control flow through a sequence of statements. We refer to the latter as **procedural invocables**. To reflect this distinction, the semantics defines separate rules for procedural invocables, tables, and externs, as shown in Figure 2.9.

$$\frac{\begin{array}{c} s \vdash \overline{p_{\mathsf{in}} := v_{\mathsf{in}}} \Downarrow_{\mathsf{write}} s' \\ \Gamma, p, s' \vdash stmt \Downarrow (s'', \mathsf{return}\ v) \\ s'' \vdash \overline{p_{\mathsf{out}}} \Downarrow_{\mathsf{read}} \overline{v_{\mathsf{out}}} \end{array}}{\Gamma, p, s \vdash (\mathsf{fproc}\ (\overline{p_{\mathsf{in}}}, \overline{p_{\mathsf{out}}}, stmt), \overline{v_{\mathsf{in}}}) \Downarrow (s'', \overline{v_{\mathsf{out}}}, \mathsf{return}\ v)}\ \text{E-FProc}$$

$$\frac{\begin{array}{c} \Gamma, p, s \vdash \overline{key} \Downarrow \overline{v} \qquad s_{\mathsf{pst}}(p.n_{\mathsf{table}}) = \overline{entry} \\ (\overline{v}, \overline{kind}, \overline{entry}) \Downarrow_{\mathsf{match}} n_{\mathsf{action}}(\overline{exp_2}) \qquad (n_{\mathsf{action}}, \overline{exp_1}) \in \overline{action} \\ \Gamma, p, s \vdash n_{\mathsf{action}}(\overline{exp_1}, \overline{exp_2}) \Downarrow (s', \mathsf{return\ null}) \end{array}}{\Gamma, p, s \vdash (\mathsf{ftable}(n_{\mathsf{table}}, \overline{key}, \overline{kind}, \overline{action}), [\,]) \Downarrow (s', [\,], \mathsf{return}\ n_{\mathsf{action}})}\ \text{E-FTable}$$

$$\frac{\overline{dv_{\mathsf{in}}} := [\overline{v_{\mathsf{in}}}]_v \qquad \Gamma, p, s_{\mathsf{pst}} \vdash (n_{\mathsf{class}}, n_{\mathsf{meth}}, \overline{dv_{\mathsf{in}}}) \Downarrow_{\mathsf{ext}} (s'_{\mathsf{pst}}, \overline{dv_{\mathsf{out}}}, sig)}{\Gamma, p, (s_{\mathsf{local}}, s_{\mathsf{pst}}) \vdash (\mathsf{fextern}\ (n_{\mathsf{class}}, n_{\mathsf{meth}}), \overline{v_{\mathsf{in}}}) \Downarrow ((s_{\mathsf{local}}, s'_{\mathsf{pst}}), [\overline{dv_{\mathsf{out}}}]_{sv}, sig)}\ \text{E-FExt}$$

Figure 2.9: Semantic rules for invocable execution.

- E-FProc: Applies to procedural invocables. Evaluation copies input arguments into parameters in the stack frame, executes the body statement, and finally reads output parameters.
- E-FTable: Applies to tables. Key expressions are evaluated to values; table entries are retrieved from the persistent state s_{pst}. A matching entry is

selected based on the key values, invoking the corresponding action via the call-expression judgment.

- E-FExt: Applies to extern functions and methods. Evaluation is delegated to a backend-specific extern judgment. Since nondeterminism arises from core P4 semantics, the extern judgment operates over plain values, requiring value conversions before and after evaluation.

In summary, invocation semantics bridge static instantiation and dynamic execution, precisely capturing the interaction between compile-time program structure and runtime behavior.

2.4 Comparison with Petr4

Petr4 [16] introduced the first formal semantics for P4, consisting of a pen-and-paper semantics for a subset of the core language and an executable OCaml interpreter. Although these two artifacts were designed to be consistent, they are not formally connected by mechanized proofs. In contrast, our semantics is fully mechanized in the Coq proof assistant, and our reference interpreter—an executable OCaml program—is extracted directly from the operational semantics. The interpreter shares code with the formal relations and is proven correct with respect to them.

Beyond these implementation differences, the two approaches reflect a deeper divergence in semantic design philosophy. Petr4 evaluates P4 programs by mixing instantiation and execution, adopting techniques from general-purpose languages such as dynamic memory allocation, instantiation, and initialization. In contrast, our semantics leverages P4's domain-specific constraints to cleanly separate instantiation from execution. Specifically, our instantiation phase:

- assigns static locators and paths, eliminating dynamic memory allocation,

- builds a static environment, eliminating dynamic instantiation, and
- initializes state objects, eliminating dynamic initialization.

In the following subsections, we compare the two semantics in detail.

2.4.1 Allocation of Memory Locations

In Petr4's core semantics, the execution state consists of two maps:

$$\text{Environment } \epsilon := \text{Name} \to \text{Location}$$

$$\text{Store } \sigma := \text{Location} \to \text{Value}$$

The environment ϵ maps names to dynamically allocated memory locations, and the store σ maps locations to runtime values, including both data (such as variables and state objects) and code (such as closures for functions).

Memory locations in Petr4 are allocated at runtime in two cases:

- Names of state objects use their control-plane names as locations.
- All other names—including variables, instances, classes, and functions—allocate fresh locations dynamically, typically managed by an incrementing counter.

Accessing a name thus requires a two-step lookup: first through the environment ϵ to find the location, and then through the store σ to retrieve or update the value.

In contrast, our program state stores only runtime data. Code definitions, constants, and types are stored separately in the static environment Γ.

Our semantics eliminates dynamic memory allocation entirely by assigning all locations statically during the instantiation phase (Section 2.2.1). Variables and state objects are each assigned unique locators and paths. As a result, our program state no longer keeps track of unused locations nor maintains a dynamic mapping from

names to locations, as Petr4 does. Instead, variables and state objects are accessed directly through their statically assigned locations, eliminating a layer of indirection.

This design not only simplifies the operational semantics but also aligns with the P4 compilation process, where memory layouts are determined statically.

2.4.2 Initialization of State Objects

State objects maintain architecture-defined or user-defined state that persists across packets. While both Petr4 and our semantics associate state objects with control-plane names for consistent identification, their initialization strategies differ:

- **Petr4:** State objects are initialized dynamically at runtime. Each access checks whether the object has been initialized: if not, a new object is created on the spot; otherwise, the existing object is reused. This dynamic check introduces an asymmetry between the first and subsequent accesses, complicating the execution semantics by requiring conditional branches.

- **Our semantics:** State objects are initialized statically during the instantiation phase (Section 2.2.3). Precomputing state objects before packet execution ensures that they are available at the start of processing, mirroring how externs are preconfigured in the data plane. This design guarantees uniform execution across packets and simplifies both reasoning and implementation.

2.4.3 Instantiation of Non-Extern Classes

Since Petr4 uses a single-phase execution model that interleaves instantiation and execution, it evaluates non-extern class declarations, class instantiations, and invocable declarations at runtime by constructing closures. Formally, a closure is a record containing:

- the code definition (both declarations and statements), and

- an environment ϵ capturing the variable bindings in scope at closure creation.

Capturing the environment ϵ ensures that updates to free variables remain visible when the closure is eventually used for execution.

Closures in Petr4 take several forms:

- Declaring a control stores a constructor closure in the store σ.[5]
- Instantiating a control or declaring a non-extern invocable stores a general closure in the store σ.
- Declaring a table stores a table closure in the store σ.

$$
\begin{array}{lrll}
\text{value } val & ::= & \mathsf{cclos}(\epsilon, \mathsf{ctrl}(\overline{d\ x : \tau})(\overline{x_{\mathsf{c}} : \tau_{\mathsf{c}}})\{\overline{decl}\ stmt\}) & \text{(constructor closure)} \\
& | & \mathsf{clos}(\epsilon, \overline{X}, \overline{d\ x : \tau}, \tau, \overline{decl}\ stmt) & \text{(general closure)} \\
& | & \mathsf{table}\ \ell(\epsilon, \overline{key}, \overline{act}) & \text{(table closure)} \\
& | & \ldots &
\end{array}
$$

For example, when a control is instantiated from its constructor closure, Petr4 evaluates the constructor parameters $\overline{x_c}$, stores their values in the store σ and the closure's environment ϵ, and stores the resulting general closure—all at fresh memory locations to ensure instance isolation. Later, invoking the control's `apply` method retrieves the closure, restores closure's environment ϵ, and executes the stored code.

While closure-passing mirrors techniques from functional programming, it adds runtime complexity: closures and runtime values intermingle in the store σ, and code definitions are duplicated across instances.

In contrast, our semantics handles class instantiation and function definitions statically:

- Instantiation records each instance's path and class name in the static environment Γ_{inst}, resolving any references. Constructor parameters and constants are evaluated and stored in Γ_{const}.

[5]Petr4 does not support parsers, but they would likely be handled analogously.

- Code definitions for non-extern invocables are stored once in Γ_{fun}. Since declarations are fully processed during instantiation, only statements are retained in Γ_{fun}.

- At runtime, our semantics manage variable scope through stack frames using the invocable lookup judgment (Section 2.3.4), without storing environments of free variables dynamically.

This static approach eliminates closure-passing, reduces runtime complexity, and enables modular reasoning about classes and functions.

2.4.4 Handling of Uninitialized Bits

Both Petr4 and our semantics acknowledge that reading uninitialized bits can yield nondeterministic behavior. However, they handle this nondeterminism differently:

- **Petr4:** Models uninitialized reads using a "havoc" operator that can produce arbitrary sequences of values. The operator can be parameterized to allow target-specific behavior. However, the interpreter concretizes a single sequence per execution, limiting systematic exploration of nondeterminism.

- **Our semantics:** Represents uninitialized bits explicitly as $\perp$, with nondeterministic operational rules that admit all possible behaviors (Section 2.3.3). This systematic treatment allows reasoning about all possible outcomes, faithfully capturing hardware-level nondeterminism.

The fundamental differences between Petr4 and our semantics stem from our design choices aimed at improving clarity, aligning with P4's compilation model, and enabling mechanized verification. Specifically, our semantics:

- separates instantiation from execution,

- eliminates dynamic memory allocation,
- pre-initializes state objects,
- avoids dynamic instantiation and closure-passing, and
- systematically models nondeterminism.

These features provide a simpler foundation that aligns with the P4 language specification and supports building verification systems for P4—a broader objective explored in Wang's dissertation [46].

2.5 Architecture Semantics

Although architecture is not part of P4's core semantics, it is essential for formalizing program evaluation. Each P4 architecture semantics defines:

- **Switch model** (Section 2.5.1): execution flow over programmable components.
- **Extern semantics** (Section 2.5.2): behavior of architecture-specific externs.

Our semantics is parameterized over an architecture, providing modular support across hardware targets. This parameterization is formalized through the `Target` class (Figure 2.10), where `exec_prog` defines the switch model as an inductive relation, and `extern_sem` defines the extern semantics as a class. Supporting a new architecture requires supplying a `Target` instance specifying both components.

2.5.1 Switch Model

P4 program execution begins with the top-level package instance `main`, parameterized by the architecture's parser, control, and deparser components. The switch model semantics governs how these components are invoked and connected to process each packet.

```
Class Target := {
  exec_prog: ..;
  extern_sem: ExternSem;
}.
Class ExternSem := {
  extern_info: Type;
  extern_env := PathMap.t extern_info;
  state_object: Type;
  persistent_state := PathMap.t state_object;
  construct_extern: ..;
  exec_extern: ..;
}.
```

Figure 2.10: Coq interface for architecture semantics.

Formally, execution is described by a relation linking the input packet and initial persistent state to the output packet and final persistent state. Figure 2.11 illustrates this inductive relation for the VSS architecture.

$$\frac{\begin{array}{c} s_{\mathsf{pst\_1}} := s_{\mathsf{pst\_0}}[\mathsf{main.parser.pkt} \mapsto pkt_{\mathsf{in}}; \mathsf{main.deparser.pkt} \mapsto [\,]] \qquad \_\mathrm{md} := \{port_{in}; 0\} \\ \Gamma, \epsilon, ([\_md], s_{\mathsf{pst\_1}}) \vdash \mathsf{main.parser.apply}(\_hdr) \Downarrow (s_2, \mathsf{normal}) \\ \Gamma, \epsilon, s_2 \vdash \mathsf{main.pipe.apply}(\_hdr,\ \_md)) \Downarrow (s_3, \mathsf{normal}) \\ \Gamma, \epsilon, s_3 \vdash \mathsf{main.deparser.apply}(\_hdr)) \Downarrow ((\_, s_{\mathsf{pst\_4}}), \mathsf{normal}) \\ pkt_{\mathsf{out}} := s_{\mathsf{pst\_4}}(\mathsf{main.deparser.pkt}) \end{array}}{\Gamma, s_{\mathsf{pst\_0}}, pkt_{\mathsf{in}}, port_{in} \vdash \mathsf{Switch(..)\ main} \Downarrow (s_{\mathsf{pst\_4}}, pkt_{\mathsf{out}})}$$

Figure 2.11: Inductive relation for the switch model in the VSS architecture.

This relation updates the persistent state with the input packet pkt_{in} and initializes the stack frame with architecture-specific metadata $\_md$. It sequentially invokes the `parser`, `pipe`, and `deparser` using the call-expression judgment, wiring parameters between them. For example, the `parser` outputs the extracted header (`_hdr`) as an `out` parameter, which is passed as an `inout` parameter to the pipeline and deparser.[6] In effect, the switch model orchestrates parameter preparation, coordinates

[6]The `packet_in` and `packet_out` parameters are not passed explicitly; as directionless parameters, they must be compile-time-known and are treated as constructor parameters, not part of the runtime call.

the invocation of `main`'s components, and produces the final output packet pkt_{out}, maintaining persistent state across execution.

The VSS architecture is simple: it processes one packet at a time in strict sequence. By contrast, architectures like Intel Tofino introduce additional mechanisms—such as packet replication, mirroring, and resubmission—which require modeling extra built-in pipelines. We explore these architecture-specific complexities in [48].

2.5.2 Extern Semantics

The extern semantics describe the behavior of architecture-defined extern classes and functions. These are encapsulated in the `ExternSem` class (Figure 2.10), whose key components include:

- `extern_env` (Γ_{ext}): the extern environment (Section 2.2.2), which stores extern instance metadata, `extern_info`;
- `persistent_state` (s_{pst}): the persistent state (Section 2.3.1), which stores state objects, `state_object`;
- `construct_extern`: the instantiation function (Section 2.2.3), which collects `extern_info` and initializes `state_object` during the instantiation phase;
- `exec_extern`: the extern judgment (Section 2.3.4), which defines the semantics of extern method and function calls.

We now focus on the last component, `exec_extern`. Among the roughly 40 extern classes and functions defined in Intel Tofino, we highlight two key stateful externs: `Register` and `RegisterAction`. As defined in Figure 2.1, a `Register` holds runtime state and exposes `read` and `write` methods. A `RegisterAction` wraps a register and provides an `execute` method, which performs a read-modify-write sequence by invoking the user-defined abstract method `apply` to compute modification.

```
1  Register<bit<32>, bit<1>>(1, 0) reg_cnt;
2  /* Abstract methods are implemented in the initializer block. */
3  RegisterAction<bit<32>, bit<1>, bit<32>>(reg_cnt) ra_incr = {
4    void apply(inout bit<32> value, out bit<32> rv) {
5      value = value + 1;
6      rv = value;
7    }
8  };
```

Figure 2.12: `Register` and `RegisterAction` for counters in `count_forward.p4`.

Abstract methods are essential in Tofino because registers are bound to pipeline stages and can be accessed only once per packet. As a result, operations requiring both a read and a write, such as incrementing a counter, must be performed atomically within a single access. The abstract method provides a programmable mechanism for specifying these in-place updates.

Figure 2.12 shows the expanded code from `count_forward.p4` (Figure 2.2), where a `RegisterAction` is instantiated to increment the `Register` instance. Later, invoking `ra_incr.execute(0)` inside the action `act_incr` increments the zeroth register cell, writes back the result, and returns it to `num_pkts` (line 24, Figure 2.2).

Figure 2.13 presents selected semantic rules formalizing these three behaviors.

$$\frac{\begin{array}{c} width, type, size := \Gamma_{\mathsf{ext}}(p) \qquad reg := s_{\mathsf{pst}}(p) \qquad \mathsf{width}(i) = width \\ v := (0 \leq i < size)\ ?\ reg(i)\ :\ [[type]_{sv}]_v \end{array}}{\Gamma, p, s_{\mathsf{pst}} \vdash (\text{“Register”}, \text{“read”}, [i]) \Downarrow_{\mathsf{ext}} (s_{\mathsf{pst}}, [\,], \mathsf{return}\ v)}\ \text{E-Tf-RegRead}$$

$$\frac{\begin{array}{c} width, type, size := \Gamma_{\mathsf{ext}}(p) \qquad reg := s_{\mathsf{pst}}(p) \qquad \mathsf{width}(i) = width \\ s'_{\mathsf{pst}} := (0 \leq i < size)\ ?\ s_{\mathsf{pst}}[p \mapsto reg[i \mapsto v]]\ :\ s_{\mathsf{pst}} \end{array}}{\Gamma, p, s_{\mathsf{pst}} \vdash (\text{“Register”}, \text{“write”}, [i, v]) \Downarrow_{\mathsf{ext}} (s'_{\mathsf{pst}}, [\,], \mathsf{return\ null})}\ \text{E-Tf-RegWrite}$$

$$\frac{\begin{array}{c} p_{\mathsf{reg}} := \Gamma_{\mathsf{ext}}(p) \qquad def := \Gamma_{\mathsf{ext}}(p.\mathsf{apply}) \\ width, type, size := \Gamma_{\mathsf{ext}}(p_{\mathsf{reg}}) \qquad reg := s_{\mathsf{pst}}(p_{\mathsf{reg}}) \qquad \mathsf{width}(i) = width \\ v_{reg} := (0 \leq i < size)\ ?\ reg(i)\ :\ [[type]_{sv}]_v \\ \Gamma, p, ([\,], s_{\mathsf{pst}}) \vdash (def, [v_{reg}]_{sv}) \Downarrow ((\_, s_{\mathsf{pst}}), [v'_{reg}, v_{ret}], \mathsf{return\ null}) \\ s'_{\mathsf{pst}} := (0 \leq i < size)\ ?\ s_{\mathsf{pst}}[p_{\mathsf{reg}} \mapsto reg[i \mapsto v'_{reg}]]\ :\ s_{\mathsf{pst}} \end{array}}{\Gamma, p, s_{\mathsf{pst}} \vdash (\text{“RegisterAction”}, \text{“execute”}, [i]) \Downarrow_{\mathsf{ext}} (s'_{\mathsf{pst}}, [\,], \mathsf{return}\ v_{ret})}\ \text{E-Tf-RActExe}$$

Figure 2.13: Selected semantic rules for extern judgment in Intel Tofino.

- E-Tf-RegRead: Reads the register value at index i if valid; otherwise, returns a nondeterministic value.
- E-Tf-RegWrite: Writes a value to index i if valid; otherwise, leaves the register unchanged.
- E-Tf-RActExe: Executes a `RegisterAction` by reading the register at index i, invoking the `apply` method, writing back the updated result, and outputting the return value.

2.6 Clarifying the P4 Specification

Although P4 has gained traction as both a specification and programming language, its official definition lacks a precise semantic foundation. The P4 specification—a 170-page document maintained by the Language Design Working Group (LDWG)—is written in a mix of informal prose, code snippets, diagrams, and grammar rules. While generally well-organized, it omits a formal semantic model, leaving many language constructs ambiguous or underspecified.

This lack of formal grounding has also led to inconsistencies across the P4 ecosystem. Open-source compilers such as `p4c` implement the language based on interpretations of the informal specification, resulting in divergences and, in some cases, bugs.

Our formalization establishes a rigorous semantic foundation for P4 by closely following the specification (version 1.2.2) [36]. When we encountered omissions, inconsistencies, or ambiguities, we filed issues in the specification repository and, when applicable, reported related compiler bugs in the `p4c` repository. In total, 23 specification issues were filed, 17 of which have been resolved and incorporated in the latest version (version 1.2.5) [37]; 4 compiler issues, were filed, all of which have been addressed. Table 2.2 summarizes these findings.

	Section	Issue	Git	Status	p4c Bug
Expression	8.5, 8.6	Bit slicing index types are not clearly defined.	955	Released	No
	8.5	Concatenation is incorrectly omitted from the allowed operations on unsigned bitstrings.	956	Released	No
	8.5, 8.6	Concatenation is not properly excluded from the binary operations that require same-type operands.	956	Released	No
	8.9.2	Concatenation and shift are not properly excluded from the binary operations that permit implicit casts.	957	Released	No
	8.9.2	Implicit cast rules for serializable enums are unclear.	958	Released	Yes
	8.7	Right operand types of shift are not explicitly defined.	959	Released	No
	8.11, 8.12	Implicit conversions of lists, tuples, structs, and headers are not clearly specified.	953	Stalled	No
	8.10-12, 8.14-15	Allowed comparisons between lists, tuples, structs, and headers are not clearly specified.	960	Stalled	Yes
	8.10	Explicit casts incorrectly omit derived types.	961	Released	No
	8.7	Bit slicing of integers is incorrectly unspecified.	1015	Released	No
	8.22	The determinization when reading uninitialized bits is vague and confusing.	988	Stalled	Maybe
	8.13	Allowed types in set operations are underspecified.	969	Released	No
Name	17.3	Control plane objects incorrectly omit value sets.	962	Released	No
	6.8	Name duplication and name shadowing are undefined.	974	Stalled	Maybe
	6.4	Naming conventions for built-in methods, fields and keywords are inconsistent.	1004	Stalled	Yes
Instantiation	11.3, App.H	Instantiation is incorrectly permitted as a possible statement.	975	Released	Yes
	17.2	Compile-time known values are not clearly specified.	932	Released	Maybe
	12.10, 13.4	The behavior of local instantiations and variables during parser and control instantiation is underspecified.	926	Released	No
	10.3.1	Abstract methods introduce back doors that may allow undesired behavior such as recursion and parser invocation in controls.	973, 976, 979	Stalled	Maybe
Invocable	App.F	Parameter type rules incorrectly omit extern functions as a possible invocable.	972	Released	No
	6.7.2	Optional parameters are incorrectly disallowed in parser and control types.	977	Released	No
	13.2	Default action does not correctly default to `NoAction` when not specified in a table.	933	Released	No
	13.1	Sources of action data are defined ambiguously.	914	Released	No

Table 2.2: Specification issues identified during our formalization of P4 semantics.

Section gives the section number in the P4 specification (version 1.2.2) [36]. **Git** gives the issue number in the P4 specification repository [35]. A **status** of "stalled" means the issue is acknowledged but unresolved; "released" indicates the fix is included in a published version. Some issues also reflect `p4c` compiler bugs.

Over half of the issues involve expressions such as bit slicing, shifting, and casting. Because their behavior is described in prose, the specification is often inconsistent or incomplete. For example, issue 956 concerns the concatenation operator (Figure 2.14): although the specification claims that concatenation applies to both signed and unsigned bitstrings, it is listed only under operations for signed bitstrings and omitted from the list of binary operators that permit mixed signedness.

```
bit<8> x = 8w0x0001;
int<4> y = 4s0xFF;
bit<16> xx = x ++ x;      // 16w0x00010001
int<8> yy = y ++ y;       // 8s0xFFFF
int<12> xy = y ++ x;      // 12s0xFF0001
bit<12> yx = x ++ y;      // 12w0x0001FF
```

Figure 2.14: `++` concatenates two bitstrings, taking the signedness of the left operand.

Another example, issue 958, highlights ambiguity in the treatment of serializable enums. The specification permits implicit casts to the enum's underlying bitstring type but does not specify where such casts are valid. We clarified this by exhaustively testing cases in `p4c`, consulting the LDWG, and submitting a comprehensive enumeration of valid cast scenarios. Discrepancies were reported to the `p4c` repository.

Beyond expressions, we uncovered more fundamental flaws in the evaluation model, particularly in instantiations and invocables. Issue 975 revealed that the grammar erroneously allowed instantiations as statements, despite the fact that neither `p4c` nor Tofino supports this. This discrepancy led to a correction of the grammar. Similarly, issue 933 was resolved by specifying that a table's default action defaults to `NoAction` when unspecified.

While we resolved most issues through pull requests, six remain open. These reflect deeper challenges in P4's design and formalization, which fall into three categories: generalization difficulties, backward compatibility, and hardware constraints.

Generalization complexities: operations on composite types

Issues 953 and 960 concern implicit conversions between composite types, including list expressions, tuples, struct-valued expressions, structs, and headers. The specification lists a few isolated cases—such as assigning a list to a struct or comparing a struct-valued expression to a struct—but lacks general rules. In contrast, `p4c` supports a broader and more coherent set of conversions that resemble implicit casts. Ideally, these behaviors would be formalized as implicit casts, but this would require new types for list and struct-valued expressions, along with support for nested casts. Moreover, since `p4c` currently implements these conversions as internal rewrites, aligning them with cast-based semantics would entail substantial changes to this compiler. The LDWG has not yet reached consensus on how to proceed with this redesign.

Backward compatibility: naming rules

Issue 974 notes that the specification does not clearly define name duplication or shadowing. Formalizing stricter rules would risk breaking legacy code across targets. Similarly, issue 1004 highlights inconsistent naming conventions: built-in methods appear in both `camelCase` (e.g., `isValid`) and `snake_case` (e.g., `push_front`). Because many of these names are exposed in control-plane APIs, renaming them would break compatibility with existing tools.

Hardware constraints: abstract methods and uninitialized bits

Abstract methods, introduced primarily to support Tofino externs such as `RegisterAction`, expose two key design dilemmas.

Issue 973 illustrates a tension between expressiveness and safety: allowing abstract methods to call methods of the same instance increases functionality, but risks enabling recursion—explicitly prohibited by the specification. Issue 976 reflects a similar trade-off: enabling instantiations alongside abstract method definitions in an

initializer block increases expressiveness but may inadvertently violate the existing instantiation restrictions—such as allowing instantiating a control within a parser. In both cases, added flexibility weakens safety guarantees.

Issue 979 illustrates a different dilemma: abstract methods were originally restricted to top-level variables and its parameters to align with a simple hardware access model. However, practical use cases in Tofino require access to variables from the enclosing scope, which was later enabled via ad hoc annotations like `@synchronous`. This retrofit favors pragmatism at the cost of semantic clarity.

These challenges highlight the difficulty of evolving abstract methods into an architecture-agnostic construct. Given their limited use outside Tofino, it remains unclear how to develop abstract methods into a robust, architecture-neutral abstraction.

Issue 988 concerns uninitialized bits. While the specification permits their reads to return nondeterministic values, it does not specify when such reads occur. Our semantics clarifies this through a careful discussion with the LDWG (Section 2.3.3), but the LDWG has not yet converged on a definition that balances semantic clarity with hardware efficiency.

Reflections

These unresolved issues underscore the difficulty of evolving a domain-specific language like P4. Efforts to improve semantic clarity are often constrained by competing goals: language generality, hardware specificity, and backward compatibility. Many issues stem from leaning too far in one direction—overgeneralizing the language or tailoring it too narrowly to hardware use cases.

In summary, our formalization revealed numerous flaws in the P4 specification, most of which have now been addressed. Others remain open due to fundamental trade-offs between semantic precision and implementation realities. This experience

demonstrates the value of mechanized semantics not only for reasoning about programs, but also for shaping language design.

It is worth noting that other formalization efforts, such as Petr4, have also uncovered specification bugs. However, their focus was on building a working interpreter for a specific target architecture. As a result, they made simplifying assumptions rather than questioning specification gaps, and thus did not uncover some of the broader issues we identified.

Chapter 3

Verifiable Modular Data Structures

Chapter 2 formalized the semantics of P4, including stateful behavior on the Intel Tofino architecture. However, enabling stateful network applications in practice requires more than semantic clarity—it demands the ability to design, implement, and verify the underlying data structures in the data plane.

To maintain per-flow state at line rate under tight resource constraints, programmable switches commonly use **approximate data structures**, which trade accuracy for scalability. Deploying these structures in hardware introduces two key challenges:

- **Design adaptation**: Translates a high-level data structure into a hardware-efficient implementation that both respects architectural constraints and remains up-to-date with fast-changing traffic.
- **Correctness assurance**: Ensures that the resulting implementation faithfully preserves the intended properties, despite low-level adaptations.

This chapter addresses both challenges through two key contributions:

- **A modular synthesis framework** (Section 3.1): Decomposes a data structure into modular building blocks for an efficient and scalable P4 implementation; and
- **A layered verification framework** (Section 3.3): Connects the low-level implementation to high-level specifications using two models of the data structure linked by refinement.

The synthesis framework introduces a modular design based on two units: **rows**, which **shard** state across pipeline stages, and **panes**, which **rotate** to maintain a sliding window of data. Each data structure is synthesized by instantiating these components as control blocks, combined with common preprocessing and postprocessing logic. This modular design improves code clarity, enables parameterized code generation, and supports semi-modular verification: since rows and panes are reused across the data structure, their correctness can be verified once and applied throughout.

Despite the modular design, verifying the correctness of the entire data structure remains challenging due to low-level behaviors such as distributing operations across rows and panes and aggregating their results. To bridge this mismatch in abstraction, we introduce a layered verification framework with two models: a **concrete model** that closely reflects the modular structure of the P4 implementation; an **abstract model** that captures the high-level logical behavior of the data structure. We prove that the implementation refines the concrete model, and that the concrete model refines the abstract model. This layered approach allows correctness properties to be stated and proved at the abstract level and preserved in the implementation via refinement. The result is a structured and tractable verification process that ensures end-to-end correctness.

As a case study, we apply our synthesis and verification frameworks to a sliding-window Bloom filter on Intel Tofino (Sections 3.2, 3.4). We demonstrate how it can

be modularly implemented, formally modeled, and verified to uphold its no-false-negative guarantee. This illustrates how the synthesis and verification frameworks enable generating a library of reliable and efficient approximate data structures for the programmable data plane.

3.1 Modular Synthesis Framework

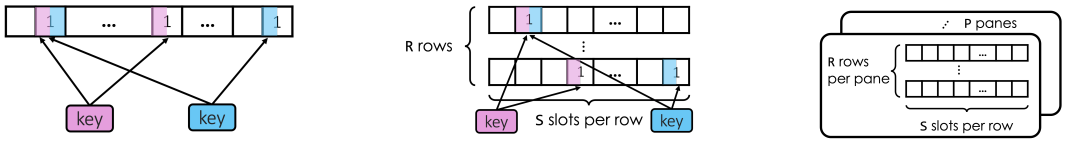
This section presents our **modular synthesis** principles, which transforms general data structure designs into implementations compatible with the data plane. While techniques such as sharding and rotation have been explored in prior works [10, 42], our key contribution is the systematic organization of them into a modular structure grounded in the hardware constraints.

This framework is not tied to any particular data structure. Instead, it defines reusable building blocks—**rows** and **panes**—and organizes their behavior across three well-defined phases: preprocessing, state operations, and postprocessing. This modular design improves implementation clarity, enables parameterized code generation (Section 3.2), and supports semi-automated verification (Section 3.4.1).

3.1.1 Sharding and Rotation

The synthesis framework organizes each data structure into modular **rows** and **panes**, which are defined once and instantiated as needed. These two abstractions enable the framework to generalize many data structures in the data plane. In this subsection, we describe how **sharding** into rows enables stage-local access and how **rotating** panes over time supports temporal freshness.

As discussed in Section 1.3.2, the programmable data plane imposes strict constraints on state access. For instance, the Intel Tofino architecture partitions memory across 10 to 20 pipeline stages. A runtime state object—conceptualized as a register



(a) Original (b) Sharding (c) Rotation

Figure 3.1: Turning a Bloom filter into a sliding-window Bloom filter.

extern—must reside entirely within a single stage. It cannot span stages and may be accessed only once per packet, at the stage that hosts it, via either read/write operations or associated register actions.

These constraints make it difficult to implement traditional data structures, which assume free, repeated access to a contiguous memory space. To adapt to this setting, a common strategy is to **shard** the data structure into smaller segments and distribute them across stages. These segments, called **rows**, enable line-rate performance while leveraging distributed memory.

Many approximate data structures naturally lend themselves to this decomposition. Consider the Bloom filter, a classic approximate set membership structure with two methods: add and query (Figure 3.1a). It uses multiple hash functions to map each element to several slots in a bit array. The add method performs **write** operations to set the corresponding bits; the query method performs **read** operations to check those bits, returning "probably yes" if all are set, and "definitely no" otherwise. Crucially, the Bloom filter guarantee no false negatives: it never misses an element that was previously added.

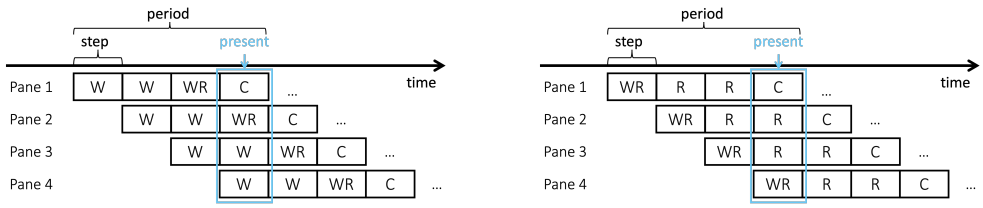
However, this access pattern is incompatible with the data plane's access model. Repeated memory access is disallowed, and allocating the entire bit array to a single register wastes memory in other stages. By sharding the Bloom filter into multiple rows—each with a distinct hash function—we map each element to one slot per row, aligning with pipeline constraints while preserving the no-false-negative guarantee (Figure 3.1b).

In addition to memory layout, temporal freshness presents a second challenge. Without periodic cleanup, memory will be quickly saturated by the high-rate network traffic. Moreover, many applications—such as network telemetry and anomaly detection—focus primarily on recent traffic behaviors. This demands mechanisms to evict stale data.

Conventional approaches maintain an exact sliding time window by associating a timestamp with each entry and deleting stale ones as a new packet arrives. However, such deletions are infeasible in data planes due to strict limits on per-packet computation. In data structures like Bloom filters, deletion is fundamentally unsupported: multiple elements may share the same bits across time, making it impossible to remove one without affecting others.

To address these limitations, we adopt a hardware-compliant cleanup strategy: **rotation**, where two copies of the data structure alternate roles—one active, the other cleaned in the background. This two-pane design, however, introduces a transient dip in coverage after each rotation, as the newly activated pane starts empty. To overcome this, we generalize the design to use $N > 2$ **panes**, with each pane representing a segment of a sliding window.

As shown in Figure 3.2a, under a rotation step of size T, each pane cycles through three phases: $(N-2)T$ of writing, T of reading and writing, and T of cleaning. This multi-pane design maintains continuity: the effective window spans $[(N-2)T, (N-1)T)$, sliding forward with time.



(a) Overlapping panes (b) Disjoint panes

Figure 3.2: Two rotation schemes on a 4-pane data structure.

This overlapping pane scheme raises a key optimization question: can panes keep data from disjoint time segments to eliminate redundancy and improve memory efficiency? For stateful logic that relies on continuity—such as TCP state machines or out-of-order packet counters—this is infeasible, as fragmented history breaks the underlying state machine. However, for state that supports temporal aggregation, the answer is yes.

The Bloom filter illustrates this alternative pane scheme well. Since set membership within a time window only requires that an element be present in any of its subintervals, results from disjoint panes can be safely combined via disjunction. This insight enables a hardware-efficient **sliding-window Bloom filter** (Figure 3.1c). In this construction, the add method writes only to the current writing pane, while the query method reads all $(N - 1)$ active panes and returns their bitwise disjunction (Figure 3.2b). This design preserves the Bloom filter's **no-false-negative** guarantee: if an element was added during the effective sliding window, it is guaranteed to be reported as present.

Cleaning a pane involves resetting all its entries. A simple strategy resets one slot per row in the cleaning pane on each packet arrival. To ensure full cleanup within a rotation step, this approach relies on a sufficiently dense traffic stream, as formalized in Section 3.1.3.

While the preceding discussion has assumed a dedicated cleaning pane, an alternative strategy is to eliminate that entirely by attaching a timestamp to each slot. On every read or write, the stored timestamp is compared against the current time to determine whether the entry is stale. This approach removes the need for a separate cleaning phase but requires additional bits per slot. It is hence most effective when the number of panes is small and the slot size is relatively large. For example, in a 2-pane hash table with 32-bit keys and 64-bit values, adding a 32-bit timestamp may be more memory-efficient than allocating a full third pane for cleaning.

Together, sharding and rotation allow the synthesis framework to convert abstract data structures into modular, stage-aware, and time-aware P4 implementations.

3.1.2 Preprocessing and Postprocessing

To coordinate the behavior of rows and panes, the framework inserts a preprocessing and postprocessing pipeline around the core state operations.

- **Preprocessing** computes per-row parameters such as pane operations and row indexes.
- **Postprocessing** consolidates the results from all rows and panes into a final output.

This structured control flow enables a clear separation of logic and allows the row and pane instances to exhibit different behavior based on the parameters.

During preprocessing, pane rotation is managed by a circular timer that resets after each full rotation period. A naïve implementation extracts a bit from the hardware timestamp and rotates panes whenever that bit flops from 1 to 0 [1]. For example, using the 9^{th} bit of a nanosecond-scale timestamp (0-based indexing) yields a ~ 1 μs rotation step, while the 29^{th} bit corresponds to ~ 1 s. However, this approach restricts timing granularity to powers of two.

To support more flexible rotation intervals, we propose a generalized design where rotation is triggered by counting bit flops. By setting an arbitrary threshold on the number of flops at the **timer bit** before rotation, we can achieve a broader range of timing options. For instance, combining flops at the 29^{th} bit with an 8-bit counter yields rotation periods from 1 to 256 seconds, in 1-second increments—offering both precision and range.

[1] We use the term **flop** to refer to a bit change from 1 to 0, and **flip** for a change from 0 to 1.

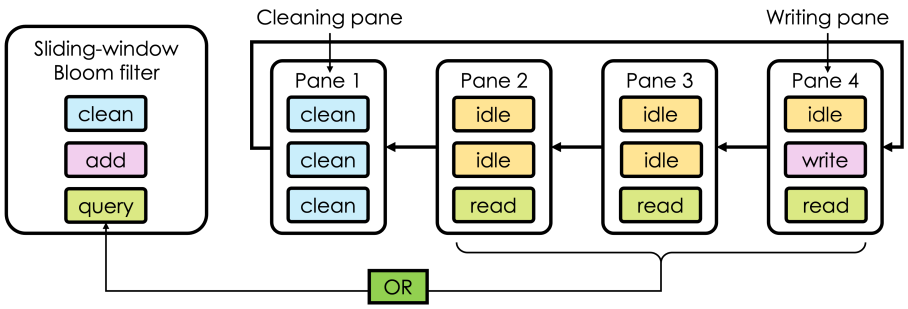


Figure 3.3: Operations dispatch in a 4-pane sliding-window Bloom filter.

Based on the timer value, together with the method invoked on the data structure, the rotation mechanism assigns a specific operation—write, read, clean, or idle—to each pane according to its role in the rotation cycle. These operations are dispatched to the corresponding pane instances and propagated uniformly to all rows within each pane. Figure 3.3 illustrates this process for a 4-pane SBF.

Each pane instance maintains one or more row instances, each operating at a particular slot index. For most operations, hash-based indexing ensures uniform distribution and minimizes contention. For cleaning, incremental indexing is used to guarantee deterministic coverage of all slots. These indexes are computed during preprocessing and dispatched alongside the operations.

Together, these preprocessing steps produce for every row a set of parameters—the operation and the index—that govern its behavior on state. After all pane and row operations are executed at given indexes, the postprocessing phase aggregates the intermediate results to produce a final output. Depending on the data structure, this may involve computing a disjunction, summing values, or selecting a representative result. This final step consolidates partial results across rows and panes, completing a method call on the data structure.

3.1.3 Dense Flow

Because all control logic in our synthesized data structures is packet-driven, time-sensitive behaviors, such as cleaning and rotation, depend on a sufficiently dense flow of packets. This subsection analyzes the traffic requirements to ensure correctness.

First, cleaning is performed incrementally: the cleaning pane resets one slot per row on each incoming packet. If a row contains S slots, at least S packets must arrive during each rotation step T (in ns) to complete a full cleanup. Otherwise, stale entries may persist and compromise the assumptions of data freshness.

Second, pane rotation is triggered by observing bit flops in the hardware timestamp, which requires also seeing bit flips to reset the last seen bit. If the timer bit is set to the b-th bit, a flip-flop occurs every 2^b ns. To capture all flops, at least one packet must arrive within that interval; otherwise, the rotation timer may lag behind.

Together, these constraints impose a minimum packet arrival rate of:

$$\max\left(\frac{s}{T}, \frac{1}{2^b}\right) \quad \text{packets per ns.}$$

When traffic falls below this threshold, a configurable packet generator—a built-in component of the Intel Tofino—can inject supplemental packets at a controlled rate. These packets can be engineered to trigger only the clean method of the data structure to avoid any side effects.

Together, these synthesis principles—rows and panes, proprocessing and post-processing, dense flows—allow designers to transform abstract data structures into modular implementations tailored to programmable data planes.

3.2 Synthesizing a Sliding-Window Bloom Filter

After exploring the general synthesis principles in Section 3.1, we demonstrate the automation capabilities of our synthesis framework with a concrete example: a sliding-window Bloom filter. This case study shows how data-structure-specific processing logic, together with a handful of parameters, can be automatically synthesized into deployable P4 code.

Each data structure in our framework is defined by a reusable template, parameterized by a set of design choices. Only the data-structure-specific logic needs to be provided in the template—namely, the row state operations and the final aggregation behavior. While different data structures may require different parameters, most share the following common ones:

- the number of panes P,
- the number of rows per pane R,
- the number of slots per row S, and
- the rotation step T.

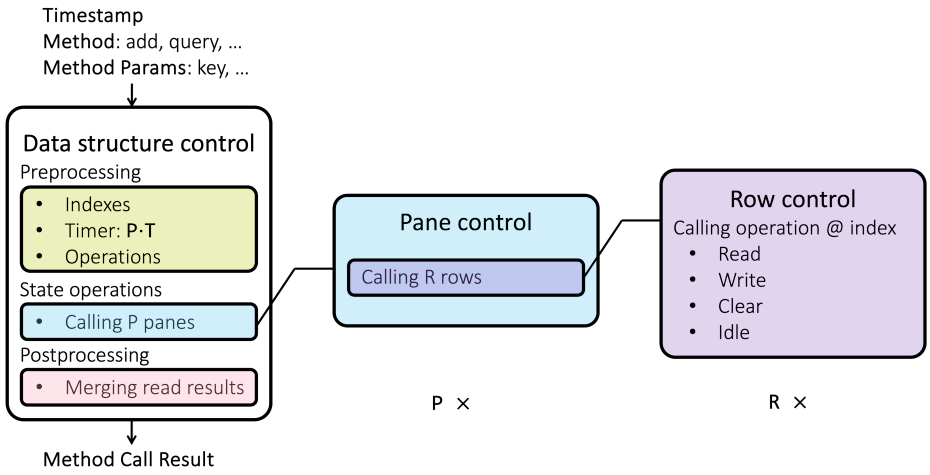


Figure 3.4: Modular synthesis framework.

```
1  control Row(in bit<8> op, in bit<18> index, out bit<8> res)  {
2    /* Register */
3    Register<bit<8>, bit<18>>(262144, 0) reg_row;
4    /* RegisterAction and action for write operation */
5    RegisterAction<bit<8>, bit<18>, bit<8>>(reg_row) ra_write = {
6      void apply(inout bit<8> value, out bit<8> rv) {
7        value = 1; rv = 1;
8      }
9    };
10   action act_write() { res = ra_write.execute(index); }
11   .. // RegisterAction and action for read and clean operations
12   /* Table for calling operations */
13   table tbl_row {
14     key = { op : exact; }
15     actions = { act_write(); act_read(); act_clean(); NoAction(); }
16     const entries = { WRITE : act_write();
17                       READ  : act_read();
18                       CLEAN : act_clean();
19                       IDLE  : NoAction(); }
20   }
21   apply {
22     tbl_row.apply();
23   }
24 }
```

Figure 3.5: The `Row` control in a sliding-window Bloom filter.

As shown in Figure 3.4, given a template and its parameters, the framework emits three P4 control blocks, for rows, panes and the data structure.

We begin with the `Row` control, which defines the per-row behavior of the sliding-window Bloom filter (Figure 3.5). It takes an operation code `op` and a slot index `index` as input and returns the result `res` (line 1). The parameter $S = 2^{18} = 262144$ automatically decides the size of each row's register `reg_row` [2] (line 3). Meanwhile, the Bloom filter template fills the register actions for write, read, and clean operations (line 4-11). Combining the parameter with the template, our synthesis framework can generate the `Row` control, triggering the appropriate register action at the given `index` based on the operation code `op` in the table `tbl_row` (line 12-20).

[2] Although each Bloom filter slot only needs a single bit, the register uses 8-bit values due to Tofino-specific alignment constraints.

```
27 control Pane(inout pane_md_t pane_md)  {
28   Row() row_1;
29   Row() row_2;
30   Row() row_3;
31   apply {
32     row_1.apply(pane_md.op, pane_md.index_1, pane_md.res_1);
33     row_2.apply(pane_md.op, pane_md.index_2, pane_md.res_2);
34     row_3.apply(pane_md.op, pane_md.index_3, pane_md.res_3);
35   }
36 }
```

Figure 3.6: The `Pane` control in a sliding-window Bloom filter.

Similarly, the `Pane` control is parameterized by the number of rows per pane R (Figure 3.6). Here, it instantiates $R = 3$ `Row` instances and applies each with the same operation code `op` and a distinct index, enabling row-specific processing.

At the top level, the `SBF` control provides the client-facing interface for a sliding-window Bloom filter (Figure 3.7). It receives a method code, the method parameter and the timestamp, and returns the final result in `res` (line 37-38). Parameterized by the number of panes $P = 4$ and the rotation period $T = 30\ s$, it applies four panes, using preprocessed operations and indexes (line 90-94). The preprocessing logic—for index computation, timer update, and parameter dispatch—is shared across data structures and reused through code generation (line 83-89).

The rotation step T, deciding the frequency of panes rotation, is realized with a timer that counts 1-to-0 flops at the timer bit inside timestamps. The framework explores two design choices: which bit to monitor (from the 48-bit Tofino timestamp), and the rotation and reset value in a fixed-width counter (e.g., 16-bit, bounded by 2^{16}). For $T = 30\ s$, the framework selects the 21^{st} bit, which flops roughly every 4 ms. It then emits code that increments a counter on each flop, rotates every 7153 flops, and resets at 28611—yielding a rotation step of $\sim 30.002\ s$ with negligible error.

As shown in Figure 3.8, the timer is implemented as a pair of 16-bit registers: `tb` tracks the timer bit and is updated by every packet, and `fc` counts the number of flops and wraps at the period threshold (lines 40–63). This design arises from two

```
37 control SBF(in bit<4> method, in key_t key,
38             in bit<48> timestamp, inout bit<8> res) {
39   ds_md_t ds_md;              // Local metadata struct
40   ..                          // Preprocessing declarations
77   Pane() pane_1;
78   Pane() pane_2;
79   Pane() pane_3;
80   Pane() pane_4;
81   ..                          // Postprocessing declarations
82   apply {
83     /* Preprocessing */
84     act_hash_index_1();
85     act_hash_index_2();
86     act_hash_index_3();       // Compute a hash index per row
87     act_clean_index();        // Increment clean index
88     tbl_timer.apply();        // Update timer
89     tbl_params.apply();       // Dispatch operations & indexes
90     /* State operations */
91     pane_1.apply(ds_md.pane_md_1);
92     pane_2.apply(ds_md.pane_md_2);
93     pane_3.apply(ds_md.pane_md_3);
94     pane_4.apply(ds_md.pane_md_4);
95     /* Postprocessing */
96     tbl_agg_wins.apply();     // Merge rows into final result
97   }
98 }
```

Figure 3.7: The `SBF` control for a sliding-window Bloom filter.

constraints: P4 lacks a modulo operator, and Tofino does not allow extracting a single bit from a timestamp and using it in a register action within the same stage. To work around these limitations, the timer logic is split into two table entries that execute different register actions based on the current packet's timer bit. The framework also synthesizes the dispatch table `tbl_params`, which matches on both the timer and method code to assign the appropriate pane operation and row index (lines 64–74).

This example demonstrates how the synthesis framework bridges designs and code. The automation enabled by this synthesis process forms the basis for a library of efficient data structures. In Chapter 4, we show how this library is integrated into the compiler of a high-level language, enabling users to express approximate stateful computations declaratively while relying on synthesized P4 implementations for efficient hardware execution.

```
/* Timer update */
struct timer_t{bit<16> tb; bit<16> fc;}     // timer bit; flop count
Register<timer_t, bit<1>>(1, {0, 0}) reg_timer;
RegisterAction<timer_t, bit<1>, bit<16>>(reg_timer) ra_flop = {
  void apply(inout timer_t value, out bit<16> rv) {
    if (value.tb == 1) {                        // If timer bit flops,
      if (value.fc == 28611) { value.fc = 0; }     // reset counter,
      else { value.fc = value.fc + 1; } }   // or increment counter;
    value.tb = 0; rv = value.fc;          // always update timer bit.
  }
};
RegisterAction<timer_t, bit<1>, bit<16>>(reg_timer) ra_flip = {
  void apply(inout timer_t value, out bit<16> rv) {
    value.tb = 1; rv = value.fc;          // Always update timer bit.
  }
};
action act_flop() { ds_md.timer = ra_flop.execute(0); }
action act_flip() { ds_md.timer = ra_flip.execute(0); }
table tbl_timer {
  key = { timestamp : ternary; }
  actions = { act_flop(); act_flip(); }
  const entries = { 0 &&& 0x200000 : act_flop();    // Timer bit = 0
                    1 &&& 0x200000 : act_flip(); }  // Timer bit = 1
}
/* Parameter dispatch  */
table tbl_params {
  key = { method : exact; ds_md.timer : range; }
  actions = { act_params_1(); .. }
  const entries = {
    (ADD, 0..7152)       : act_params_1(WRTIE, CLEAN, IDLE, IDLE);
    (ADD, 7153..14305)   : act_params_2(IDLE, WRTIE, CLEAN, IDLE);
    (ADD, 14306..21458)  : act_params_3(IDLE, IDLE, WRTIE, CLEAN);
    (ADD, 21459..28611)  : act_params_4(CLEAN, IDLE, IDLE, WRTIE);
    .. }
}
..
```

Figure 3.8: Preprocessing declarations in the SBF control.

3.3 Layered Verification Framework

Verifying data-plane programs is inherently challenging due to the low-level and domain-specific nature of P4. Synthesizing complex data structures into such programs introduces additional complexity, as algorithms must be adapted to accommodate hardware constraints. To ensure correctness, we adopt a **layered verification** strategy—a well-established paradigm in formal methods. By structuring correctness

proofs across multiple abstraction levels, from a high-level abstract model down to the synthesized P4 code, we achieve modular, maintainable, and rigorous end-to-end guarantees.

3.3.1 Abstraction Layers for Verification

Our verification framework is organized into three abstraction layers (Figure 3.9):

- **Abstract model**: A high-level functional model written in Coq that captures the intended behavior of the data structure in mathematical terms. This model abstracts away implementation details and serves as the high-level description against which correctness is judged. For example, an abstract model of a sliding-window Bloom filter might be modeled as an approximate set with pane expiration.

- **Concrete model**: A low-level functional model written in Coq that mirrors the algorithmic logic of the synthesized P4 code. It incorporates architecture-aware details such as fixed-width registers and circular timers while remaining compact. It serves as the translation of the P4 implementation in a general-purpose language.

- **P4 program**: The data structure written in P4 that represents an imperative implementation for a specific target architecture. This code is parsed into the Coq representation as a P4light AST (Section 2.1.2) to enable formal reasoning.

Each layer has a well-defined role: the abstract model defines the required behavior, the concrete model provides an executable algorithm, and the P4 program embodies the implementation. Verification proceeds by proving **refinement** between adjacent layers: we prove that each layer faithfully implements the more abstract one above it. Hence, we establish two key refinement lemmas in Coq:

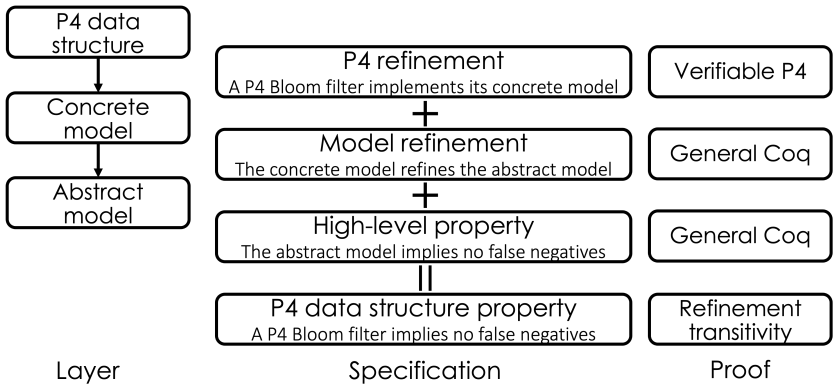


Figure 3.9: Three-layer composition verifies correctness for P4 data structures.

- **Model refinement**: The concrete model refines the abstract model. This involves defining a simulation relation R between internal states of the two models and proving that all method calls preserve R. This proof relies on standard Coq techniques such as induction and equational reasoning and does not involve P4 reasoning.

- **P4 refinement**: The P4 program implements the concrete model. This is established using **Verifiable P4** [47], a verification framework that allows reasoning about P4 programs in Coq (discussed in Section 3.3.2). We show that for any input and P4 runtime state, the P4 execution produces results consistent with the concrete model.

These two lemmas compose: if the P4 program refines the concrete model and the concrete model refines the abstract model, then the P4 program refines the abstract model. Formally:

$$\texttt{P4 program} \sqsubseteq \texttt{concrete model} \sqsubseteq \texttt{abstract model} \Rightarrow \texttt{P4 program} \sqsubseteq \texttt{abstract model}$$

This composition, a direct consequence of refinement transitivity, yields an end-to-end correctness theorem: any property verified on the abstract model also holds for the synthesized P4 implementation.

Our framework embraces a mechanized verification approach: all model definitions, refinement lemmas, properties, and proofs reside in Coq—a software proof assistant that checks proofs in its trusted kernel. This ensures that the final correctness theorem is also machine-checked end to end.

A layered verification framework has two major benefits:

- **Maintainability**: When modifying high-level properties, functional models, implementation strategies or target architecture, one only needs to update the corresponding abstraction level and related proofs. This modularity localizes proof effort.

- **Separation of expertise**: Experts in data structure design can reason about models without understanding P4, while P4 engineers can verify implementation correctness without learning about data structures. Clear model interfaces facilitate collaboration.

This layered approach has been successfully applied in other domains, including SHA-256 cryptographic hash function [2], HMAC deterministic random bit generator [49], forward erasure correction decoding [14], and floating-point arithmetic [3], where refinement from their C implementations is verified using Verified Software Toolchain (VST) [4], a verification framework for reasoning about C. Our work applies similar principles to P4 through Verifiable P4.

3.3.2 Verifiable P4

While the proofs above the concrete model are purely mathematical, the P4 refinement lemma requires reasoning about P4. Chapter 2 introduces our mechanized semantics for formalizing P4 behavior, on which Verifiable P4 [47] is built to reason about correctness. It enables users to specify and verify the behavior of a P4 program in Coq.

The framework combines three key components:

- **Specification**: A rich language for expressing the expected behavior of a P4 program, typically as a precondition and a postcondition it must satisfy.
- **Program logic rules**: A set of inference rules that relate program syntax to logical assertions, allowing stepwise reasoning over program behavior to prove specifications.
- **Automation tactics**: A collection of tactics that automatically apply program logic rules to simplify Coq proof scripts.

In practice, verifying a P4 program involves specifying the behavior of a function or control, then starting from a symbolic state described by precondition, applying program logic rules to step through every syntactic construct (e.g., assignment statements, action invocation) and proving that the resulting symbolic state implies the expected postcondition. The automation tactics proposed in Verifiable P4 relieves much of the manual effort by handling routine steps like variable substitution and logical entailments. Verifiable P4 is **foundational**: its program logic rules are proven sound in Coq with respect to the formal semantics. As a consequence, if a specification can be proven using the program logic, it is guaranteed to hold under all executions conforming to that semantics. For a detailed discussion of Verifiable P4, please refer to [46, 47].

By combining layered refinement with Verifiable P4, we achieve end-to-end verification for synthesized P4 data structures. This ensures that the final implementation is not only efficient but also provably faithful to its high-level properties.

3.4 Verifying a Sliding-Window Bloom Filter

This section demonstrates how to verify the correctness of the sliding-window Bloom filter synthesized in Section 3.2. We follow the layered verification framework introduced in Section 3.3, which involves defining functional models, specifying refinement lemmas, and developing Coq proofs. We focus here on the first two tasks, and for proof scripts and automation tactics, we refer readers to our codebase [45].

We proceed from the bottom up: we first present the concrete functional model (Section 3.4.1) and its refinement to the P4 program (Section 3.4.2), then the abstract model (Section 3.4.3) and its relation to the concrete model (Section 3.4.4), and finally show how high-level correctness properties can be proven and lifted to the P4 implementation (Section 3.4.5).

3.4.1 Concrete Functional Model

Our starting point is a concrete functional model, designed to closely mirror the structure of the synthesized P4 program (Figure 3.10). This model also organizes state modularly: a `row` is an array of booleans (line 10); a `pane` is an array of rows (line 14); and an `sbf` is a record containing an array of panes (`sbf_panes`), a clean index (`sbf_ci`), and a timer (`sbf_timer`) (line 19). Each method corresponds to a sequence of functions in the model. For example, `sbf_add` (line 27-37) adds an element to the data structure by invoking `pane_write`, which writes to a write pane using an array of hash indexes `is` by calling `row_write` to set each boolean to `true`. Query and clean methods are defined similarly. This structural alignment allows each method or operation on a control in P4 to be related directly to a corresponding function in the concrete model, facilitating specification and proof of refinement.

Beyond structural alignment, the functional model benefits from the expressiveness of a general-purpose language. Free from P4's language constraints, the model

```
1  Section Con.
2  (* Parameters & Types *)
3  Context (num_panes num_rows num_slots step tb_pos: Z).
4  Definition ff_length := Z.pow 2 tb_pos.     (* Flip-flop interval *)
5  Definition flop_length := Z.pow 2 (tb_pos+1).    (* Flop interval *)
6  Definition step_flops := step/flop_length.          (* Flops/step *)
7  Definition period_flops := num_panes*step_flops.  (* Flops/period *)
8  Definition listn (T: Type) size := { l: list T | Zlength l = size }.
9  (* Row *)
10 Definition row := listn bool num_slots.
11 Definition row_write (r: row)(i: Z): row := r.[i := true].
12 ..
13 (* Pane *)
14 Definition pane := listn row num_rows.
15 Definition pane_write (p: pane)(is: listn Z num_rows): pane :=
16   map2 row_write p is.
17 ..
18 (* Sliding-window Bloom filter *)
19 Record sbf := mk_sbf
20   {sbf_panes: listn pane num_panes; sbf_ci: Z; sbf_timer: bool*Z}.
21 Definition update_timer (timer: bool*Z)(tstamp: Z): bool*Z :=
22   let '(tb, fc) := timer in
23   let curr_tb := Z.odd (tstamp / ff_length) in
24   if tb && (negb curr_tb) then             (* If timer bit flops, *)
25     (false, Z.modulo (fc+1) period_flops)  (* incr/reset counter; *)
26   else (curr_tb, fc).                  (* always update timer bit. *)
27 Definition sbf_add (f: sbf)(tstamp: Z)(is: listn Z num_rows): sbf :=
28   let '(mk_sbf panes ci timer) := f in
29   (* Preprocessing *)
30   let new_ci := Z.modulo (ci+1) num_slots in  (* Incr clean index *)
31   let timer := update_timer timer tstamp in       (* Update timer *)
32   let cp := snd timer / step_flops in               (* Clean pane *)
33   let wp := Z.modulo (cp-1+num_panes) num_panes in  (* Write pane *)
34   (* State operations *)
35   let panes := panes.[cp := pane_clean panes[cp] ci] in
36   let panes := panes.[wp := pane_write panes[wp] is] in
37   mk_sbf panes new_ci timer.
38 ..
39 End Con.
```

Figure 3.10: Concrete functional model for a sliding-window Bloom filter.

is clearer and more concise. For instance, `row_write` is defined in one line (line 11), compared to a register action and an action needed to implement the same behavior in P4[3]. The timer logic, while staying faithful to the same idea of counting bit flops in the timestamp, is captured by a single `update_timer` function (line 21-26) instead

[3]See code definition at line 4-10 in Figure 3.5.

of spanning multiple register actions and actions in P4[4]. With the modulo operator available in Coq, index wrapping is simplified when updating the clean index, timer state and pane indices, abstracting away low-level arithmetic manipulations and making it easier to reason in Coq.

Another advantage of the functional model is its parameterization. Unlike P4, which hardcodes design parameters, the Coq model is fully parametric. Parameters like `num_panes`, `num_rows`, and `num_slots` are explicitly introduced (line 3). The model defines a dependent type `listn` to define arrays with statically enforced lengths (line 8), ensuring that components such as `row`, `pane`, and `sbf_panes` adhere to their intended sizes. Hence, functions such as `pane_write` are defined generically via list combinators like `map2`—eliminating the need for repeated instantiations and invocations as seen in P4[5].

This modular and parametric structure facilitates verification in two ways. First, model properties can be proved generically for any parameter values. Second, it enables **semi-modular verification** of the P4 code: since the data structure in P4 consists of multiple instances of the same control, once one instance is verified, specifications and proofs for others can be automatically generated with minimal modification. Although these copies need to be specified and verified separately, their structural similarity allows proof scripts to be replayed via shell scripts. The full P4 refinement proof thus scales through a small set of verification templates, aligning with the modularity of the model and implementation.

In summary, the concrete functional model balances algorithmic fidelity with abstraction, serving as an intermediate layer between the P4 implementation and the abstract model.

[4]See code definition at line 43-57 in Figure 3.8.
[5]See code definition in Figure 3.5, 3.6, and 3.7.

3.4.2 P4 Refinement

We now bridge the bottom two layers by proving that the P4 program refines the concrete functional model—that is, the per-packet behavior in P4 matches the behavior defined by the model. Verifiable P4 allows us to specify behavior for a P4 invocable[6] using pre- and postconditions on the program state[7]. Verifying each invocable against its specification contributes to a top-level proof that the P4 data structure implements the concrete model with hierarchical composition.

Figure 3.11 shows the general form of a Verifiable P4 specification. The structure consists of context clauses (PATH, MOD), logical variables (WITH), and the main specification in PRE/POST clauses.

```
WITH x⃗, PATH p MOD m M
  WITH y⃗,
    PRE (ARG P⃗, MEM Q⃗, EXT R⃗)
    POST (EX z⃗, RET v, ARG P⃗′, MEM Q⃗′, EXT R⃗′)
```

Figure 3.11: General specification form in Verifiable P4.

- WITH $\vec{x}$: Universal variables usable in the entire specification (e.g., a path p).
- PATH p: Identifies the scope[7] of the invocable.
- MOD m M: Lists modifiable local variables (m) and runtime state objects[7] (M).
- WITH $\vec{y}$: Universal variables usable in both pre- and postconditions.
- PRE: Precondition describes the program state before the invocation:
  - ARG $\vec{P}$: input arguments.

[6] See definition in Section 2.1.1.
[7] See definition in Section 2.3.1.

  - MEM $\vec{Q}$: Predicates for local variables in the stack frame[7].
  - EXT $\vec{R}$: Predicates for runtime state objects in the persistent state[7].

- POST: Postcondition describes the program state after the invocation:
  - EX $\vec{z}$: Optional existential variables for predicates.
  - ARG_RET $\vec{P'}$ v: output arguments ($\vec{P'}$) and return value (v).
  - MEM $\vec{Q'}$: Predicates for local variables in the updated stack frame.
  - EXT $\vec{R'}$: Predicates for runtime state objects in the updated persistent state.

In our setting, where the goal is to prove that a P4 implementation refines a functional model, we write specifications that relate the arguments, local variables, and runtime state objects in the P4 code to their counterparts in the model. Each specification thus serves as a contract that connects the behavior of a P4 invocable with the corresponding function in the model.

```
Definition Con_sbf_add_spec : func_spec :=
  WITH (* p *),
    PATH p
    MOD None [p]
    WITH (con_sbf : Con.sbf) (key : Val) (tstamp : Z) ,
      PRE
        (ARG [P4Bit 8 ADD;
              val_to_sval key;
              P4Bit 48 tstamp;
              P4Bit 8 1]
        (MEM []
        (EXT [Con.sbf_repr p con_sbf])))
      POST
        (ARG_RET [P4Bit 8 1] ValBaseNull
        (MEM []
        (EXT [Con.sbf_repr p
              (Con.sbf_add con_sbf tstamp (v_to_hashes key))]))).
```

Figure 3.12: Specification for the add method on an `SBF` control instance (e.g. `sbf.apply(ADD, ..)`), matched against the concrete model.

Figure 3.12 shows the specification for the add method on the data structure. It states that, for any concrete state `con_sbf`, inserted P4 value `key`, and timestamp `tstamp`, if a P4 `SBF` control instance at path `p` receives the operation code `ADD` along with the other parameters[8], and its register represents the concrete state `con_sbf` via the predicate `Con.sbf_repr` (line 12), then after invocation, the updated register represents the new concrete state after calling the add function (line 16). The representation predicate `Con.row_repr` ensures correspondence between between the P4 registers and the boolean arrays in the concrete model. The specification essentially defines the preservation of this representation predicate before and after the add method.

Specifications and representation predicates are defined similarly for each method across all controls. Because specifications abstract over internal implementations, the resulting proofs compose hierarchically: changes within a control do not affect dependent proofs, as long as its specification remains unchanged.

To verify that a P4 invocable satisfies its specification, we apply program logic rules with the help of automation tactics. For instance, proving that the `SBF` add method meets its specification requires applying the verified refinement lemmas for each of its pane instances. As shown in Figure 3.13, this results in a sequence of similar tactic calls. Although each pane is explicitly instantiated in the P4 code, the specifications and proof scripts follow a consistent structure, enabling automatic generation. While each refinement lemma must still be applied explicitly, the process remains scalable and semi-automated.

This modular hierarchy allows Verifiable P4 to prove that the P4 data structure refines the concrete model: the program state represents the model state, and this representation is preserved across all method calls on the data structure.

[8]See code definition at lines 37-38 in Figure 3.7.

```
1  Lemma Con_sbf_add_ref :
2    func_sound ge sbf_fd nil Con_sbf_add_spec.
3  Proof.
4    ..
5    step_call verify_pane1.Con_pane_ref.
6    ..
7    step_call verify_pane2.Con_pane_ref.
8    ..
9    step_call verify_pane3.Con_pane_ref.
10   ..
11   step_call verify_pane4.Con_pane_ref.
12   ..
13 Qed.
```

Figure 3.13: P4 refinement lemma and proof for the add method, demonstrating that the P4 definition `sbf_def` conforms to the concrete model.

3.4.3 Abstract Functional Model

The concrete functional model mirrors the P4 implementation closely, but this tight coupling makes it ill-suited for reasoning about high-level properties of the data structure. For example, the classic no-false-negative property of a sliding-window Bloom filter—which guarantees that any element added during the effective sliding window is reported as present—depends on two aspects: the precise window range and access to ground-truth elements. Neither is directly available in the concrete model, which encodes time via a low-level `sbf_timer` pair and hashes elements in boolean arrays.

To address this, we introduce an abstract functional model that discards implementation details in favor of a more mathematical presentation. This model explicitly tracks time, models each pane as a set of added elements, and elevates the dense flow assumption (Section 3.1.3) from an implicit implementation precondition to explicit validity checks. As shown in Figure 3.14, the abstract model state is defined by the Coq record `sbf_core` (line 7), consisting of:

- `sbf_rotate_tstamp`: the timestamp of the next pane rotation, which derives the effective window as $[(\texttt{sbf\_rotate\_tstamp} - (\texttt{num\_panes} - 1) \cdot \texttt{step}),\ \texttt{tstamp})$;
- `sbf_last_tstamp`: the timestamp of the most recently observed packet;

```
Section Abs.
(* Parameters *)
Context {header_t : Set}.
Context (num_panes num_rows num_slots step tb_pos: Z).
Definition ff_length := Z.pow 2 tb_pos.     (* Flip-flop interval *)
(* Sliding-window Bloom filter *)
Record sbf_core := mk_sbf
  { sbf_rotate_tstamp: Z; sbf_last_tstamp: Z;
    sbf_num_cleans: Z; sbf_rw_panes: list (list header_t) }.
Definition sbf = option sbf_core.
Definition sbf_rotate (f: sbf)(tstamp: Z): sbf :=
  match f with
  | None => None
  | Some (mk_sbf rotate_tstamp last_tstamp num_cleans rw_panes) =>
    let timer := (last_tstamp <=? tstamp) &&
                 (tstamp <=? last_tstamp + ff_length) in
    let rotate := tstamp >=? rotate_tstamp in
    let clean := num_cleans >=? num_slots in
    match timer, rotate, clean with
    | false, _, _ => None             (* Timer fails => corrupted *)
    | true, false, _ => Some f              (* No rotation needed *)
    | true, true, false => None    (* Cleaning fails => corrupted *)
    | true, true, true =>                     (* Rotation applied *)
      let rw_panes := rw_panes.[1 .. (num_panes-2)] ++ [[]] in
      Some (mk_sbf (rotate_tstamp + step) last_tstamp 0 rw_panes)
    end.
Definition sbf_add (f: sbf)(tstamp:Z)(header:header_t): sbf :=
  match sbf_rotate f tstamp with      (* Apply checks & rotation *)
  | Some (mk_sbf rotate_tstamp _ num_cleans rw_panes) =>
    let wp := num_panes - 2 in
    let rw_panes := rw_panes.[wp := rw_panes.[wp] ++ [header]] in
    Some (mk_sbf rotate_tstamp tstamp (num_cleans+1) rw_panes)
  | None => None
  end.
..
End Abs.
```

Figure 3.14: Abstract functional model for a sliding-window Bloom filter.

- `sbf_num_cleans`: the number of cleans applied to the current clean pane;
- `sbf_rw_panes`: a list of `(num_panes - 1)` active panes, ordered from oldest to newest, each storing an unbounded set of inserted elements.

We omit the clean pane from this list, as it contains only stale data irrelevant to functional behavior. Tracking the number of clean operations allows us to determine whether the clean pane is empty upon rotation. The abstract state is wrapped in an

`option` type called `sbf` (line 10): `None` represents an invalid status where the dense flow assumption has been violated.

The dense flow assumption, realized by the packet generator, is not enforced in P4 nor embodied by the concrete model.[9] However, that is essential for rotation and cleaning to function correctly, and we would like to check that when reasoning about high-level properties. Therefore, the abstract model enforces the assumption by checking in `sbf_rotate` (line 11-26) the timer and cleaning conditions:

- At least one packet arrives in every flip-flop interval `ff_length` to keep the P4 timer synchronized with time;
- If the packet timestamp exceeds `sbf_rotate_tstamp` (i.e., a rotation is due), the clean pane has received enough clean operations to be emptied.

If either check fails, the filter is invalid and returns `None`. Otherwise, when rotation is triggered, the function drops the oldest one from the active panes, appends an empty one to the end, advances the rotation timestamp, and resets the clean counter. By making the model validity conditions explicit, we can reason about them in model properties.

The `sbf_add` method (line 27-34) begins by invoking `sbf_rotate`. If the filter remains valid, it appends the new element to the newest active pane, updates `sbf_last_tstamp`, increments the clean counter, and returns the updated filter. Other functions, such as `sbf_query` and `sbf_clean`, follow the same pattern: check and rotate the filter, then apply method-specific logic. Specifically, for queries, we simulate the data structure behavior by hashing the stored elements in all active panes—inefficient in practice but convenient for verification.

This abstract model provides a clean, logical view of the sliding-window Bloom filter. Both time and data are modeled explicitly, and the validity is also checked, making high-level properties straightforward to state and prove (Section 3.4.5).

[9] We verify that this assumption is indeed fulfilled by the packet generator in [48].

3.4.4 Model Refinement

We formalize a refinement relation between the concrete and abstract models, ensuring that every observable behavior of the concrete model is accounted for by the abstract model. Analogous to the P4 refinement in Section 3.4.2, where we define representation predicates to relate program state to concrete model state and prove their preservation across method calls, we now define a simulation relation `sbf_sim` that relates concrete state to their abstract counterparts and prove that it is preserved across all methods.

```
Section Sim.
(* Shared parameters *)
Context (num_panes num_rows num_slots step tb_pos: Z).
Definition ff_length := Z.pow 2 tb_pos.     (* Flip-flop interval *)
Definition flop_length := Z.pow 2 (tb_pos+1).    (* Flop interval *)
Definition step_flops := step/flop_length.          (* Flops/step *)
Definition period_flops := num_panes*step_flops.  (* Flops/period *)
(* Concrete model *)
Variable con_sbf : Con.sbf.
Definition con_ci := con_sbf.(sbf_ci).
Definition con_panes := con_sbf.(sbf_panes).
Definition con_timer := con_sbf.(sbf_timer).
(* Derived values from concrete timer *)
Definition timer_tb := fst con_timer.                (* Timer bit *)
Definition timer_fc := snd con_timer.              (* Flop counter *)
Definition cp := timer_fc / step_flops.       (* Clean pane index *)
(* Abstract model *)
Variable abs_sbf : Abs.sbf_core.
Definition abs_num_cleans := abs_sbf.(sbf_num_cleans).
Definition abs_rw_panes := abs_sbf.(sbf_rw_panes).
Definition abs_last_tstamp := abs_sbf.(sbf_last_tstamp).
Definition abs_rotate_tstamp := abs_sbf.(sbf_rotate_tstamp).
...
End Sim.
```

Figure 3.15: Parameters and components used in the simulation relation.

We restrict attention to data structures in valid status by assuming the abstract state is not `None`—that is, we operate under the dense flow assumption. Under this assumption, we relate a concrete state `con_sbf` of type `Con.sbf` to an abstract state `abs_sbf` of type `Abs.sbf_core`. Using the aliases defined in Figure 3.15, the simu-

lation relation `sbf_sim` specifies how each concrete component simulates its abstract counterpart:

- **Clean Pane**: The concrete clean pane `con_panes[cp]` simulates the abstract clean pane by tracking slot-level cleaning progress using a clean index `con_ci`. The abstract model uses `abs_num_cleans` to count the number of cleaned slots, which must be reflected in the concrete model by ensuring that the same number of slots immediately preceding `con_ci`, with wrap-around, contain `false`, indicating they have been cleaned. If `abs_num_cleans` $\geq$ `num_slots`, then the entire concrete clean pane must contain only `false`.

- **Active Panes**: For each index `i` $\in$ `[0..cp-1]++[cp+1..num_panes-1]`, the concrete active pane `con_panes[i]` simulates the abstract active pane at index `(i-cp-1) mod num_panes` in `abs_rw_panes`. For each such pair, the set booleans in the concrete pane must match those resulting from adding all elements of the corresponding abstract pane into a fresh concrete pane.

- **Timer**: The concrete timer `con_timer`, which is a pair of the timer bit `timer_tb` and the flop counter `timer_fc`, simulates the two timestamps in the abstract model, `abs_last_tstamp` and `abs_rotate_tstamp`:

  - **Timer bit value:** `timer_tb` stores the `tb_pos`-th bit of the last packet's timestamp `abs_last_tstamp`:

    $$\texttt{Z.odd}\ (\texttt{abs\_last\_tstamp}/\texttt{ff\_length}) = \texttt{base\_bit}$$

  - **Timer Bit Position:** The next rotation timestamp `abs_rotate_tstamp` aligns with the flop granularity determined by the `tbit_pos`-th bit:

    $$\texttt{flop\_length} \mid \texttt{abs\_rotate\_tstamp}$$

– **Flop counter**: `timer_fc` counts elapsed flops since the start of the current rotation, consisting of the completed steps and current step in progress:

$$\begin{aligned}&(\texttt{cp} \cdot \texttt{step\_flops})\\ +&(\texttt{abs\_last\_tstamp} - (\texttt{abs\_rotate\_tstamp} - \texttt{step}))/\texttt{flop\_length}\\ =&\texttt{timer\_fc}\end{aligned}$$

- **Invariants**: Basic bounds must hold for a valid concrete state [10]:

$$\begin{aligned}\texttt{con\_ci} &\in [0,\ \texttt{num\_slots})\\ \texttt{timer\_tb} &\in \{0,\ 1\}\\ \texttt{timer\_fc} &\in [0,\ \texttt{period\_flops})\end{aligned}$$

Together, these conditions ensure that the concrete state accurately simulates the abstract state. To prove model refinement, we show that the simulation relation `sbf_sim` is preserved by all methods on the data structure. The following lemma (Figure 3.16) shows preservation for `add`; similar lemmas are proved for `query` and `clean`, using the Coq proof assistant.

```
Lemma Sbf_add_model_ref :
  forall con_sbf con_sbf' abs_sbf abs_sbf' tstamp header,
  sbf_sim con_sbf abs_sbf ->
  Abs.sbf_add .. abs_sbf tstamp header = Some abs_sbf' ->
  Con.sbf_add .. con_sbf tstamp (h_to_hashes header) = con_sbf' ->
  sbf_sim con_sbf' abs_sbf'.
```

Figure 3.16: Model refinement lemma for the add method, showing preservation of the simulation relation.

[10] The length bound for `con_panes` is enforced by the array type.

3.4.5 End-to-End Correctness

Layered verification of the P4 program enables us to achieve a rigorous end-to-end correctness guarantee while simplifying the proof structure. By composing the two verified refinement steps—from the P4 implementation to a concrete model, and from that concrete model to an abstract model—we obtain an overall refinement from the P4 program all the way up to a high-level abstract specification. This layered approach allows us to lift high-level behavioral properties proven at the abstract level down to the P4 implementation, rather than having to prove those properties directly on the complex P4 code. We outline this end-to-end refinement composition and demonstrate how it facilitates proving a key property (the no-false-negative guarantee) in a modular way.

Overall refinement.

We compose our P4 refinement and model refinement in our layered verification framework to establish an overall refinement of the P4 implementation against the abstract model. In other words, any behavior observable in the P4 program is guaranteed to be permissible under the abstract model.

As discussed in Section 3.4.2, where we state that the P4 program refines the concrete model with specifications defined in Verifiable P4, here we also define specifications for the abstract models. Figure 3.17 shows such a specification for the `add` method, which mirrors the concrete specification almost exactly, with the key difference being the representation predicate `Abs.sbf_repr`, which connects the P4 registers to the abstract model state.

Defining `Abs.sbf_repr` directly—by linking inserted elements in the abstract state to P4 registers—would be challenging to prove in one step. This is precisely why we advocate for layered verification. By introducing the concrete model as an intermediate layer, we can now define a layered representation relation `Abs.sbf_repr` between

```
Definition Abs_sbf_add_spec : func_spec :=
  WITH (* p *),
    PATH p
    MOD None [p]
    WITH (abs_sbf : Abs.sbf) (header : header_t) (tstamp : Z),
      PRE
        (ARG [header_to_sval header;
              P4Bit 8 ADD;
              P4Bit 48 tstamp;
              P4Bit 8 1]
        (MEM []
        (EXT [Abs.sbf_repr p abs_sbf])))
      POST
        (ARG_RET [P4Bit 8 1] ValBaseNull
        (MEM []
        (EXT [Abs.sbf_repr p (Abs.sbf_add abs_sbf tstamp header)])))
   .
```

Figure 3.17: Specification for the add method, matched against the abstract model.

the abstract state and the P4 state to hold if there exists some concrete state that both simulates the given abstract state and is represented by the P4 state. That is:

$$\texttt{Abs.sbf\_repr p4\_sbf abs\_sbf} := \exists\, \texttt{con\_sbf}.\ \texttt{Con.sbf\_repr p4\_sbf con\_sbf} \wedge \texttt{sbf\_sim con\_sbf abs\_sbf}$$

This layered construction pays off: instead of proving an opaque end-to-end refinement in one leap for the add method, we can reuse the P4 refinemene lemma `Con_sbf_add_ref` (Figure 3.12) and the model refinement lemmas `Sbf_add_model_ref` (Figure 3.16) that we have proved earlier. This structure dramatically simplifies the verification process and enables scalable verification of realistic P4 programs. This layered verification approach lets us structure correctness proofs across abstraction levels, culminating in a guarantee that the P4 program faithfully implements the abstract functional model.

```
Context (num_panes step : Z).
Definition window_length_lo := step * (num_panes - 2).
Lemma Abs_no_false_neg_lemma :
  forall abs_sbf tstamp tstamp' header,
  tstamp <= tstamp' <= tstamp + window_length_lo ->
  valid_by abs_sbf tstamp ->
  Abd.sbf_query
    (Abd.sbf_add abs_sbf tstamp header)
    tstamp' header = Some true.
```

Figure 3.18: No-false-negative property for the abstract model.

High-level properties.

A key advantage of the abstract model is its support for high-level reasoning. In particular, it enables us to formulate and prove correctness properties a clean and simple setting, and then lift those guarantees to the P4 program.

For the sliding-window Bloom filter, we prove a classic no-false-negative property here: any element inserted into the filter remains detectable for the duration of a conservatively defined sliding window. This is formalized in Figure 3.18. The lemma `Abs_no_false_neg_lemma` states that, for any valid abstract state `abs_sbf`, if an element `header` is added at time `tstamp`, then querying the filter for that element at any time `tstamp'` within the window length lower bound returns true. This property is proven directly over the abstract model, with no dependency on P4 details.

To lift this property to the P4 implementation, we apply the same layered strategy used in the overall refinement proof. We define a specification for the P4 `add` method whose postcondition includes the desired property. Specifically, we introduce a predicate `nfn_pred`, which holds if there exists some abstract state `abs_sbf` such that it is represented by the P4 state and it satisfies the no-false-negative condition.

This construction lets us reuse the abstract property lemma `Abs_no_false_neg_lemma` and the overall refinement lemma `Abs_sbf_add_ref`, already established in prior steps, to conclude that the P4 program satisfies the property compositionally. This completes the end-to-end correctness proof.

In summary, by layering the verification efforts, we prove a high-level property of a complex P4 program without having to reason about it directly at the implementation level. The abstract model provides a clean interface for specifying properties; the concrete model enables faithful yet tractable bridging to the P4 code; and the refinements support compositional lifting of correctness results.

Chapter 4

Traffic Control in the Data Plane

Chapter 3 presented synthesis and verification frameworks that systematically construct and verify approximate data structures for the programmable data plane. These frameworks guide the implementation process and correctness reasoning, but they leave open a fundamental question: how can these data structures be effectively used in real control applications?

In traffic control, correct implementations are only one piece of the puzzle. Developers must choose data structures suited to their application's needs, configure parameters such as the number of panes, rows, and slots, and reason about trade-offs between approximation accuracy and hardware resource constraints. These choices directly affect approximation behavior—such as the false positive rate—and are tightly constrained by target architectures—such as per-stage memory.

Making such decisions optimally requires expertise in data structures, approximation theory, P4 programming, and switch architecture. As the library of available data structures grows, manually selecting and configuring them becomes increasingly complex and error-prone.

This chapter presents **Network Approximate Programming (NAP)**, a high-level language designed to lift developers above these low-level concerns. NAP enables

developers to write control programs using **approximate dictionaries**—abstract data types that capture common design patterns across a variety of approximate data structures (Section 4.1). Each dictionary behaves as a generic key-value store, where keys identify flows and values encode per-flow state. It exposes a clean interface with simple methods like `add` and `query` for updating and retrieving per-flow state, and characterizes approximation behavior along two axes: the set of included keys, and the duration over which keys remain in state.

To compile NAP programs to P4, we build on the synthesis framework from Chapter 3. The NAP compiler selects a suitable data structure for each dictionary and explores the parameter space to find configurations that minimize theoretical error while satisfying architectural constraints (Section 4.2). While these constraints restrict implementation flexibility, they also narrow the search space, enabling a brute-force search and greedy placement algorithm to solve the optimization problem efficiently—typically in under a second. Because our synthesis framework generates parameterized data structure templates, the compiler can specialize them for a given NAP program.

We prototype NAP by extending the Lucid language [42] with features for declaring and invoking approximate dictionaries. We implement three reusable dictionary classes—`ExistDict`, `CountDict`, and `FoldDict`—and use them to express a range of traffic control applications (Section 4.3). These NAP programs are 25X–50X shorter than their P4 equivalents. Finally, we validate NAP through a case study of an approximate stateful firewall, demonstrating that the generated P4 program achieves accuracy closely matching the compiler's theoretical error estimates when deployed on the Intel Tofino.

4.1 The NAP Language

Network control applications span a wide spectrum of use cases—from traffic management and network security to in-network monitoring—yet they often share a common structural pattern (Section 1.1). Each application groups traffic into flows, extracts a key from each packet, and associates that key with an evolving notion of per-flow state. This insight forms the foundation for a unifying abstraction.

To see the breadth of this pattern, consider three representative examples:

- In an access network, rate limiters identify flows by client IP and count the number of recent incoming packets to enforce per-user limits.
- In an enterprise network, stateful firewalls track connections between internal and external hosts (e.g., using internal and external IPs), and allow return traffic only if it matches a recently observed outgoing connection.
- In a datacenter, in-network caches monitor how frequently specific keys are requested and store values for the most popular ones based on recent access patterns.

Though their objectives differ, all these applications manipulate key-value state under tight resource constraints.

This leads to a central design question: how can we provide a single abstraction expressive enough to describe these diverse tasks while accounting for approximation behavior?

We answer this by introducing the notion of an **approximate dictionary**. An approximate dictionary offers a key-value interface that captures common functionality across approximate data structures. It supports classic dictionary operations such as `add` and `query`, but its behavior is governed by two key dimensions of approximation:

- **Inclusion direction:** the dictionary may overapproximate (include unintended keys) or underapproximate (omit valid keys) the stored state.
- **Temporal retention:** the dictionary maintains state only over a recent time window, such as a tumbling or sliding window, gradually evicting older entries.

These axes reflect practical trade-offs in control applications. For example, a stateful firewall may prefer overapproximation to ensure that no legitimate traffic is blocked, even at the cost of occasionally allowing unsolicited packets (Section 1.4.2). In contrast, a rate limiter may prefer underapproximation to avoid throttling innocent users. If light flows are overapproximated, they may be mistakenly classified as heavy and unfairly penalized, risking violations of service-level agreements. Underapproximation avoids such false positives: it may miss some violators, but ensures that compliant users are never misclassified. This tradeoff is often acceptable in practice, as most systems can tolerate occasional overuse—especially when enforced alongside global rate limits.

Some applications do not require guarantees in either direction. For example, traffic analytics systems detect anomalies—such as volume spikes or distribution shifts—over coarse-grained flows to flag subsets of traffic for further inspection. These systems act as lightweight pre-filters: as long as relative trends across flows are preserved, committing to a specific error direction is often unnecessary.

In the temporal dimension, most network applications do not require complete history but instead rely on a bounded view of recent activity—as seen in our examples. Since querying arbitrary historical windows is infeasible in the data plane, dictionaries approximate temporal behavior using sliding or tumbling windows.

These insights motivate the core abstractions in the NAP language. Rather than requiring developers to manually select data structures and configure parameters, NAP enables them to express control intent declaratively in terms of approximate dictionaries. In the remainder of this section, we show how this abstraction sim-

plifies the implementation of real-world traffic control programs. We introduce the NAP dictionary interface and demonstrate how its clean design supports concise, approximation-aware control logic.

4.1.1 Approximate Dictionary

The core abstraction in NAP is the approximate dictionary, which generalizes a family of approximate data structures through a unified key-value interface. Each dictionary maps a key—typically derived from packet headers or data plane metadata—-to a value representing flow-level state. A dictionary can be `created`, keys can be `added`, and values can be `queried` using keys. Some applications may use `add_query`, which performs both operations simultaneously.

When a key is added, the dictionary either initializes the value if the key is new, or updates the existing value. When a dictionary is queried, if a value is associated with the key, it is converted into a numeric result and returned; otherwise, nothing is done.

Dictionary class.

NAP prototypes three dictionary classes, each representing a distinct category of state:

- `ExistDict` stores flow keys and answers existence queries.
- `CountDict` counts the number of packets associated each key.
- `FoldDict` allows user-defined logic for initializing, updating, and reading values.

These classes cover a broad range of stateful behaviors in network applications. The example below shows the first half of an approximate stateful firewall implemented in NAP. Such a firewall may use internal and external IP address pairs as

flow keys and record keys of outgoing packets to later drop unmatched incoming packets. This behavior is captured by creating an `ExistDict` dictionary keyed by the IP pair (Figure 4.1).

```
type key = {int<32> int_ip; int<32> ext_ip}
global ExistDict.t<key> seen =
  ExistDict.create(over,
                   within(sec(60), sec(90)),
                   Exist());
```

Figure 4.1: Approximate stateful firewall: dictionary creation.

Each approximate dictionary is created with three parameters: error direction, time window, and value state machine. We begin by discussing the first two, leaving the last one to Section 4.1.2.

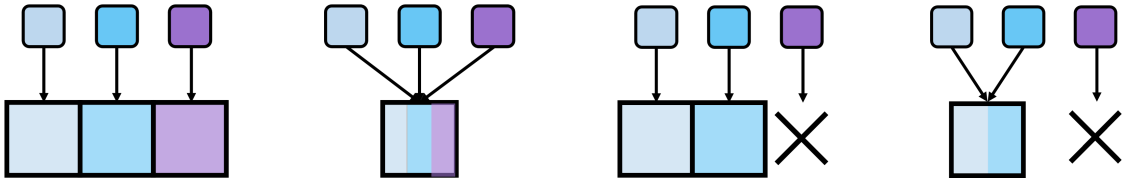
Error direction.

The first parameter characterizes how the dictionary may approximate the mapping between keys and values, capturing the type of inclusion approximation. Ideally, dictionaries store an exact 1-to-1 mapping between inserted keys and their associated values (Figure 4.2a). However, due to resource constraints, approximations may arise:

- **Overapproximation**: An overapproximate dictionary maps multiple keys to a single value, causing the value to reflect combined data (Figure 4.2b). This error direction guarantees that there is always a query result at the cost that it is an "overestimation" of the true value.

- **Underapproximation**: An underapproximate dictionary optionally adds a key: it may be added or it may not (Figure 4.2c). In other words, it ensures that the query result, if existing, is always exact, at the cost of providing no information for the rest of the keys.

- **Approximation**: Both directions of error are permitted (Figure 4.2d).



(a) Exact (b) Overapprox. (c) Underapprox. (d) Approx.

Figure 4.2: Four types of error directions.

In our stateful firewall example, overapproximation is preferable: allowing unsolicited flows through is preferable to mistakenly dropping legitimate return traffic. Hence, the dictionary is configured to overapproximate (Figure 4.1, line 3).

Time window.

The second parameter defines the temporal approximation of dictionaries by specifying how long the state of a packet remains valid. Since the value of stateful information diminishes over time, most network control applications are primarily concerned with recent activity.

In stream processing, two primary windowing constructs are commonly used:

- **Tumbling window**: Time is partitioned into contiguous, disjoint intervals.
- **Sliding window**: Time is tracked using a fixed-length interval that slides forward with the current timestamp.

As explained in Section 3.1.1, the limited computational resources in the data plane make it infeasible to maintain an exact sliding window. Instead, NAP introduces an **approximate sliding window**, which varies in length but always falls within a user-defined range. Let *curr* denote the current time and D the dictionary. NAP supports the following time windows (Figure 4.3):

- `within(lo,hi)`: a sliding window of any duration $t \in [\text{lo}, \text{hi}]$, ensuring that $\forall p \in D,\ 0 \leq curr - p.\text{time} < t$.

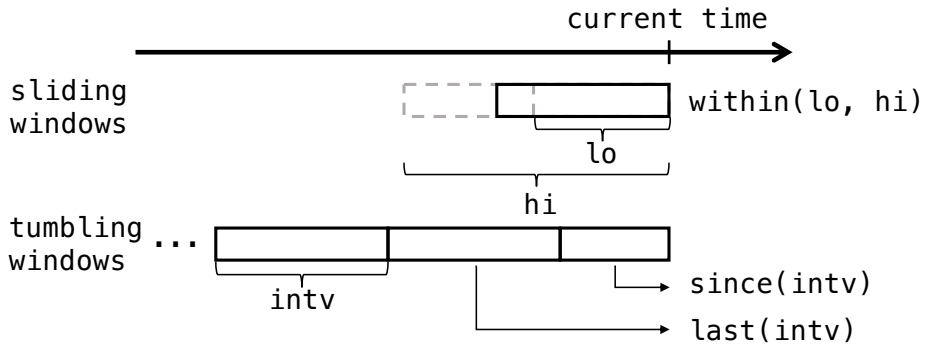


Figure 4.3: Three types of time windows.

- `since(intv)`: the current tumbling window of duration $intv$, so that $\forall p \in D$, $0 \leq curr - p.\text{time} < t$, where $t = curr \bmod intv$.

- `last(intv)`: the most recent completed tumbling window of duration $intv$, so that $\forall p \in D$, $t \leq curr - p.\text{time} < t + intv$, where $t = curr \bmod intv$.

The choice between sliding and tumbling windows depends on the application's temporal requirements. For example, a telemetry task that counts out-of-order TCP packets in each 60-second interval may use `since(60)` to track the current interval or `last(60)` to access the previous one. Tumbling windows avoid overlapping state across time, making them more memory-efficient than sliding windows when state is not mergeable, and enabling better approximation accuracy under the same resource budget. They are well suited for applications where disjoint time intervals suffice.

In contrast, some applications require a continuous view of recent history. In the stateful firewall example, an incoming packet is considered solicited if it matches the key of any outgoing packet seen within the last 60 seconds. This requirement imposes a lower bound on the time window. It is satisfied by a configuration of `within(60, 90)` (Figure 4.1, line 4): all flows observed in the last 60 seconds are guaranteed to be retained, while those older than 90 seconds are excluded. The flexibility of a dynamic window length enables efficient implementation while preserving the 60-second temporal guarantee.

Dictionary methods.

NAP exposes a small set of high-level methods on approximate dictionaries. Figure 4.4 shows the second half of the approximate stateful firewall program, where these methods are invoked within a Lucid packet handler. Outgoing packets are added to the dictionary (lines 10–12); incoming packets query the dictionary to check whether their IP pair was recently observed (lines 14–16). If not, the packet is dropped (line 19).

As we can see from this stateful firewall program, the lightweight and expressive interface of approximate dictionaries allows developers to implement traffic control logic concisely and declaratively.

```
6  handle pkt_in(pkt_t p) {
7    bool s = true;
8    if (p.ingress_port == INT_PORT)
9    then {
10     ExistDict.add(seen,
11                   {ext_ip = p.ip.dst;
12                    int_ip = p.ip.src}); }
13   else {
14     s = ExistDict.query(seen,
15                         {ext_ip = p.ip.src;
16                          int_ip = p.ip.dst}); }
17   if (s)
18   then { p.drop_ctl = NO_DROP; }
19   else { p.drop_ctl = DROP; }
20 }
```

Figure 4.4: Approximate stateful firewall: dictionary methods.

4.1.2 Value State Machine

The third parameter to an approximate dictionary is a value state machine, which specifies how per-flow state evolves in response to packets. This abstraction is motivated by hardware constraints of data planes, which typically allow only a single state access per packet. As a result, all state changes must be expressed as atomic functions over the current packet and the prior state. NAP captures this computation pattern using a structured state machine interface. When a key is added, its value

is either initialized (if absent) or updated (if present). This behavior is uniformly described using three functions:

- an initialization function that sets the state based on the packet data;
- an update function that modifies the state using the existing value and the packet data;
- a read function that extracts a numeric result from the internal state.

This pattern generalizes per-flow state logic and corresponds to the standard `Fold` construct in functional programming [32]. A `Fold` reduces a sequence of elements—here, packets—into a single value by iteratively applying an update function to accumulate state. This abstraction aligns naturally with data-plane programming, where each packet incrementally updates per-flow state.

For example, the state machine for an `ExistDict`—which tracks whether a key has ever been seen—can be expressed as `Fold(init, upd, read)`, where the component functions are defined in Figure 4.5. The state is initialized to `true` on the first packet and left unchanged thereafter, so the read result always returns `true` once the key is added. Since this behavior is shared across all `ExistDict` instances, it is abbreviated as `Exist()` in dictionary creation (Figure 4.1, line 5). A similar fold-based state machine underlies `CountDict`, which initializes a counter to one and increments it with each following packet.

```
1 type state_t = {bool b}
2 fun state_t init(pkt_t p)
3   { return {b = true}; }
4 fun state_t upd(pkt_t p, state_t s)
5   { return s; }
6 fun bool read(state_t s)
7   { return s.b; }
```

Figure 4.5: Value state machine for `ExistDict`.

Predefined state machines support common cases, but more sophisticated logic often requires customization. To support this, NAP provides `FoldDict`, which allows users to customize the value state machine.

Figure 4.6 defines a `FoldDict` for tracking out-of-order (OOO) TCP packets. The state is a pair of 32-bit integers: `fst` stores the last seen TCP sequence number, and `snd` accumulates the OOO count (line 3). The state machine is defined by the `init` (lines 4-5), `upd` (lines 6-9), and `read` functions (lines 10-11), and assembled at creation (lines 12-13). On each packet, the update function records the new sequence number and increments the counter if it is smaller than the previous one. The read function returns the current OOO count. This `FoldDict` allows NAP programs to flag potentially congested flows based on observed reordering.

```
1  type key_t = {int<32> src_ip; int<32> dst_ip;
2                int<16> src_port; int<16> dst_port}
3  type state_t = {int<32> fst; int<32> snd}
4  fun state_t init(pkt_t p)
5    { return {fst = p.tcp.seq_no; snd = 0}; }
6  fun state_t upd(pkt_t p, state_t s)
7    { return {fst = p.tcp.seq_no;
8              snd = s.snd + 1 if s.fst > p.tcp.seq_no
9                              else s.snd}; }
10 fun int<32> read(state_t s)
11   { return s.snd; }
12 global FoldDict.t<key_t> ooo = FoldDict.create
13   (under, since(sec(60)), Fold(init, upd, read));
```

Figure 4.6: `FoldDict` for out-of-order packet detection.

Formally, value state machines must compile to state access externs supported by the target hardware. On architectures like the Intel Tofino, state is manipulated via register actions, which perform a single read-modify-write sequence using the underlying stateful ALU. This directly shapes NAP's design: the syntax of each function must conform to the computational model of the stateful ALU.

The update function, for instance, must return the new register entry in a single `return` statement, computed as a function of the current packet and the existing register entry. Its body must satisfy the following constraints:

- The function may use at most two 32-bit fields from the input packet, and the register entry may store up to two 32-bit values.
- The return statement constructs the new state using either basic expressions or ternary expressions that select between basic expressions based on comparisons.
- A basic expression consists of an ALU-supported operation over at most one register value and at most one packet field[1].
- Comparisons must conform to the ALU-compatible form[2], and each update function may include at most two unique comparisons.

The initialization function follows the same rules, except it operates only on packet fields since there is no prior state. The read function returns a single 32-bit register value extracted from the current register entry.

These syntactic constraints mirror the internal structure of the stateful ALU on Tofino, which receives up to two 32-bit values from the register and two from the PHV, and performs gated computation via its comparison and arithmetic units. The stateful ALU produces an updated register entry and a single 32-bit return value.

Programming register actions in P4 often requires reasoning about how control logic maps onto these low-level ALU constraints. Developers may need to restructure seemingly simple logic—e.g., `if-else` branches—to fit into the stateful ALU's fixed data paths. Even experienced P4 programmers often struggle to ensure their register actions are compilable.

[1]Supported operations include unary and binary arithmetic and bitwise operations such as addition and XOR, applied to constants, register values, and packet fields.

[2]Each comparison must follow the structure `reg_value + pkt_field + const comp_op 0`, where `comp_op` is a supported comparison operator.

NAP alleviates this burden by exposing a carefully chosen subset of the logic supported by register actions. Its value state machine syntax is designed to balance expressiveness and regularity: it supports a broad class of practical use cases while enabling predictable compilation. The NAP compiler statically checks user-defined state machines against the above constraints. If a state machine is valid, it is guaranteed to compile to the hardware. If not, the compiler reports detailed feedback pinpointing the violation.

While these restrictions rule out some patterns technically supported by the Intel Tofino—such as falling back to the old state by default—they prioritize simplicity over completeness. In practice, the current design has proven expressive enough to support a wide range of applications while significantly reducing the complexity of writing hardware-compliant state logic.

4.2 Compiling to the Data Plane

To compile NAP programs into executable P4 code, the compiler builds on the modular synthesis framework from Chapter 3, which provides parameterized templates for verifiable data structures. The compilation process selects an appropriate data structure for each dictionary (Section 4.2.1), determines the corresponding time window implementation (Section 4.2.2), and assigns parameter values to minimize theoretical error while satisfying hardware constraints (Section 4.2.3). The final output is a concrete P4 program that runs on the Intel Tofino.

4.2.1 Selecting the Data Structure

Given an approximate dictionary in NAP, the compiler selects an appropriate data structure based on the dictionary class—`ExistDict`, `CountDict`, or `FoldDict`—and the specified error direction—`exact`, `over`, `under`, or `approx` (Table 4.1). Chapter 3

describes how data structures are generally sharded into rows; here, we show how different multi-row layouts are applied to the selected data structure.

Exact dictionary.

Exact dictionaries are implemented using exact tables indexed directly by the key. To support multi-row layouts, the key is sliced into a row index and a slot index, enabling a partitioned table design across rows. Because this representation maintains an exact mapping between keys and values, it requires $|value| \cdot 2^{|key|}$ bits of memory, where $|key|$ and $|value|$ are the bit-widths of the key and value, respectively. If the estimated memory exceeds the target capacity, the compiler raises an error and recommends adding approximation.

Overapproximate dictionary.

Overapproximate dictionaries ensure that every key maps to some value in memory. To achieve this, the key is hashed into a smaller index space within shared registers, introducing the possibility of collisions. Two types of data structures are used in this case: hash tables and sketches. Although both use multi-row layouts, they differ in how keys are mapped and how values are aggregated.

In a multi-row hash table, the key is hashed twice: first to select a row, then to compute the slot index within that row. Each key maps to a single row and a single

Error direction	`ExistDict`	`CountDict`	`FoldDict`
`exact`	exact table		
`over`	Bloom filter	count-min sketch	hash table
`under`	hash table w. full fingerprints		
`approx`	**Bloom filter**, all of the rest above, hash table w. partial fingerprints	**count-min sketch**, all of the rest above, hash table w. partial fingerprints	All of above, **hash table w. partial fingerprints**

Table 4.1: Data structure choices.

slot, where the corresponding value state machine is stored. This data structure is the choice for an overapproximate `FoldDict`, since individual values are not easily mergeable across multiple slots.

In contrast, sketches hash each key into a slot in every row. Each row uses a different hash function, and query results are aggregated across rows to mitigate the effects of collisions. This approach is applicable when the value can be combined across rows to improve accuracy. For instance, a Bloom filter is used for `ExistDict` under overapproximation: each row sets a bit to `true` for the hashed slot during insertion, and membership queries perform a logical AND across all R rows. Due to collisions, Bloom filters may report false positives. Similarly, `CountDict` under overapproximation uses a count-min sketch, which maintains an array of counters per row. Each key incrementally updates one slot in each row, and queries return the minimum counter value to reduce overestimation.

Underapproximate dictionary

Underapproximate dictionaries guarantee that only one key maps to a given value. This is achieved with a hash table that stores full fingerprints: the entire key is stored alongside the value to disambiguate collisions. Even if multiple keys hash to the same index, the original key can be checked during insertions and queries to ensure correctness. This data structure may introduce false negatives by missing keys if the intended slot is already occupied.

Approximate dictionary.

Generally approximate dictionaries permit both false positives and false negatives, allowing more flexible trade-offs. Any of the above data structures may be used. Additionally, when using hash tables with fingerprints, the compiler relaxes the re-

quirement to store full keys. Partial fingerprints obtained by hashing the key also suffice.

While it is possible to avoid hashing by slicing the key into a row index, slot index, and fingerprint, this method assumes a uniform key distribution to minimize collisions. In practice, key values such as IP addresses and ports often exhibit hierarchical or skewed distributions, resulting in uneven row utilization. To ensure balanced multi-row indexing under realistic traffic patterns, the NAP compiler relies on hash functions for both row and slot selection.

Because data structures with different error directions are not directly comparable in terms of accuracy or resource usage, NAP applies sensible defaults: Bloom filters for `ExistDict`, count-min sketches for `CountDict`, and hash tables with partial fingerprints for `FoldDict`.

Final remarks.

For overapproximate `FoldDict`, the randomization inherent in hashing may produce aggregate values that are difficult to interpret, especially when keys have structure (e.g., IP addresses). In such cases, developers can define an exact dictionary using selected key bits—for instance, truncating IP addresses to /16 prefixes—to align aggregates with semantically meaningful groups. Since this optimization relies on traffic patterns and domain knowledge, NAP leaves it to users: they may slice the key and set the error direction to `exact` in NAP to evaluate feasibility in the data plane. This approach demonstrates that NAP balances the high-level control intents with user-guided refinement.

4.2.2 Time Window Implementation

NAP supports time windows to restrict which key-value pairs in a dictionary are considered valid. Chapter 3 introduced rotating panes as a modular mechanism for

maintaining temporal freshness, providing two multi-pane layouts—overlapping and disjoint—and two cleanup strategies—dedicated cleaning panes and per-slot timestamps.

This section explains how the data structure type determines the appropriate multi-pane layout, and how the combination of time window and cleanup strategy decides the required number of panes P.

Multi-pane layouts selection.

The choice between overlapping and disjoint layouts is driven by the data structure choice. Sketches with temporally aggregatable values naturally adopt overlapping layouts, which merge panes to form the effective window. In contrast, exact tables and hash tables typically require disjoint panes, as their value state machines do not support merging across time.

Approximate sliding window.

An approximate sliding window, `within(lo,hi)`, retains entries seen for some dynamic recent time interval bounded by `lo` and `hi`. Regardless of the chosen multi-pane layout, the required number of panes P depends solely on the cleanup strategy:

- **Dedicated cleaning pane**: To ensure the reading pane(s) always span an interval of length between `lo` and `hi`, each rotation step must not exceed (`hi` $-$ `lo`), resulting in:
$$P \geq \left\lceil \frac{\texttt{hi}}{\texttt{hi} - \texttt{lo}} \right\rceil + 1$$
- **Per-slot timestamps**: No dedicated cleaning pane is required, yielding:
$$P \geq \left\lceil \frac{\texttt{hi}}{\texttt{hi} - \texttt{lo}} \right\rceil$$

Tumbling window.

Tumbling windows segment time into consecutive intervals. The variant `since(intv)` retains entries from the current interval, while `last(intv)` captures the immediately preceding one. The required number of panes again depends on the cleanup strategy:

- **Dedicated cleaning pane**: `since(intv)` uses 2 panes (writing and cleaning); `last(intv)` uses 3 (writing, reading, and cleaning).
- **Per-slot timestamps**: `since(intv)` requires only 1 pane; `last(intv)` requires 2, as the cleaning pane is eliminated.

Cleanup strategy selection.

Given the time window, the compiler chooses between dedicated cleaning panes and per-slot timestamps by comparing memory overheads based on P and the per-flow state size (Table 4.2). Per-slot timestamps are typically preferred when P is small and slot sizes are large, as is often the case for hash tables and tumbling windows.

4.2.3 Sizing the Data Structure

Approximate data structures inherently introduce errors, with their parameters directly determining both the error magnitude and resource usage. The synthesis framework introduced in Chapter 3 generates modular P4 templates—rows, panes, and data structures—while leaving key parameters, such as the number of panes (P), rows per

Time window	Clean pane	Per-slot timestamp
`within(lo,hi)`	$\geq \lceil \frac{hi}{hi-lo} \rceil + 1$	$\geq \lceil \frac{hi}{hi-lo} \rceil$
`since(intv)`	$= 2$	$= 1$
`last(intv)`	$= 3$	$= 2$

Table 4.2: Pane count P required under time windows and cleanup strategies.

Data structures	What to minimize	Theoretical error
Bloom filter	False positive rate	min. $1-(1-(1-(1-\frac{1}{S})^{\frac{M}{P}})^R)^P$
Count-min sketch	Upper bound of expected error	min. $e^{-R}+\frac{e}{S}$
Hash table w. full fingerprint	Key misses	min. $M-SR$
Hash table w. partial fingerprint	Key collision probability	$f=S(1-(1-\frac{1}{S})^{\frac{M}{R}})$ $k=\frac{M}{fR}$ $c=(1-\frac{1}{2^F})^{k-1}$ min. $Rf(k-c)/M$

Table 4.3: Theoretical errors of data structures.

pane (R), slots per row (S), and fingerprint length (F), configurable. The NAP compiler selects appropriate parameter values to generate a concrete P4 program, aiming to minimize theoretical error while meeting data plane resource constraints—yielding a classic constrained optimization problem.

Optimization objective.

For each approximate data structure, the compiler aims to minimize its theoretical error. This error is a function depending on parameters like P, R, S, and F, along with input traffic characteristics such as the average number of distinct keys within the target time window (M). Table 4.3 summarizes error formulas for each supported data structure. When multiple approximate dictionaries are defined within a NAP program, the compiler combines their errors into a unified optimization objective.

Parameter constraints. At first glance, the parameter space may seem large. For example, consider the parameter tuple (P, R, S) of a Bloom filter. Given a fixed memory budget n, there are $O(n(\log n)^2)$ distinct tuples whose product $P \times R \times S$ fits within n. However, the actual space is far smaller thanks to constraints imposed by the data plane. These constraints significantly simplify the optimization process by allowing the compiler to pre-prune infeasible tuples:

- **Time constraints**: The number of panes P is restricted by the selected time window and cleanup strategy (see Table 4.2 and Section 4.2.2).

- **Memory constraints**: The total number of rows $P \times R$ must not exceed the available registers, typically limited by the number of stateful ALUs. For instance, a 10-stage pipeline with 5 stateful ALUs per stage imposes the constraint $P \times R \leq 50$.

- **Computational constraints**: The number of slots S must be a power of two to avoid expensive modulo operations during hashing. For a 1MB register, S may take at most 22 values among $2^1, \ldots, 2^{22}$. Furthermore, because register memory is allocated in fixed-size blocks, smaller values of S that consume the same number of blocks as larger ones can be pruned without loss of generality.

- **Architectural constraints**: Additional limitations arise from hardware-specific factors, such as the number of available hash units, supported register widths, and the total number of pipeline stages.

Collectively, these constraints reduce the parameter search space from millions of combinations to just hundreds of valid tuples, making exhaustive enumeration tractable.

Greedy placement algorithm. Due to the manageable size of the parameter space, the NAP compiler employs a straightforward greedy placement algorithm:

- Enumerate all parameter tuples satisfying the constraints.

- Compute theoretical error for each tuple using formulas.

- Rank the tuples in ascending order of error.

- Convert the NAP program into a dependency graph of data structures and imperative components, respecting topological order.

- For each ranked tuple, simulate placement of the dependency graph onto the hardware pipeline following a topological order, exhausting available resources stage by stage.
- Select the first parameter tuple that fits entirely within available hardware resources, and output the corresponding P4 program along with its theoretical error.

This greedy heuristic is guaranteed optimal for programs where approximate dictionaries form a linear dependency chain—a common case due to constraints in the data plane.

Discussion.

The NAP compiler avoids the complexity of global optimization techniques such as integer linear programming. Instead, it relies on structural regularities and hardware-imposed constraints to guide the search. Users benefit from not needing to define custom cost functions or hardware mappings: each supported data structure comes with a built-in, theoretically grounded error metric. The expected error, a byproduct of compilation, is reported alongside the generated program, enabling users to reason about the trade-offs introduced by approximation. The resulting system is lightweight, automated, and practical.

4.3 Evaluation

We evaluate NAP along three key dimensions: expressiveness of the language, performance of the compiler, and end-to-end correctness and efficiency of the generated programs. We first assess expressiveness by implementing a diverse set of applications in NAP and comparing them to their compiled P4 outputs (Section 4.3.1). Next, we

analyze compilation time and optimization behavior across applications and benchmarks (Section 4.3.2). Finally, we demonstrate functional correctness and hardware efficiency by running the stateful firewall on the Intel Tofino and evaluating its performance on a campus traffic trace (Section 4.3.3).

4.3.1 Language Design

To evaluate expressiveness and ease of programming, we implement a diverse set of nine example applications in NAP, including network telemetry, network monitoring, and network control applications.[3] Each of them utilizes one or more approximate dictionaries. All of these practical applications can be expressed within 30 lines of code (LoC). In comparison to their compiled P4 counterparts, NAP substantially reduces programming effort, achieving a reduction of 25X to 50X in LoC (Table 4.4).

It is worth noting that NAP generates highly modularized P4 programs, potentially resulting in an even greater reduction in LoC when compared to hand-written P4 programs. For reference, Lucid [42] generates a 2267-LoC P4 program for an approximate stateful firewall, significantly longer than our 555-LoC P4 output.

4.3.2 Compiler Performance

In measuring the compiler effiency, we can see that all example programs compile to hardware-targeted P4 code in under one second, with single-dictionary applications completing in less than 0.01 seconds.

To stress-test the compiler, we introduce synthetic benchmarks that incrementally increase the number of overapproximate `ExistDict`s, each of which is compiled into a Bloom filter. As expected, compilation time increases sharply with the number of dictionaries: each additional Bloom filter increases the parameter space space multiplicatively, causing compilation time to scale by roughly a factor of 100 (Table 4.4).

[3]The full code for these applications is provided in Appendix A.

Application	LoC		Compile time (s)	Configurations	
	NAP	P4		Total	Optimal rank
Single dictionary					
Stateful firewall [5]	16	555	0.0055	525	87 (16.6%)
DNS amplification mitigation [5]	16	582	0.0056	525	87 (16.6%)
FTP monitoring [5]	18	798	0.0035	64	32 (50.0%)
Heavy hitter detection [5]	11	595	0.0049	126	3 (2.4%)
Traffic rate measurement by IP/8	12	466	0.0040	14	1 (0.8%)
TCP out-of-order monitoring [32]	23	559	0.0043	66	22 (33.3%)
Heterogeneous dictionaries					
TCP superspreader detection [5]	24	842	0.0130	3274	730 (22.3%)
TCP SYN flood detection [5]	24	842	0.0130	3274	730 (22.3%)
NetCache [25]	24	802	0.0394	9726	5049 (51.9%)
Bloom filter benchmarks					
1-Bloom filter	17	555	0.0055	525	87 (16.6%)
2-Bloom filter	30	960	0.1743	88053	4053 (4.6%)
3-Bloom filter	43	1289	25.94	6648.2K	367.4K (5.5%)
4-Bloom filter	56	1618	2278.87	261.0M	24.5M (9.4%)

Table 4.4: Network applications and benchmarks.

This experiment highlights two key points. First, NAP supports efficient compilation for real-word applications. Second, the exponential growth in parameter configurations exposes the limits of scaling: while small applications explore hundreds of configurations, the 4-Bloom filter benchmark must process over 261 million.

A closer breakdown reveals that the majority of time is spent generating and evaluating parameter tuples. For example, compiling the 4-Bloom filter program takes an average of 2278.87 seconds, with 2020.27 seconds used to generate all configurations and 255.23 seconds used to fit the top 9.4% onto hardware. These results suggest concrete opportunities for future improvement: further reducing the size of the parameter space through smarter pruning, or accelerating the fitting process by better guiding the search.

4.3.3 Stateful Firewall Case Study

To evaluate the practical viability of NAP programs, we deploy our approximate stateful firewall program onto the Intel Tofino. The program is first compiled to P4,

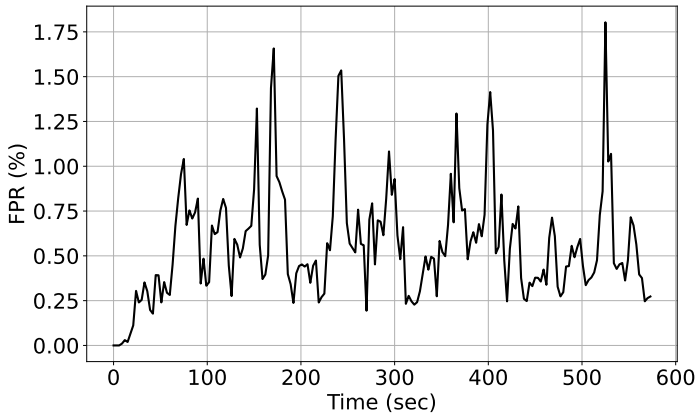


Figure 4.7: False positive rate fluctuations over ten minutes.

utilizing a 4-pane and 3-row Bloom filter with a theoretical error of 0.329%. After installing the P4 program onto the data plane, we replay a 10-minute anonymized trace captured at the Princeton University campus border starting at 2 p.m. EST on August 19, 2020. The trace comprises around 106 million packets at a rate of 185,000 packets per second.

Compared to the ground truth with an exact 60-second sliding window, our approximate stateful firewall exhibits a 0.509% false positive rate. This error is slightly higher than the theoretical error, possibly due to the temporal approximation. The NAP program adopts a $[60, 90]$-second approximate sliding window, which includes keys that the ground truth does not have in its exact window.

Figure 4.7 presents the variation of the false positive rate over time. It starts off at 0% since the initially empty Bloom filter does not allow any unsolicited incoming traffic. As it fluctuates between 0.25% and 2%, an interesting pattern emerges, characterized by spikes occurring about every 30 seconds, corresponding to the 30-second pane length. The rate gradually increases as the writing pane fills up and then drops suddenly when the oldest pane is replaced with a cleaned pane.

Resource	Hash Units	ALU Units	SRAM	TCAM
Utilization	27.3%	31.8%	25.6%	7.9%

Table 4.5: Hardware resource utilization on the Intel Tofino.

Table 4.5 displays the resource utilization for the output P4 program on the Intel Tofino. While one might intuitively expect the program to fully utilize the available memory, hash units, and stateful ALUs to minimize errors, the optimal configuration actually consumes only 25.6% of the SRAM. This seemingly underutilized allocation arises from two key factors: first, a significant portion of SRAM is architecturally unavailable for register memory; second, the preprocessing and postprocessing dependencies confine the data structures' placement to the middle pipeline stages.

These results demonstrate that NAP-generated P4 programs can achieve low false positive rates while maintaining bounded resource usage. The modular synthesis framework and parameter search work in tandem to balance performance and hardware constraints—allowing NAP programs to coexist with other applications and workloads in production environments.

4.4 Related Works

Several network programming languages have been developed for programmable data planes, each targeting different abstractions and optimization challenges.

Lucid [42] is an event-driven language designed for low-latency network control on programmable switches. While it offers a concise syntax, it requires users to manually select and define data structures. Its compiler optimizes control flows through static analysis, primarily reducing the number of pipeline stages—an effort orthogonal to NAP's focus on optimizing approximate data structures under memory constraints.

Marple [32] and Sonata [21] are network telemetry languages tailored for performance and scalability. Marple supports query-driven performance monitoring via a new key-value primitive, while Sonata leverages stream processors to offload data-plane work. However, neither system is designed to operate entirely within the programmable data plane. Newton [56] addresses in-network monitoring using approx-

imate data structures to tolerate dynamic traffic conditions, but it lacks a general abstraction for data structures and does not support reactive control. Moreover, all three are limited to offline analysis and do not support packet-by-packet actions in response to queries.

P4All [22] extends P4 with support for parameterized data structures and compiles them via integer linear programming. While it enables greater flexibility by letting users define custom objectives and data structures, this generality results in a vast solution space and prolonged compilation time. In contrast, NAP confines users to predefined approximate dictionaries. This design choice allows the compiler to leverage prior knowledge about resource usage and placement behavior, enabling efficient optimization via exhaustive enumeration and greedy placement.

Domino [41], Chipmunk [20], and Lyra [19] all focus on compiling control algorithms into P4 pipelines while optimizing resource usage, such as minimizing stage count. However, these compilers assume low-level data structures and do not address the challenge of synthesizing and optimizing parameterized approximate data structures as NAP does.

In summary, NAP advances programmable network control by enabling approximate stateful behavior entirely within the data plane, building on the event-driven model pioneered by systems like Lucid. While prior work has focused on control logic, telemetry queries, or custom data structure optimization, NAP introduces high-level approximate dictionaries. This restricted yet expressive abstraction enables modular synthesis of approximate data structures and efficient parameter selection under hardware constraints. By specializing the language and compiler around in-network approximation, NAP achieves practical automation, robust expressiveness, and strong hardware efficiency across a broad class of real-world applications.

Chapter 5

Conclusion

Modern network control applications—including traffic management, security systems, and performance monitoring—demand fine-grained and dynamic responses to evolving traffic patterns. These applications typically share a common structure: they maintain per-flow state over time and use it to make flow-level decisions on how to process packets. This pattern recurs across a variety of settings, from access networks to data centers. Yet, implementing such stateful logic at line rate remains a fundamental challenge, especially given the constraints of traditional switch hardware.

The advent of programmable switches has created new opportunities and complexities for network control. Languages like P4 allow developers to implement custom logic entirely within the data plane, enabling fast and flexible in-network applications. However, the available memory and compute resources remain limited, requiring compact representations of state. Approximate data structures offer a promising solution by trading precision for scalability. When carefully designed, these data structures can support diverse stateful control applications without exceeding hardware limits.

Despite their potential, implementing approximate stateful logic in programmable switches introduces substantial challenges. Developers must navigate trade-offs between accuracy and resource usage, while reasoning within strict hardware

constraints—such as pipeline depth, register layouts, hash unit availability, and stateful ALU capacity. Language-level limitations further complicate development: ambiguities in the P4 specification and the lack of a formal semantics hinder programming and reasoning, particularly in the presence of stateful externs and uninitialized values. Developers lack the abstractions and tools needed to manage these challenges effectively. As a result, building approximate control applications today remains a difficult and error-prone task.

5.1 Summary of Contributions

This thesis presents a comprehensive approach to enabling network control in programmable data planes by integrating formal semantics, data structure synthesis, program verification, domain-specific languages, and resource optimization. It bridges high-level abstractions and hardware-executable implementations by addressing foundational challenges in semantics, data structures, and language abstraction:

- **Formal semantics for P4**: We define a formal semantics for the P4 programming language and the Intel Tofino architecture by faithfully modeling the language specification and architecture-specific behaviors (joint work with Qinshi Wang [46, 47]). Our semantics captures P4's two-phase evaluation model, characterizes stateful externs, and accounts for uninitialized values. Fully mechanized in Coq, it serves as a foundation for verifying P4 programs. Compared to prior work, our semantics offers broader language coverage and tighter alignment with the specifications. In the process, we uncovered inconsistencies, ambiguities, and bugs in the P4 specification, offering insights for future language evolution.

- **Verifiable modular data structures**: We develop a synthesis framework for constructing approximate data structures compatible with programmable

switches (joint work with Hyojoon Kim [39]). Our modular design supports memory sharding and time window semantics, enabling practical in-network state management. We propose a layered verification strategy across three levels: modular P4 code, concrete models that closely reflect P4 behavior, and abstract models that specify correctness properties (joint work with Qinshi Wang, Shengyi Wang, and Lennart Beringer [46, 47]). This structure supports end-to-end proofs and systematically bridges the gap between formal specifications and hardware-executable code.

- **Network approximate programming language**: We design and implement NAP, a domain-specific language and compiler for expressing network control applications using approximate dictionaries (joint work with Hyojoon Kim [39]). Based on the abstraction of dictionaries over time windows, NAP allows developers to write concise control logic while relying on the compiler to automatically select, configure, and place data structures. The compiler builds on modular synthesis and constrained optimization to generate runnable P4 code, making practical deployment of network control applications on real hardware accessible.

Collectively, these contributions demonstrate how formal methods and system design can be effectively combined to realize verifiable and efficient network control. By grounding verification in formal semantics, synthesizing verified data structures, and raising the programming interface with a high-level language, this thesis offers a principled and practical methodology for building trustworthy network control.

5.2 Future Directions

This thesis lays a foundation for verifiable network control with compact data structures in programmable switches. Several promising directions remain open for ex-

tending the underlying techniques to support broader architectures, applications, and deployment scenarios.

Extending semantics and verification.

Our formal semantics faithfully captures both the P4 language and the Intel Tofino architecture. Future work may extend this foundation to other programmable targets such as eBPF, FPGAs, and smartNICs, which feature distinct architectures and memory models. Although the low-level details vary, many of the high-level techniques developed in this thesis remain broadly applicable: employing formal semantics to surface ambiguities, layering verification around reusable control abstractions, and introducing high-level languages to express control intent. Extending these ideas to new architectures would not require reinventing the framework, but would involve carefully re-mapping the core abstractions to the concurrency, memory, and pipeline models of each platform.

Verifiable P4 supports semi-modular verification by replaying proofs across multiple instances of a control. A more modular verification framework would allow each control to be verified once, ensuring correctness for all instantiations and reducing proof burden.

Finally, bridging synthesis and verification remains a compelling direction. While NAP can synthesize P4 implementations, the associated specifications and correctness proofs must still be written manually. Extending NAP to generate formal specifications and machine-checked proof artifacts would move toward verified-by-construction data plane programs.

More powerful data structures and synthesis.

NAP currently supports a core set of classic approximate data structures. Expanding this set—e.g., to include frequency-aware eviction policies or decay-based retention

schemes—would support more sophisticated applications such as in-network caching and streaming analytics.

On the compiler side, NAP's greedy placement algorithm can be augmented with hybrid strategies that combine pruning-based search with ILP formulations or simulated annealing. Such techniques may improve compilation time for large-scale applications without sacrificing solution quality.

Multi-target and distributed networks.

Modern networks comprise heterogeneous devices with distinct programming models. Today, deploying a control program across multiple targets requires rewriting it for each device. Extending NAP to support cross-device portability and multi-target deployment would allow a single program to be automatically partitioned and compiled across heterogeneous devices or between control and data planes.

A related direction involves reasoning about system-wide properties that span both control-plane and data-plane components. In particular, connecting C-based control plane logic with P4-based data plane behavior to verify global invariants would enable more holistic correctness reasoning in modern networked systems.

By pursuing these directions, future work can extend the reach of verifiable, expressive, and efficient network programming. As programmable networks continue to evolve, the integration of formal methods, data structure design, and domain-specific languages will remain central to building correct and efficient systems.

Appendix A

NAP Examples

To complement the evaluation in the main text, we provide the full code for the nine NAP example applications in this appendix (Figures A.1–A.9). These programs span diverse domains, including network telemetry, monitoring, and control, and each leverages one or more approximate dictionaries. All examples are written in fewer than 30 lines of code, demonstrating both the expressiveness and ease of programming.

```
type key_t = {int<32> int_ip; int<32> ext_ip;}
global ExistDict.t<key_t> seen =
  ExistDict.create(over, within(sec(LO), sec(HI)), Exist());
handle pkt_in(pkt_t p) {
  int<8> s = 1;
  match p.ig_intr_md.ingress_port with
  | INT_PORT -> { ExistDict.add(seen,
                                {ext_ip = p.hdr.ip.dst;
                                 int_ip = p.hdr.ip.src}); }
  | _ -> { s = ExistDict.query(seen,
                               {ext_ip = p.hdr.ip.src;
                                int_ip = p.hdr.ip.dst}); }
  match s with
  | 0 -> { p.ig_intr_dprsr_md.drop_ctl = 0x1; }
  | 1 -> { p.ig_intr_dprsr_md.drop_ctl = 0x0; }
}
```

Figure A.1: Stateful firewall.

```
type key_t = {int<32> int_ip; int<32> ext_ip;}
global ExistDict.t<key_t> seen =
  ExistDict.create(over, within(sec(LO), sec(HI)), Exist());
handle pkt_in(pkt_t p) {
  int<8> s = 1;
  match p.hdr.udp.dport, p.hdr.udp.sport with
  | 53, _ -> { ExistDict.add(seen,
                             {ext_ip = p.hdr.ip.dst;
                              int_ip = p.hdr.ip.src}); }
  | _, 53 -> { s = ExistDict.query(seen,
                                   {ext_ip = p.hdr.ip.src;
                                    int_ip = p.hdr.ip.dst}); }
  match s with
  | 0 -> { p.ig_intr_dprsr_md.drop_ctl = 0x1; }
  | 1 -> { p.ig_intr_dprsr_md.drop_ctl = 0x0; }
}
```

Figure A.2: DNS amplification mitigation.

```
type key_t = {int<32> int_ip; int<32> ext_ip; int<16> client_port;}
global ExistDict.t<key_t> seen = ExistDict.create
  (under, since(sec(INTV)), Exist());
handle pkt_in(pkt_t p) {
  int<8> s = 1;
  match p.hdr.tcp.dport with
  | 21 -> { ExistDict.add(seen,
                          {ext_ip = p.hdr.ip.dst;
                           int_ip = p.hdr.ip.src;
                           client_port = p.hdr.tcp.sport}); }
  | 20 -> { s = ExistDict.query(seen,
                                {ext_ip = p.hdr.ip.src;
                                 int_ip = p.hdr.ip.dst;
                                 client_port = p.hdr.tcp.sport}); }
  match s with
  | 0 -> { p.ctrl_md.monitor = 0x1; }
  | _ -> { p.ctrl_md.monitor = 0x0; }
}
```

Figure A.3: FTP monitoring.

```
type key_t = {int<32> src;}
global CountDict.t<key_t> counter = CountDict.create
  (over, since(sec(INTV)), Count());
handle pkt_in(pkt_t p) {
  int<32> cnt = 0;
  cnt = CountDict.add_query(counter,
                            {src = p.hdr.ip.src});
  match cnt with
  | THR .. MAX_INT -> { p.ctrl_md.monitor = 0x1; }
  | _ -> { p.ctrl_md.monitor = 0x0; }
}
```

Figure A.4: Heavy hitters.

```
type key_t = {int<8> src_prefix;}
type state_t = {int<32> cnt;}
fun state_t init(pkt_t p) { return {cnt = 1}; }
fun state_t upd(pkt_t p, state_t s) { return {cnt = s.cnt + 1}; }
fun int<32> read(state_t s) { return s.cnt; }
global FoldDict.t<key_t> counter = FoldDict.create
  (exact, within(sec(LO), sec(HI)), Fold(init, upd, read));
handle pkt_in(pkt_t p) {
  int<32> cnt = 0;
  cnt = FoldDict.add_query(counter,
                           {src_prefix = p.hdr.ip.src[0:7]});
}
```

Figure A.5: Traffic rate measurement by IP/8.

```
type key_t = {int<32> src_ip; int<32> dst_ip;
              int<32> sport; int<32> dport;}
type state_t = {int<32> prev; int<32> cnt;}
fun state_t init(pkt_t p)
  { return {prev = p.hdr.tcp.seq; cnt = 0}; }
fun state_t upd(pkt_t p, state_t s)
  { return {prev = p.hdr.tcp.seq;
            cnt = s.cnt + 1 if s.prev > p.hdr.tcp.seq else s.cnt}; }
fun int<32> read(state_t s) { return s.cnt; }
global FoldDict.t<key_t> ooo = FoldDict.create
  (under, since(sec(INTV)), Fold(init, upd, read));
handle pkt_in(pkt_t p) {
  int<32> cnt = 0;
  match p.hdr.ip.protocol with
  | 6 -> { cnt = FoldDict.add_query(ooo,
                                    {src_ip = p.hdr.ip.src;
                                     dst_ip = p.hdr.ip.dst;
                                     sport = p.hdr.tcp.sport;
                                     dport = p.hdr.tcp.dport}); }
  match cnt with
  | THR .. MAX_INT -> { p.ctrl_md.monitor = 0x1; }
  | _ -> { p.ctrl_md.monitor = 0x0; }
}
```

Figure A.6: TCP out-of-order monitoring.

```
type key_t = {int<32> src;}
global CountDict.t<key_t> syn = CountDict.create
  (over, since(sec(INTV)), Count());
global CountDict.t<key_t> fin = CountDict.create
  (under, since(sec(INTV)), Count());
handle pkt_in(pkt_t p) {
  int<32> num_syn = 0;
  int<32> num_fin = 0;
  int<32> num_unmatched = 0;
  match p.hdr.tcp.flags with
  | SYN_FLAG ->
      { num_syn = CountDict.add_query(syn, {src = p.hdr.ip.src}); }
  | _ ->
      { num_syn = CountDict.query(syn, {src = p.hdr.ip.src}); }
  match p.hdr.tcp.flags with
  | FIN_FLAG ->
      { num_fin = CountDict.add_query(fin, {src = p.hdr.ip.src}); }
  | _ ->
      { num_fin = CountDict.query(fin, {src = p.hdr.ip.src}); }
  num_unmatched = num_syn - num_fin;
  match num_unmatched with
  |THR .. MAX_INT -> { p.ctrl_md.monitor = 0x1; }
  | _ -> { p.ctrl_md.monitor = 0x0; }
}
```

Figure A.7: TCP overspreader detection.

```
type key_t = {int<32> src;}
global CountDict.t<key_t> syn = CountDict.create
  (over, since(sec(INTV)), Count());
global CountDict.t<key_t> ack = CountDict.create
  (under, since(sec(INTV)), Count());
handle pkt_in(pkt_t p) {
  int<32> num_syn = 0;
  int<32> num_ack = 0;
  int<32> num_unmatched = 0;
  match p.hdr.tcp.flags with
  | SYN_FLAG ->
      { num_syn = CountDict.add_query(syn, { src = p.hdr.ip.src}); }
  | _ ->
      { num_syn = CountDict.query(syn, { src = p.hdr.ip.src}); }
  match p.hdr.tcp.flags with
  | SYN_ACK_FLAG ->
      { num_ack = CountDict.add_query(ack, { src = p.hdr.ip.src}); }
  | _ ->
      { num_ack = CountDict.query(ack, { src = p.hdr.ip.src}); }
  num_unmatched = num_syn - num_ack;
  match num_unmatched with
  |THR .. MAX_INT -> { p.ctrl_md.monitor = 0x1; }
  | _ -> { p.ctrl_md.monitor = 0x0; }
}
```

Figure A.8: TCP SYN flood detection.

```
type key_t = {int<32> key;}
type state_t = {int<32> val;}
global CountDict.t<key_t> counter = CountDict.create
  (over, since(sec(INTV)), Count());
fun state_t init(pkt_t p) { return {val = p.hdr.data.val}; }
fun state_t upd(pkt_t p, state_t s) {
  return {val = p.hdr.data.val}; }
fun int<32> read(state_t s) { return s.val; }
global FoldDict.t<key_t> cache = FoldDict.create
  (under, since(sec(INTV)), Fold(init, upd, read));
handle pkt_in(pkt_t p) {
  int<32> cnt = 0;
  match p.hdr.data.type with
  | REQUEST ->
      { CountDict.add(counter, { key = p.hdr.data.key}); }
  | RESPONSE ->
      { cnt = CountDict.query(counter, { key = p.hdr.data.key}); }
  match cnt, p.hdr.data.type with
  | THR .. MAX_INT, RESPONSE ->
      { FoldDict.add(cache, { key = p.hdr.data.key}); }
  | _, REQUEST ->
      { p.hdr.data.val = FoldDict.query(cache,
                                        {key = p.hdr.data.key}); }
}
```

Figure A.9: NetCache.

Bibliography

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. SIGCOMM, 2014.

[2] Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM TOPLS*, 37(2), April 2015.

[3] Andrew W. Appel and Yves Bertot. C floating-point proofs layered with VST and Flocq. *Journal of Formalized Reasoning*, 13(1), January 2020.

[4] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[5] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. SIGCOMM, 2016.

[6] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. SIGCOMM, 1996.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7), July 1970.

[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3), July 2014.

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. SIGCOMM, 2013.

[10] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. CoNEXT, 2019.

[11] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. SIGCOMM, 2020.

[12] Cisco. Cisco Catalyst 9300 series switches. `https://www.cisco.com/site/us/en/products/networking/switches/catalyst-9300-series-switches/index.html`. Accessed: June 2025.

[13] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Application of sampling methodologies to network traffic characterization. SIGCOMM, 1993.

[14] Joshua M. Cohen, Qinshi Wang, and Andrew W. Appel. Verified erasure correction in Coq with MathComp and VST. CAV, 2022.

[15] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), April 2005.

[16] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: Formal foundations for P4 data planes. POPL, 2021.

[17] Yang Du, He Huang, Yu-E Sun, Shigang Chen, and Guoju Gao. Self-adaptive sampling for network traffic measurement. IEEE INFOCOM, 2021.

[18] Danielle E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. NSDI, 2016.

[19] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. SIGCOMM, 2020.

[20] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. SIGCOMM, 2020.

[21] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. SIGCOMM, 2018.

[22] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. NSDI, 2022.

[23] Intel. Intel Tofino. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html. Accessed: June 2025.

[24] Intel. Intel Tofino native architecture. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, April 2021. Accessed: June 2025.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. SOSP, 2017.

[26] Juniper. EX9200 programmable network switches. https://www.juniper.net/us/en/products/switches/ex-series/ex9200-programmable-network-switch.html. Accessed: June 2025.

[27] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. IMC, 2004.

[28] Jian Li, Hao Jiang, Wei Jiang, Jing Wu, and Wen Du. SDN-based stateful firewall for cloud. IEEE BigDataSecurity-HPSC-IDS, 2020.

[29] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High precision congestion control. SIGCOMM, 2019.

[30] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. SIGCOMM, 2019.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. SIGCOMM, 2017.

[32] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. SIGCOMM, 2017.

[33] Marcin Nawrocki, Mattijs Jonker, Thomas C. Schmidt, and Matthias Wählisch. The far side of DNS amplification: Tracing the DDoS attack ecosystem from the internet core. IMC, 2021.

[34] Open Networking Foundation. Software-defined networking: The new norm for networks. https://opennetworking.org/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf, April 2012. Accessed: June 2025.

[35] P4 Language Design Working Group. P4 specification Github. https://github.com/p4lang/p4-spec. Accessed: June 2025.

[36] P4 Language Design Working Group. $P4_{16}$ language specification, version 1.2.2. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html, May 2021. Accessed: June 2025.

[37] P4 Language Design Working Group. $P4_{16}$ language specification, version 1.2.5. https://p4.org/wp-content/uploads/2024/10/P4-16-spec-v1.2.5.html, October 2024. Accessed: June 2025.

[38] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall Professional, 2003.

[39] Mengying Pan, Hyojoon Kim, Jennifer Rexford, and David Walker. NAP: Programming data planes with approximate data structures. EuroP4, 2023.

[40] Princeton Cabernet. NAP Github. https://github.com/Princeton-Cabernet/NAP/tree/June2025. Accessed: June 2025.

[41] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. SIGCOMM, 2016.

[42] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. SIGCOMM, 2021.

[43] Changhua Sun, Bin Liu, and Lei Shi. Efficient and low-cost hardware defense against DNS amplification attacks. IEEE GLOBECOM, 2008.

[44] Verified Network Toolchain. P4light Github. https://github.com/verified-network-toolchain/petr4/tree/June2025/coq/lib/P4light. Accessed: June 2025.

[45] Verified Network Toolchain. Verifiable P4 Github. https://github.com/verified-network-toolchain/VerifiableP4/tree/June2025. Accessed: June 2025.

[46] Qinshi Wang. *Foundationally Verified Data Plane Programming*. Ph.D. dissertation, Princeton University, 2023. Advised by Andrew W. Appel.

[47] Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W. Appel. Foundational verification of stateful P4 packet processing. ITP, 2023.

[48] Shengyi Wang, Mengying Pan, and Andrew W. Appel. Comprehensive verification of packet processing. *ArXiv:2412.19908*, December 2024.

[49] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. CCS, 2017.

[50] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. NSDI, 2013.

[51] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty years after: Hierarchical core-stateless fair queueing. NSDI, 2021.

[52] Salah Eddine S. E. Zerkane, Philippe Le Parc, Frederic Cuppens, and David Espes. Software defined networking reactive stateful firewall. IFIP AICT, 2016.

[53] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. SIGCOMM, 2021.

[54] Yufei Zheng, Xiaoqi Chen, Mark Braverman, and Jennifer Rexford. Unbiased delay measurement in the data plane. APOCS, 2022.

[55] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. SIGCOMM, 2020.

[56] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-driven network traffic monitoring. CoNEXT, 2020.