

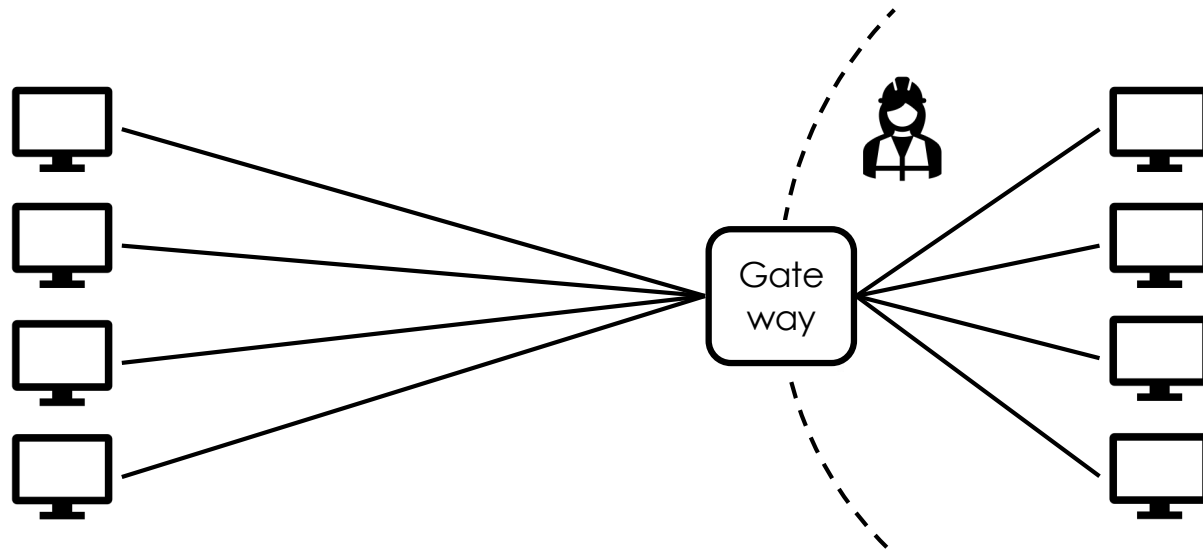
# Verifiable Traffic Control with Compact Data Structures in the Data Plane

Mengying Pan



# Traffic control

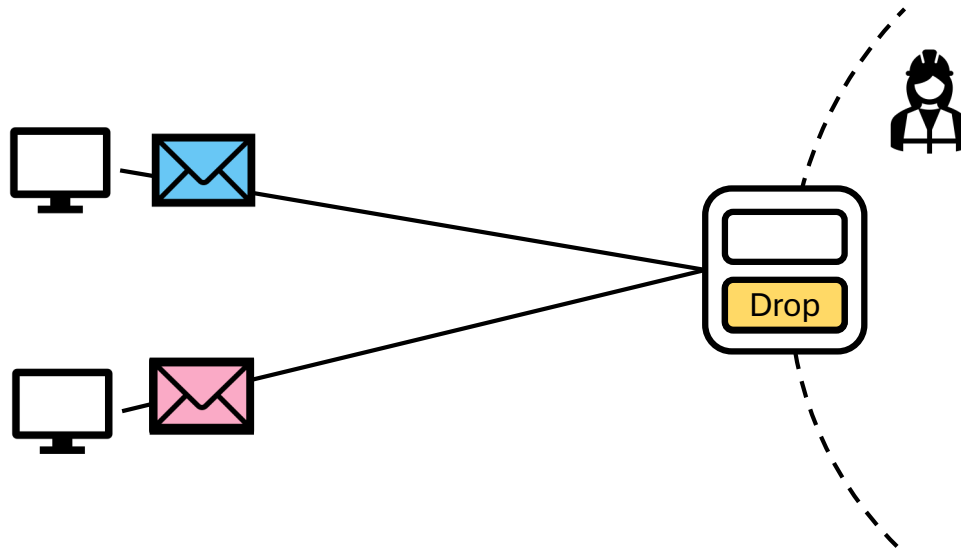
Operators want **real-time control** over network traffic.



# Traffic control

Operators want **real-time control** over network traffic.

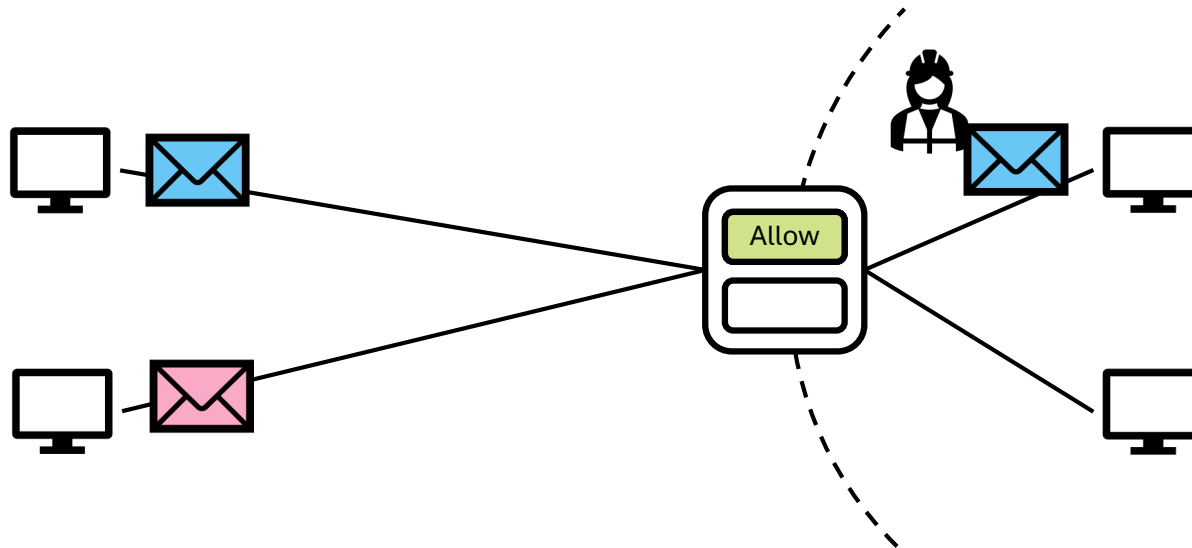
- Access network: rate-limiting large incoming flows



# Traffic control

Operators want **real-time control** over network traffic.

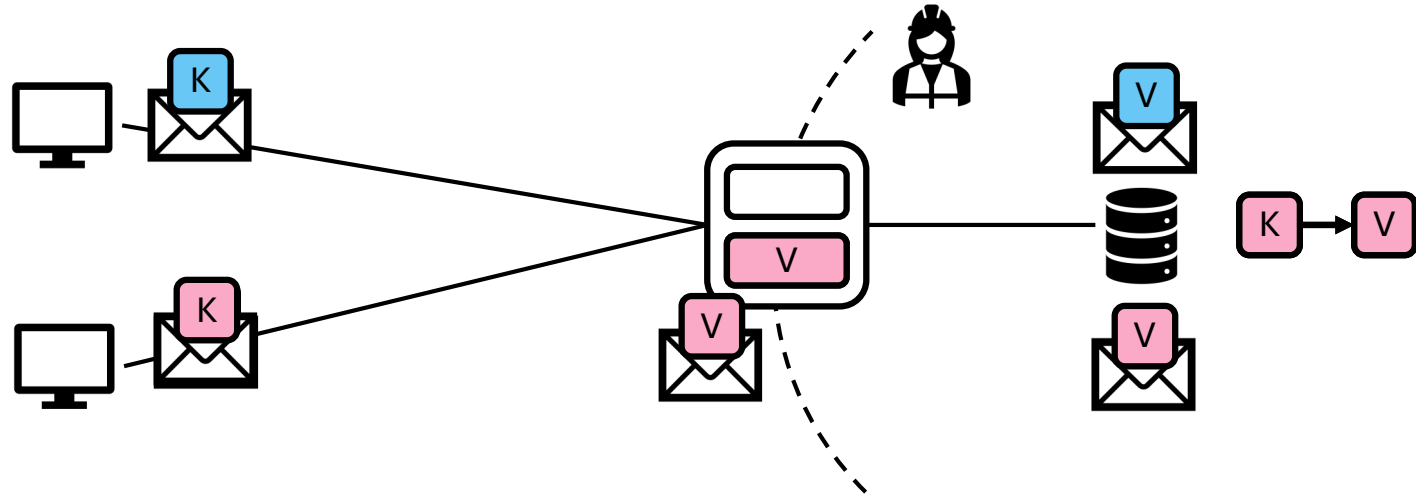
- Access network: rate-limiting large incoming flows
- Enterprise network: dropping unsolicited packets



# Traffic control

Operators want **real-time control** over network traffic.

- Access network: rate-limiting large incoming flows
- Enterprise network: dropping unsolicited packets
- Datacenter network: caching popular key-value pairs



# Stateful applications

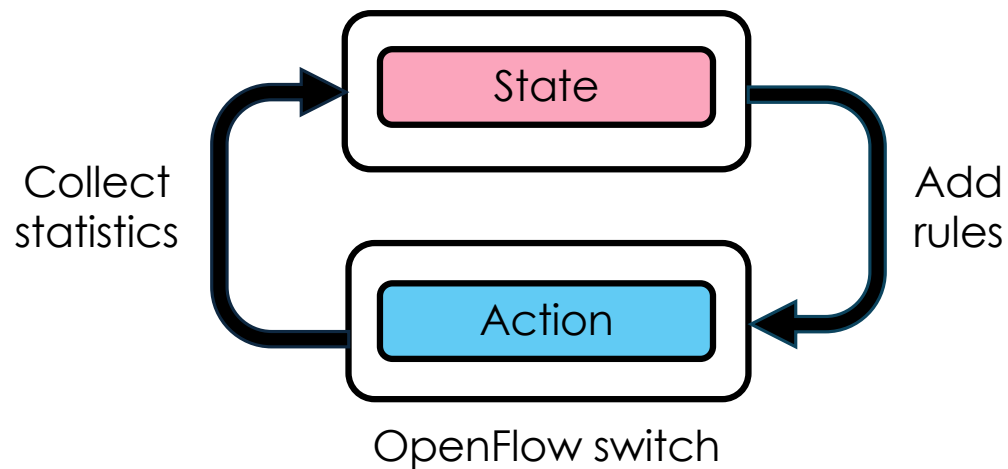
Traffic control applies **actions** on packets based on the **state**.

Example	State	Action
Rate limiter	<b>count</b> the packets	<b>limit</b> the rates of large flows
Stateful firewall	<b>record</b> the flow IDs of outgoing traffic	<b>drop</b> the incoming traffic with unmatched flow IDs
Key-value store	<b>store</b> the popular key-value pairs	<b>resolve</b> the values for popular keys

# Where to store states?

Traffic control applies **actions** on packets based on the **states**.

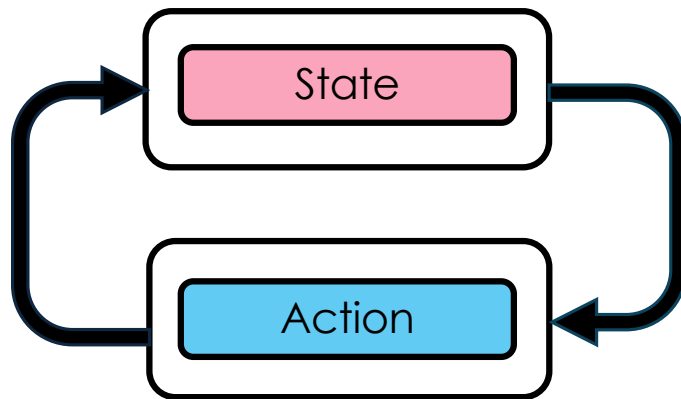
- OpenFlow-based deployment incurs high overhead and latency.



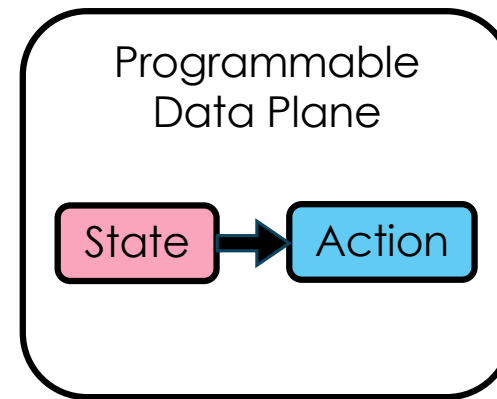
# State in the data plane

Traffic control applies **actions** on packets based on the **states**.

- OpenFlow-based deployment incurs high overhead and latency.
- Programmable data planes allows state access at line rate.



OpenFlow switch



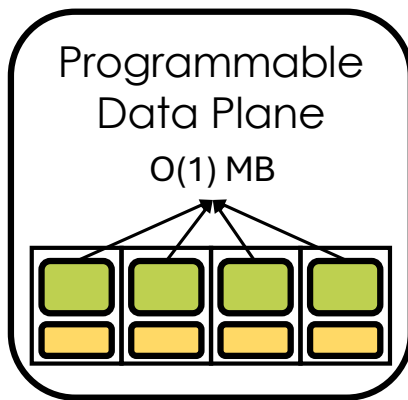
PISA(Protocol Independent Switch Architecture) switch



# PISA switches

PISA switches enable stateful network control to run in the data plane.

To maintain line-rate processing, PISA switches are inherently restricted in architectures & resources.



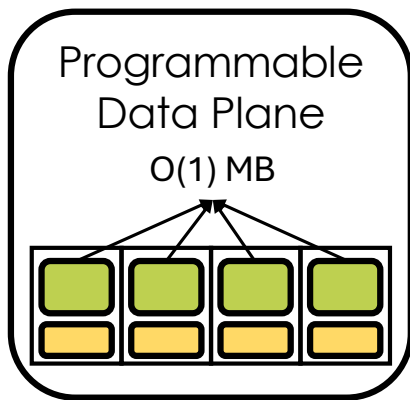
PISA switch

- **Finite-stage pipeline**
- **Restricted memory access**
- **Limited memory resources**
- **Limited computational resources**

# Resource constraints

PISA switches enable stateful network control to run in the data plane.

To maintain line-rate processing, PISA switches are inherently restricted in architectures & **resources**.

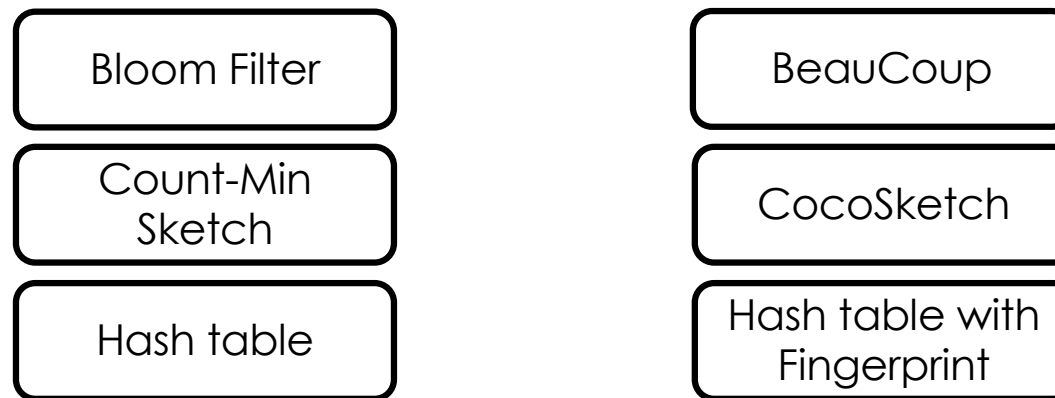


PISA switch

- **Finite-stage pipeline**
- **Restricted memory access**
- **Limited memory resources**  
Cannot keep exact per-flow state in data structures.
- **Limited computational resources**  
Cannot apply sophisticated data processing.

# Approximate data structures

- **Resource constraints** demand network control applications to use **approximate data structures** to represent state compactly.



- Approximations are **acceptable** in many applications.

B. H. Burton, *Space/time trade-offs in hash coding with allowable errors*. Communications of the ACM 1970.

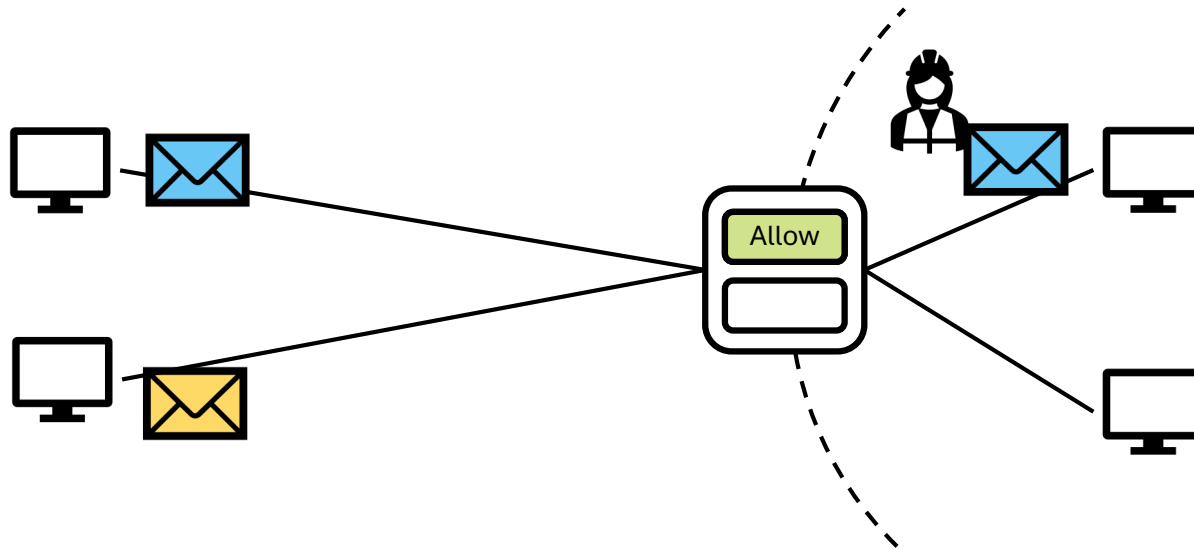
X. Chen, et al., *BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time*. SIGCOMM '20.

Y. Zhang, et al., *CocoSketch: high-performance sketch-based measurement over arbitrary partial key query*. SIGCOMM '21.

G. Cormode, *Count-Min Sketch*. 2009.

# Approximate traffic control

- Approximate stateful firewall: guarantees access to all the solicited packets, at the cost of **sometimes allowing the unsolicited ones**.

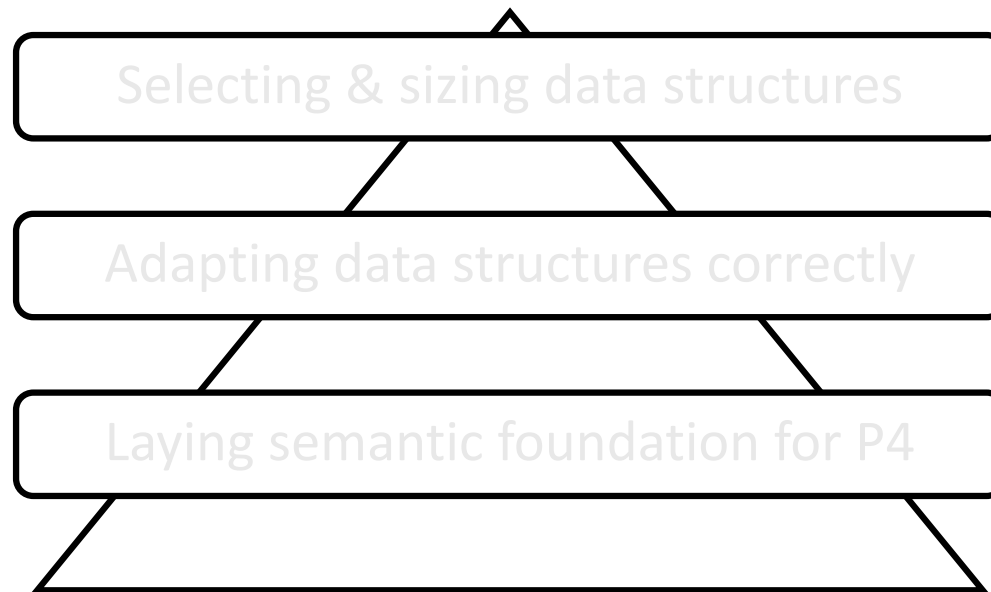


# Approximate traffic control

- Approximate stateful firewall: guarantees access to all the solicited packets, at the cost of **sometimes allowing the unsolicited ones...**
- For network control applications, it is **feasible to run entirely in the data plane** since approximation is tolerable in data structures.

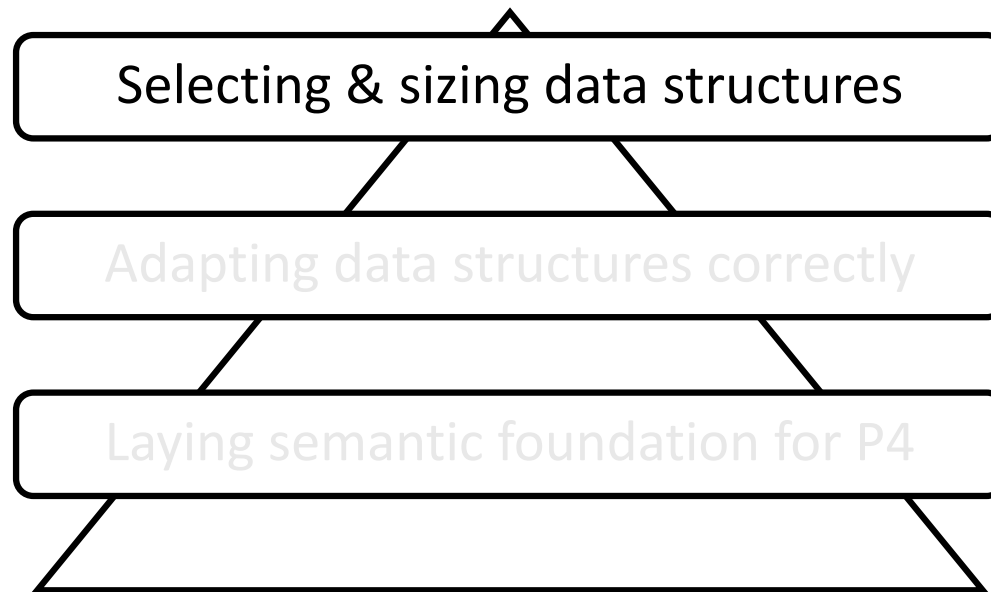
# Three challenges

Verifiable traffic control with  
approximate data structures in the data plane



# Three challenges

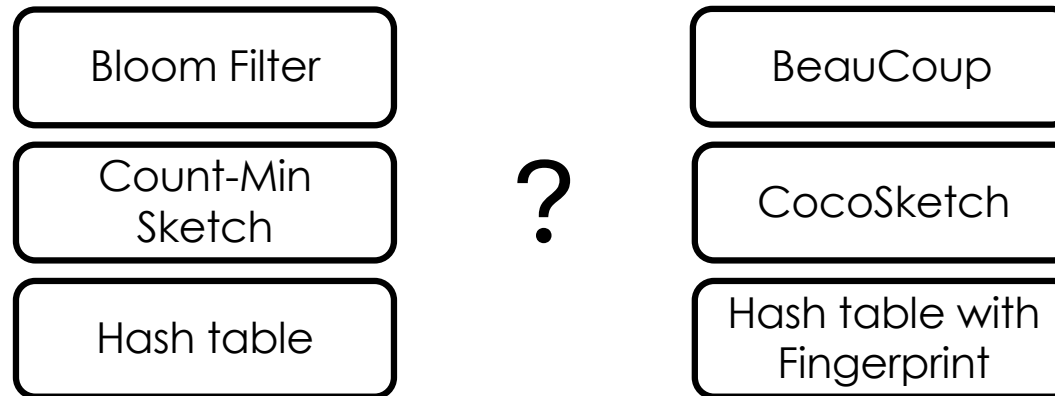
Verifiable traffic control with  
approximate data structures in the data plane



# Selecting & sizing data structures

- **Selecting** data structures

Which approximate data structure supports the application intention?






# Selecting & sizing data structures

- **Selecting** data structures

Which approximate data structure supports the application intention?

- **Sizing** data structures

How to size the data structure to minimize the approximation error?



Data Structure

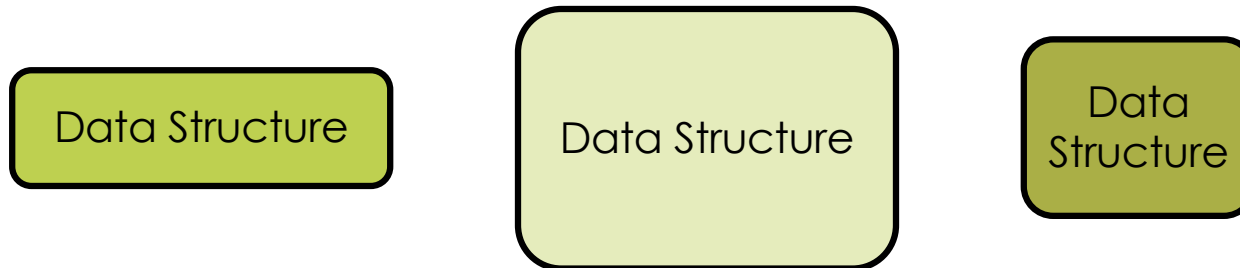
# Selecting & sizing data structures

- **Selecting** data structures

Which approximate data structure supports the application intention?

- **Sizing** data structures

How to size the data structure to minimize the approximation error?



# Selecting & sizing data structures

- **Selecting** data structures

Which approximate data structure supports the application intention?

- **Sizing** of data structures

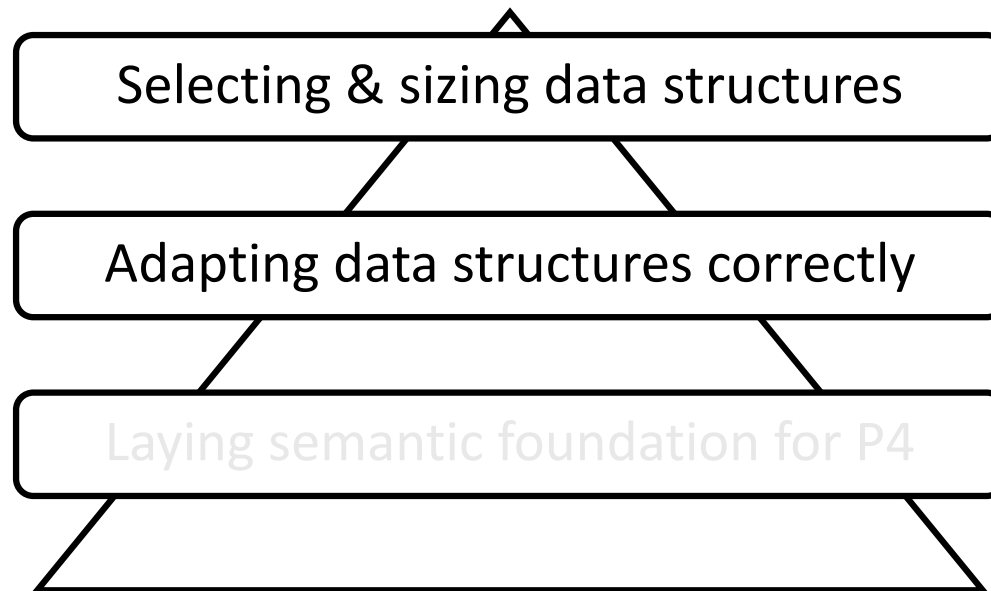
How to

error?

Programming traffic control applications is hard without expertise in approximate data structures.

# Three challenges

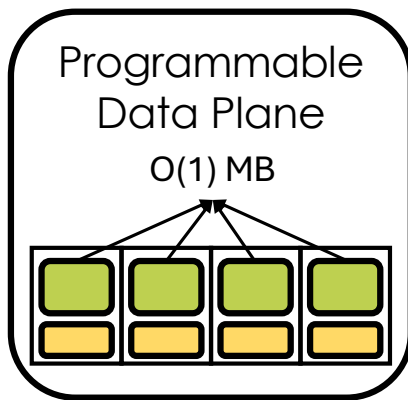
Verifiable traffic control with  
approximate data structures in the data plane



# Architectural constraints

PISA switches enable stateful network control to run in the data plane.

To maintain line-rate processing, PISA switches are inherently restricted in **architectures** & resources.



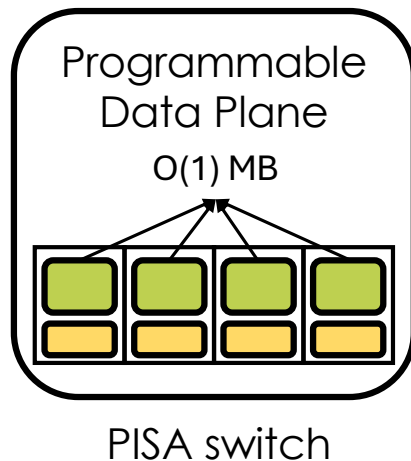
PISA switch

- **Finite-stage pipeline**  
Cannot implement general-purpose loops.
- **Restricted memory access**  
Cannot access memory across stages.
- **Limited memory resources**  
Cannot keep exact per-flow state in data structures.
- **Limited computational resources**  
Cannot apply sophisticated data processing.

# Adapting data structures...

- **Adapting** data structures

Given architectural constraints, how to **implement** data structures for the data plane?



# P4 language

P4 is **a domain-specific language** for expressing packet processing on the programmable data planes.

- **Low-level**

Hardware-oriented and C-like

- **Specialized constructs**

PISA-specific features such as actions, tables, and control blocks

- **Informal target semantics**

Fragmented vendor documents

- **Informal language semantics**

189-page P4 specification in prose & examples

# Adapting data structures... correctly!

- **Adapting** data structures

Given architectural constraints, how to **implement** data structures in the data plane?

- **Verifying** data structures

Given language complexity, how to ensure **correctness** of the adapted P4 implementation?



# Adapting data structures... correctly!

- **Adapting** data structures

Given architectural constraints, how to **implement** data structures in the data

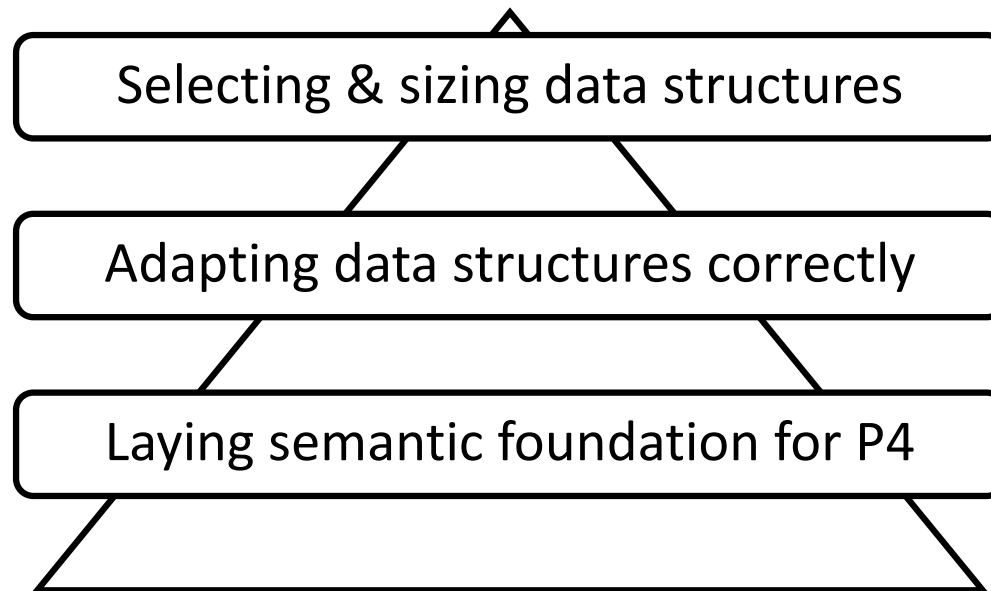
- **Verifying**

Given I  
P4 imp

Implementing data structures correctly is hard adapted  
without expertise in architectures & verification.

# Three challenges

Verifiable traffic control with  
approximate data structures in the data plane



# Laying semantic foundation for P4

- To adapt data structures correctly, we need to:
  - Write programs in P4
  - Build P4 verifiers
- **Formal semantics**: the foundation for both tasks  
The **mathematical specification of program behavior**
- Example: ++ denotes concatenation in P4.
  - Concatenating two 8-bit bitstrings should yield a 16-bit result  
 $8w0 \text{ ++ } 8w1 = 16w1$

# P4 language

P4 is **a domain-specific language** for expressing packet processing on the programmable data planes.

- **Low-level**

Hardware-oriented and C-like

- **Specialized constructs**

PISA-specific features such as actions, tables, and control blocks

- **Informal target semantics**

Fragmented vendor documents

- **Informal language semantics**

189-page P4 specification in prose & examples

# The problem of informal semantics

- **Natural-language specifications**

Both vendor docs & the P4 spec are written informally

- **Ambiguities and bugs** make it difficult to:

- Write programs in P4 with confidence
- Build reliable P4 verifiers

- Example: ++ denotes concatenation in P4.

- Concatenating two 8-bit bitstrings should yield a 16-bit result  
 $8w0 \text{ ++ } 8w1 = 16w1$
- The P4 spec was unclear for fixed-width bitstrings  
→ Formal semantics are needed to resolve this ambiguity

# The problem of informal semantics

- **Natural-language specifications**

Both vendor docs & the P4 spec are written informally

- **Ambiguity**

- Program
- Build

- Examples

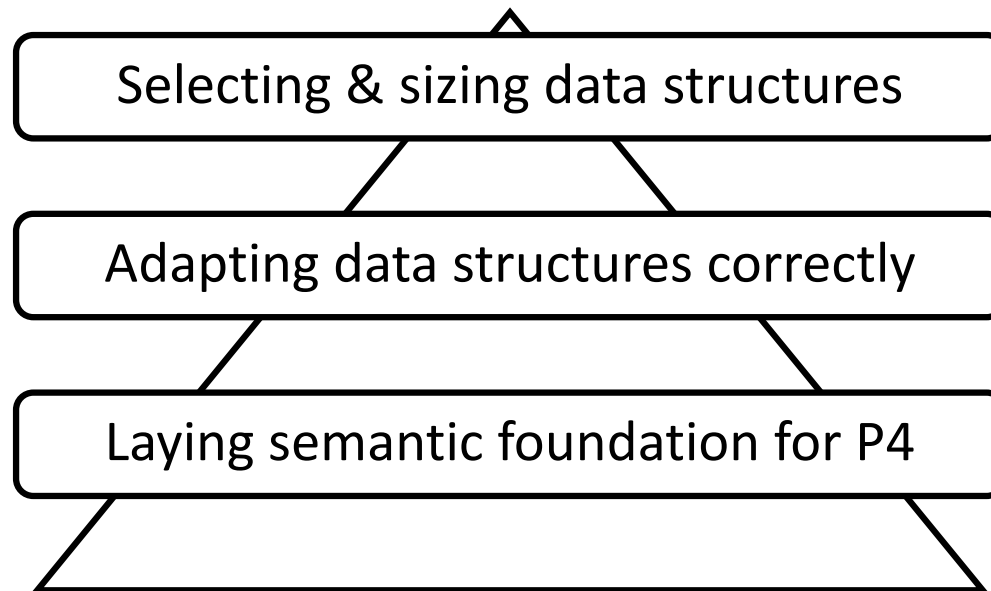
- 8w0
- 8w1
- The P4 spec was unclear for lines 17 and 18 on page 17

→ Formal semantics are needed to resolve this ambiguity

Programming the data plane is hard  
without expertise in P4.

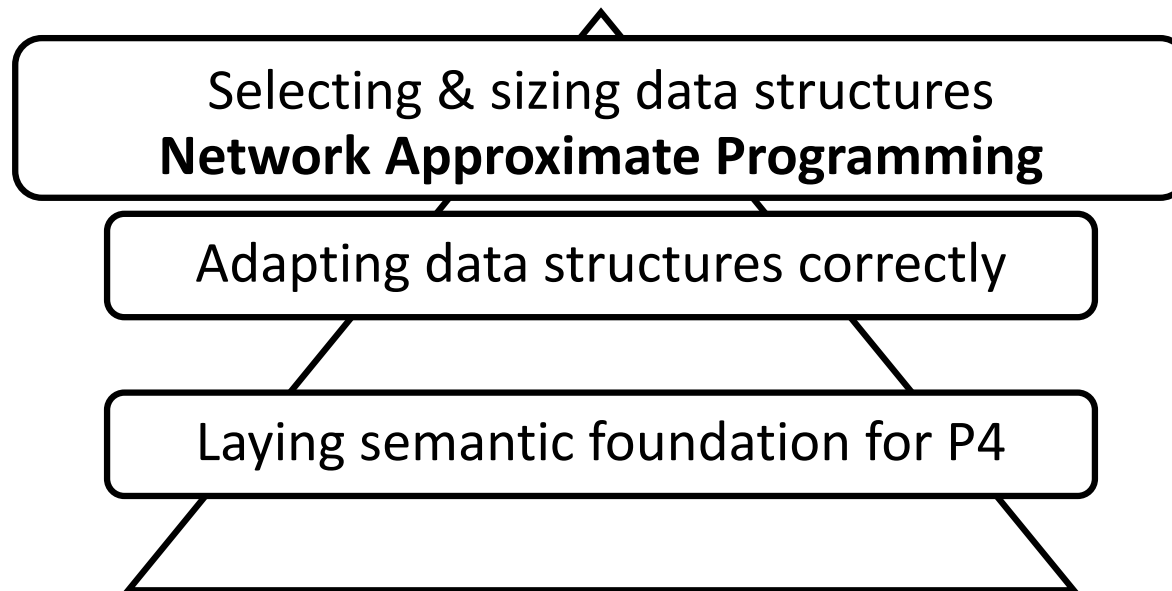
# Three challenges

Verifiable traffic control with  
approximate data structures in the data plane



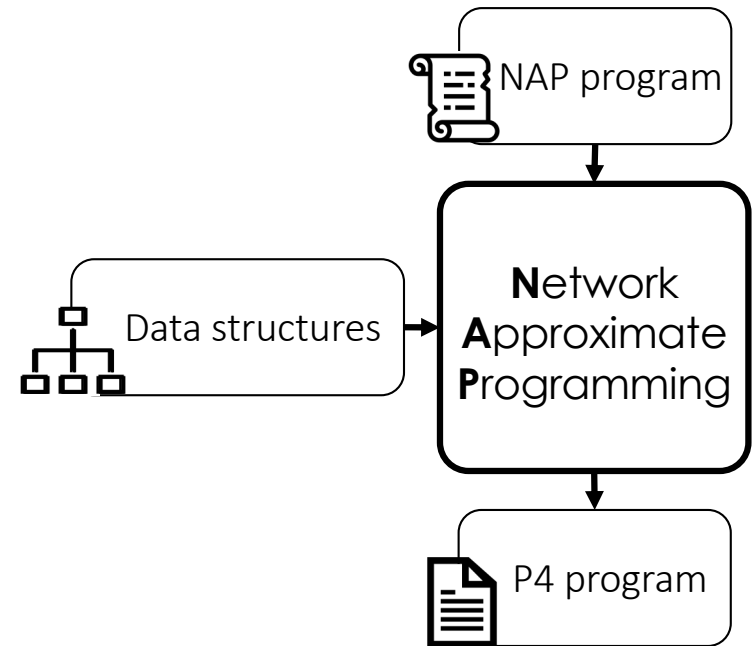
# Three contributions

Verifiable traffic control with  
approximate data structures in the data plane





# Network Approximate Programming



# Three contributions

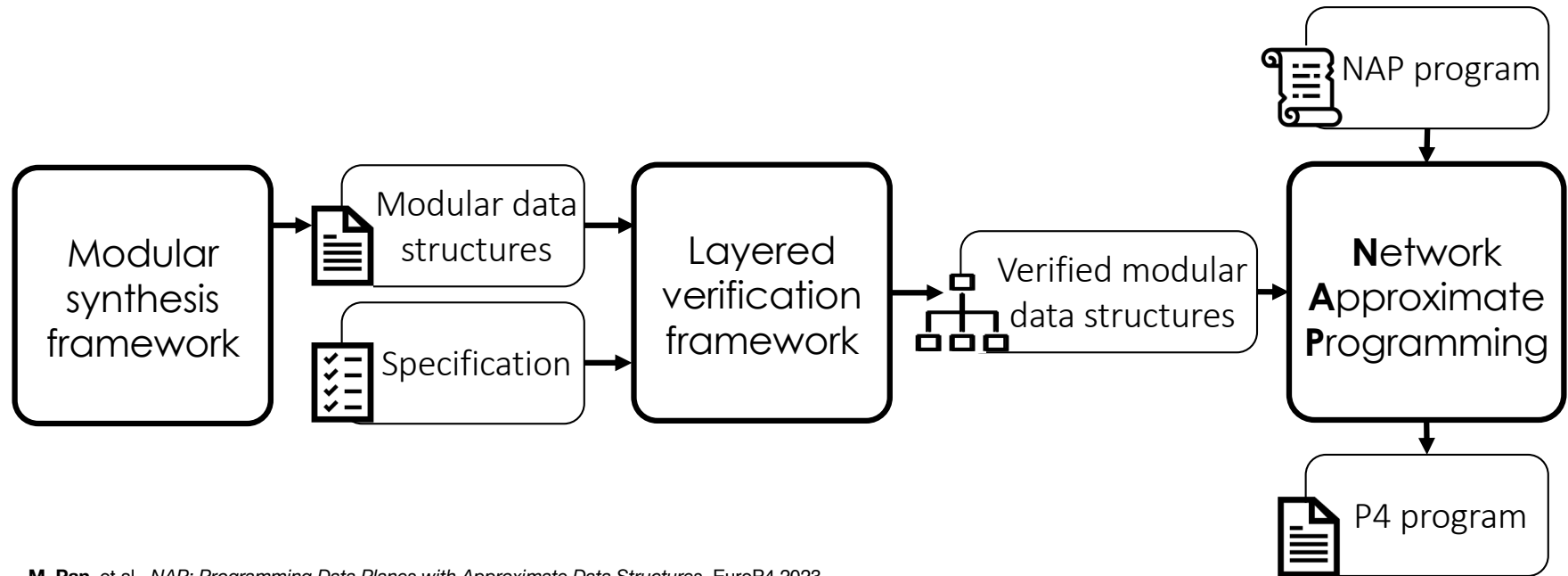
Verifiable traffic control with  
approximate data structures in the data plane

Selecting & sizing data structures  
**Network Approximate Programming**

Adapting data structures correctly  
**Verified modular data structures**

Laying semantic foundation for P4

# Verified modular data structures



M. Pan, et al., *NAP: Programming Data Planes with Approximate Data Structures*. EuroP4 2023.  
Q. Wang, M. Pan, et al., *Foundational Verification of Stateful P4 Packet Processing*. ITP 2023.

# Three contributions

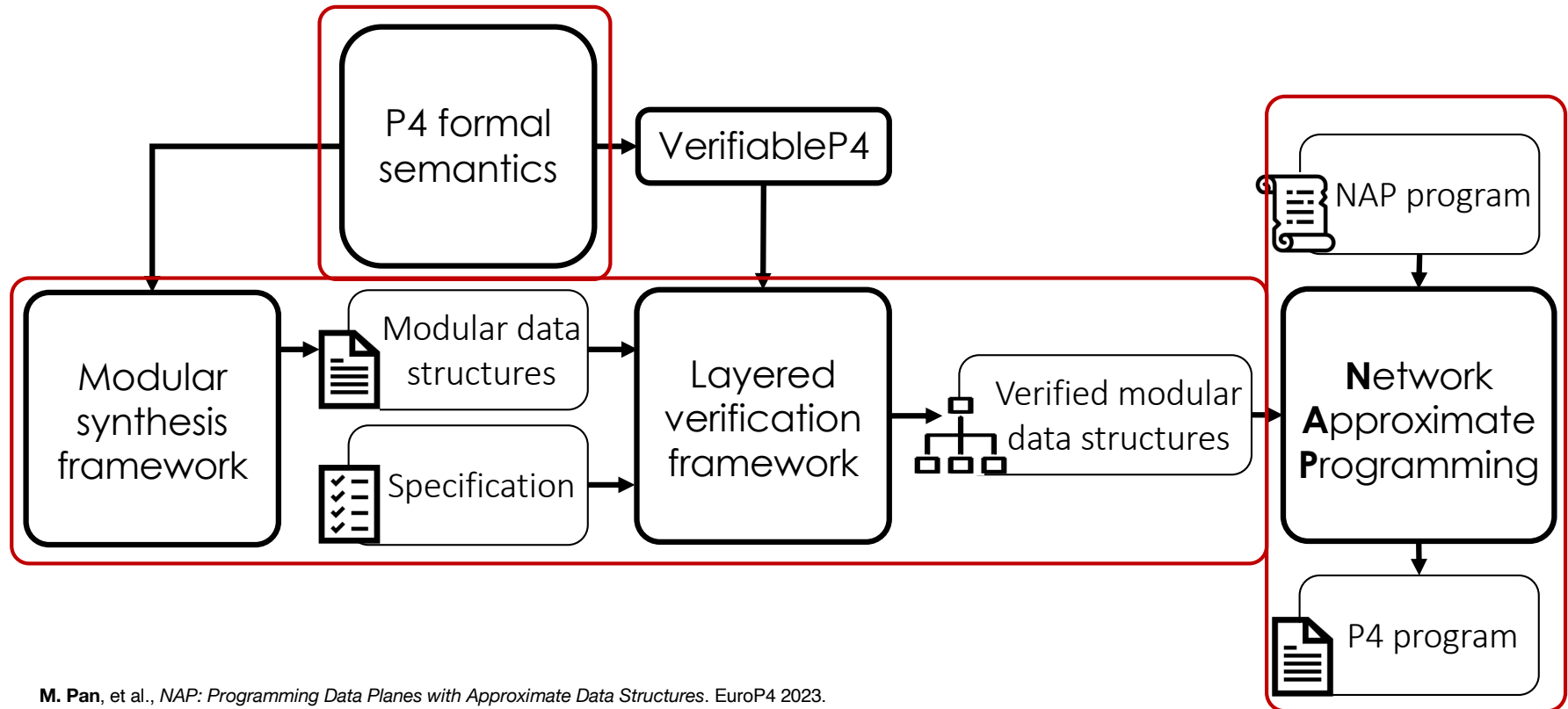
Verifiable traffic control with  
approximate data structures in the data plane

Selecting & sizing data structures  
**Network Approximate Programming**

Adapting data structures correctly  
**Verified modular data structures**

Laying semantic foundation for P4  
**P4 formal semantics**

# P4 formal semantics



# Outline

Motivations, challenges & contributions

Network Approximate Programming

Verified modular data structures

P4 formal semantics

Conclusions & future directions

# Outline

Motivations, challenges & contributions

Network Approximate Programming

Verified modular data structures

P4 formal semantics

Conclusions & future directions

# Selecting & sizing data structures

**Problem:** Developers must manually **select and size data structures**, requiring deep domain knowledge and extensive tuning.

**Limitations:** Existing high-level languages lack support for approximate data structures or automate selection/sizing.

	Approximate data structures	Selecting data structures	Sizing data structures
Marple	Only hash tables	No	No
Sonata	Only sketches	No	Yes
Newton	Yes	No	No
Lucid	Yes	No	No
P4All	Yes	No	Yes



# Network approximate programming

**Approach:** **NAP**, a high-level language for approximate network control

**Insight:**

- **Language:** A simple universal **abstraction** for approximate data structures
- **Compiler:** Translates NAP programs into P4
  - **Selecting data structures:** guided by high-level control intent
  - **Sizing data structures:** lightweight greedy optimizer with pre-pruning

# Identifying the common pattern

How to design an abstraction that works universally across network control applications?

- **Key:** flow identifier
- **Value:** stateful information

Example	State	Key	Value
Rate limiter	<b>count</b> the packets	Source IP	Number of packets
Stateful firewall	<b>record</b> the flow IDs of outgoing traffic	(Internal IP, External IP)	Existence
Key-value store	<b>store</b> the popular key-value pairs	Application key	Application value

# Identifying common patterns

- **Key:** flow identifier
- **Value:** stateful information

Bloom Filter

BeauCoup

Count-Min  
Sketch

CocoSketch

Hash table

Hash table with  
Fingerprint

# Approximate dictionary abstraction

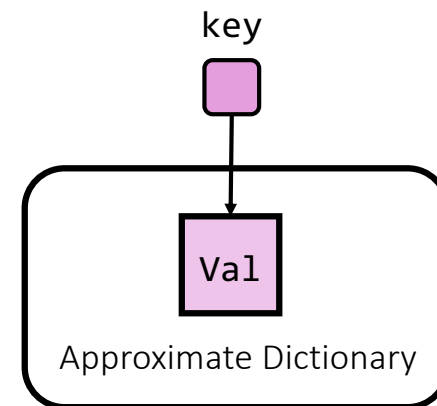
- **Key:** flow identifier
- **Value:** stateful information

Approximate dictionaries represent a wide variety of approximate data structures in a uniform way.

Approximate  
Dictionary

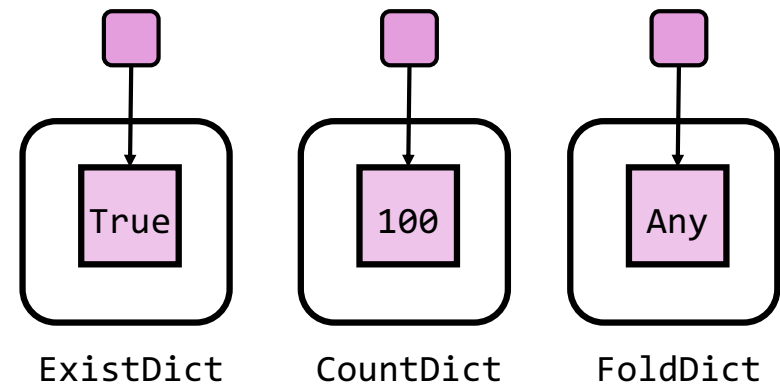
# Basic dictionary operations

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)



# Dictionary classes

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
  - Exist: Query(key) -> Bool
  - Count: Query(key) -> Int
  - Fold: Query(key) -> Any



# Two approximation dimensions

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - Inclusion approximation
  - Temporal approximation

# Error directions

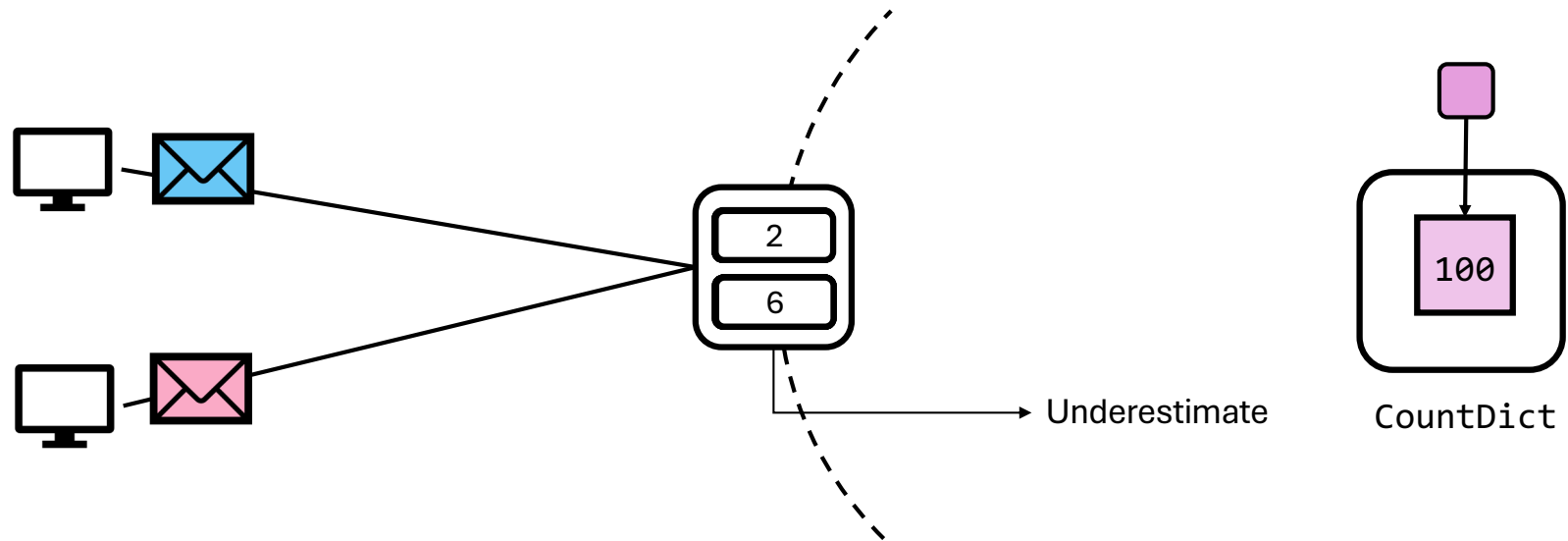
Many applications tolerate errors, but favor **a specific direction**.



# Error directions

Many applications tolerate errors, but favor **a specific direction**.

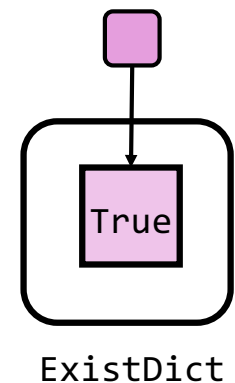
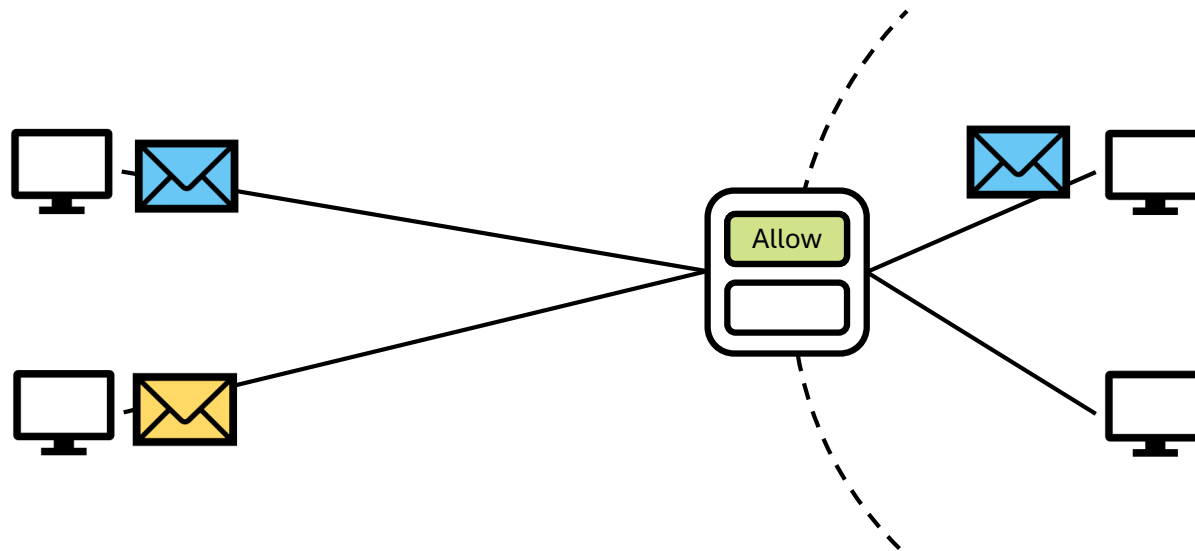
- Rate limiter: underapproximate the counts



# Error directions

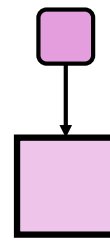
Many applications tolerate errors, but favor **a specific direction**.

- Rate limiter: underapproximate the counts
- Stateful firewall: overapproximate the ID set



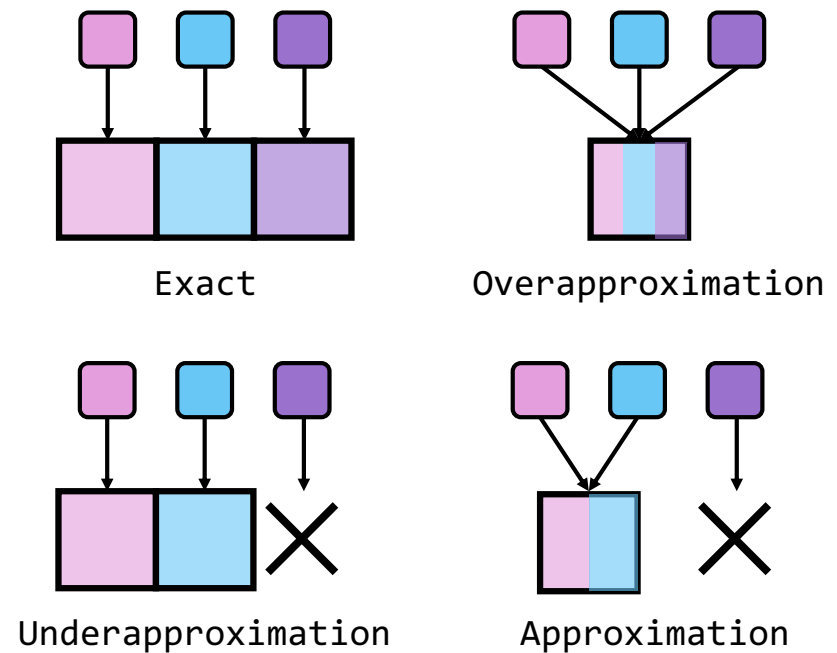
# Error directions in dictionaries

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction



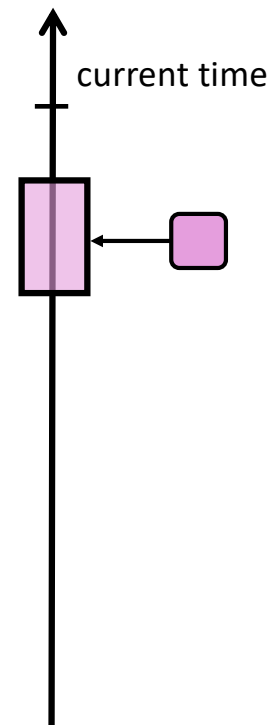
# Error directions in dictionaries

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction



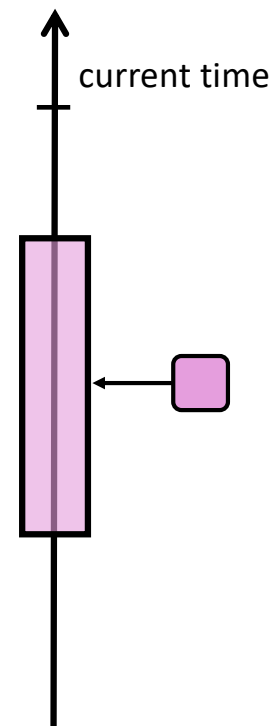
# Time windows in dictionaries

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction
  - **Temporal approximation:** time window



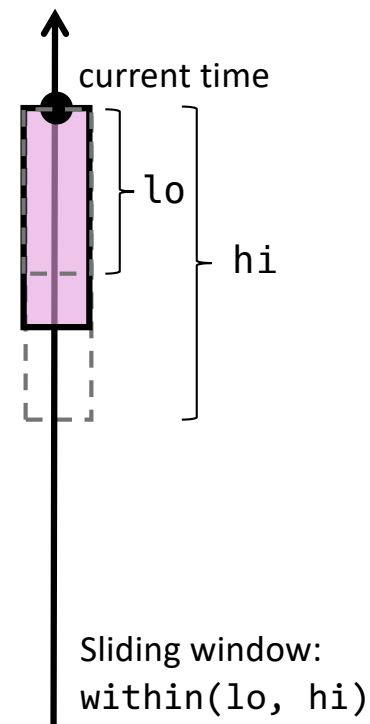
# Time windows in dictionaries

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction
  - **Temporal approximation:** time window



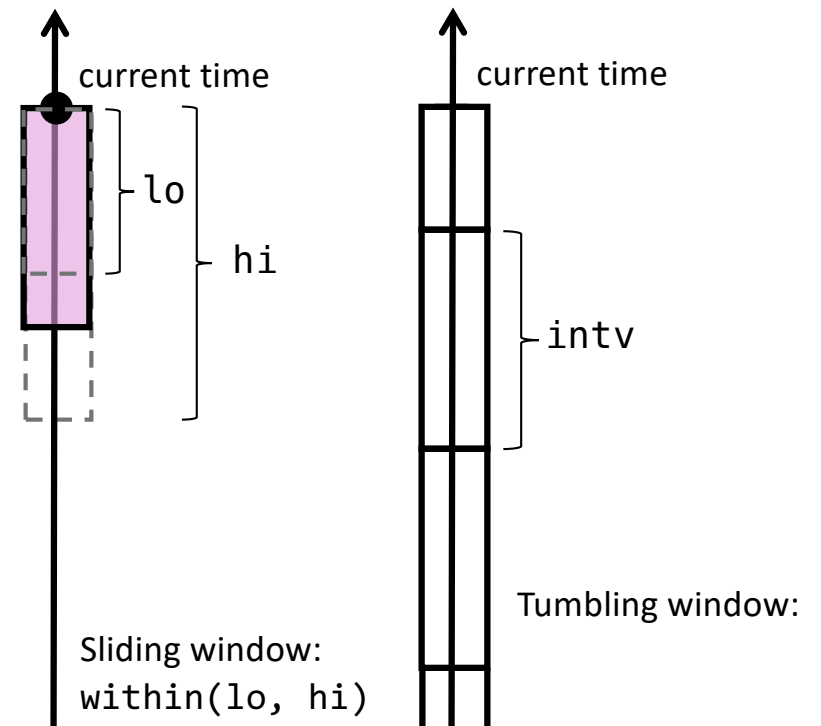
# Sliding time window

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction
  - **Temporal approximation:** time window



# Tumbling time window

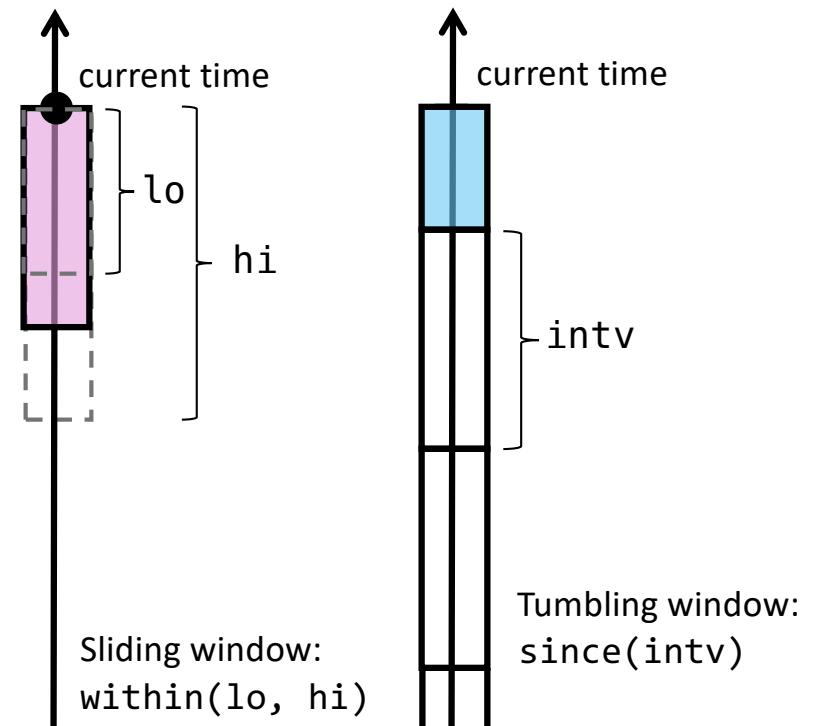
- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction
  - **Temporal approximation:** time window





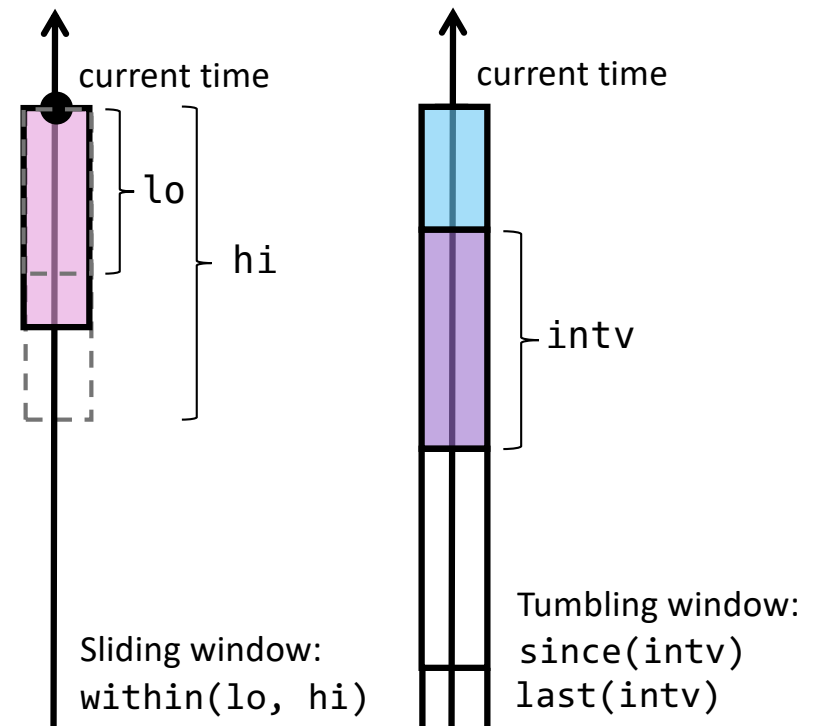
# Tumbling time window

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction
  - **Temporal approximation:** time window



# Tumbling time window

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - `Create<key>(parameters)`
  - `Add(key)`
  - `Query(key)`
- **Dictionary Class:** value updates
- **Parameters:**
  - **Inclusion approximation:** error direction
  - **Temporal approximation:** time window



# Example: approximate stateful firewall

- **Key:** flow identifier

- **Value:** stateful information

- **Operations:**

- Create<key>(parameters)
- Add(key)
- Query(key)

- **Dictionary Class:** value updates

- **Parameters:**

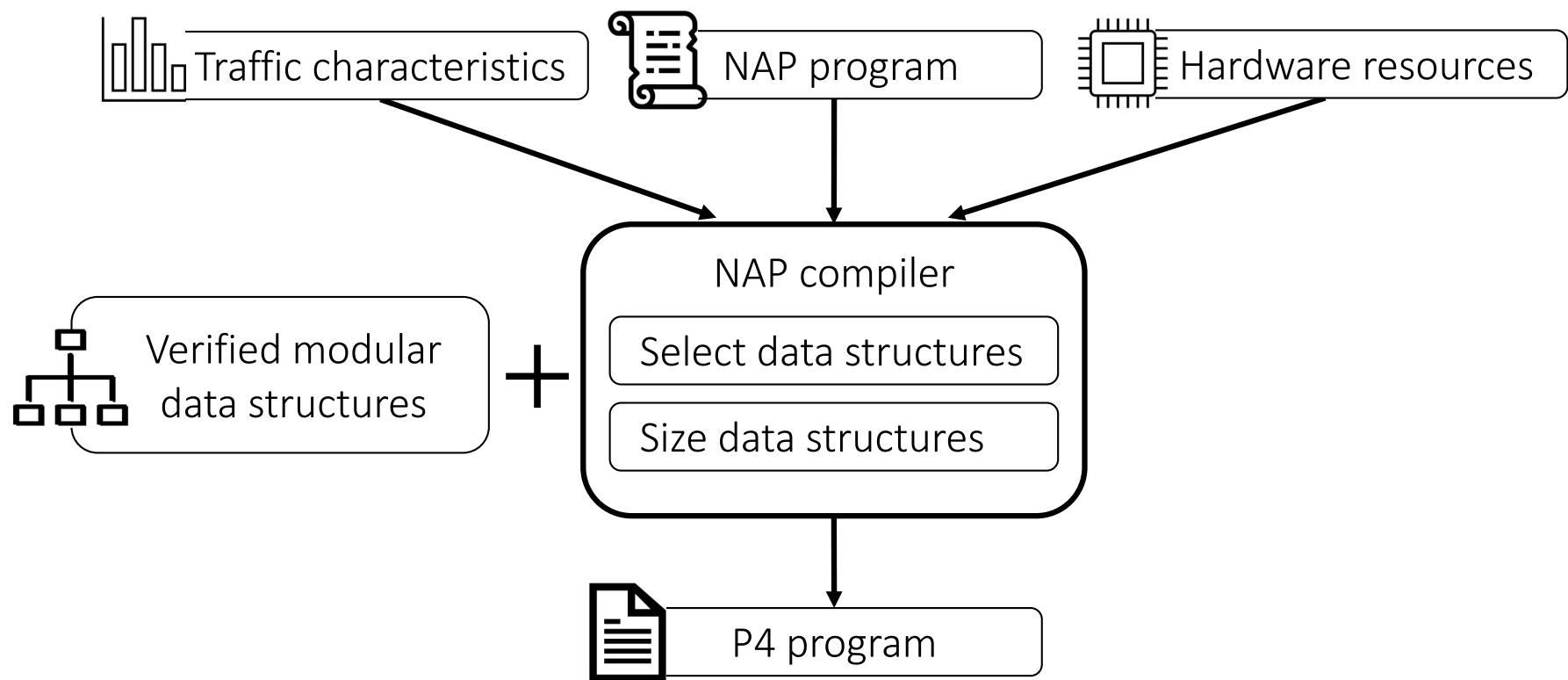
- **Inclusion approximation:** error direction
- **Temporal approximation:** time window

```
type key = {int eip; int iip}
ExistDict<key> IDset =
  ExistDict.create (over,
                    within(sec(60),sec(90)),
                    ExistDict())

...
bool sol = true;
match p.ig_md.ig_port with
| INTERNAL_PORT -> {
  ExistDict.add(IDset, {eip = p.ip.dip;
                       iip = p.ip.sip});}
| _ -> {
  sol = ExistDict.query(IDset,
                        {eip = p.ip.sip;
                         iip = p.ip.dip});}
}

...
```

# NAP compiler



# Compiler: select data structures

- **Dictionary classes:**

- ExistDict
- CountDict
- FoldDict

- **Error directions:**

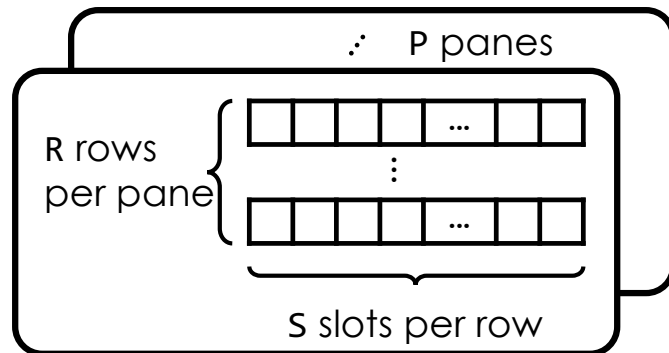
- Exact
- Overapproximation
- Underapproximation
- Approximation

	ExistDict
Exact	Exact array
Over	Bloom filter
Under	Hash table w. full fp
Approx	All of above, Hash table w. partial fp

```
type key = {int eip; int iip}  
ExistDict<key> IDset =  
    ExistDict.create (over,  
                      within(sec(60),sec(90)),  
                      ExistDict())
```

# Compiler: size data structures

- Verified modular data structures
- **Parameterized** implementation:  
(P, R, S) tuples
- **Constrained optimization**



Variables:

- P: number of panes
- R: number of rows per pane
- S: number of slots per row

Minimize:

The expected false positive rate of a Bloom filter

Constrained by:

- Time constraints
- Memory constraints
- Computational constraints
- Architectural constraints

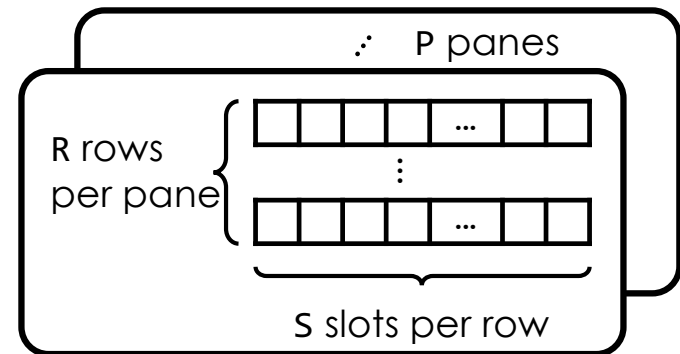
# Pruning size search space

Optimizing the size parameters is straightforward in NAP.

- Users don't need to define utility functions.
- The size of search space is (surprisingly) small.
  - Few size parameters
  - Limited memory resources
  - Practical parameter choices

(P, R, S) tuples:

- $P \times R$ : bounded by the number of registers
  - $\leq 50$  for 10-stage pipeline w. 5-register/stage
- S: power of 2, bounded by register size
  - $2^1, 2^2, \dots, 2^{23}$  for 1 MB register



# A lightweight greedy optimizer

Optimizing the size parameters is straightforward in NAP.

- Users don't need to define utility functions.
- The size of search space is (surprisingly) small.
- Greedy optimization algorithm:
  - Compute the utility of all the possible parameter tuples.
  - Rank the parameter tuples based on their utility.
  - Allocate parameter tuples in order until finding the best one that fits.



# Evaluations

- **Generalizability**

- A diverse set of nine example applications in network telemetry, monitoring, and control

Applications	LoC		Compile Time (s)
	NAP	P4	
Single Dictionary			
Stateful firewall	15	555	0.0055
DNS amplification mitigation	15	582	0.0056
FTP monitoring	20	798	0.0035
Heavy hitter detection	8	595	0.0049
Traffic rate measurement by IP/8	12	466	0.0040
TCP out-of-order monitoring	19	559	0.0043
Multiple Dictionaries			
TCP superspreader detection	20	842	0.0130
TCP SYN flood detection	20	842	0.0130
NetCache	22	802	0.0394

# Evaluations

- **Generalizability**
- **Simplicity**
  - All example applications expressed within 30 LoC
  - A reduction of 25X to 50X in LoC

Applications	LoC		Compile Time (s)
	NAP	P4	
Single Dictionary			
Stateful firewall	15	555	0.0055
DNS amplification mitigation	15	582	0.0056
FTP monitoring	20	798	0.0035
Heavy hitter detection	8	595	0.0049
Traffic rate measurement by IP/8	12	466	0.0040
TCP out-of-order monitoring	19	559	0.0043
Multiple Dictionaries			
TCP superspreader detection	20	842	0.0130
TCP SYN flood detection	20	842	0.0130
NetCache	22	802	0.0394

# Evaluations

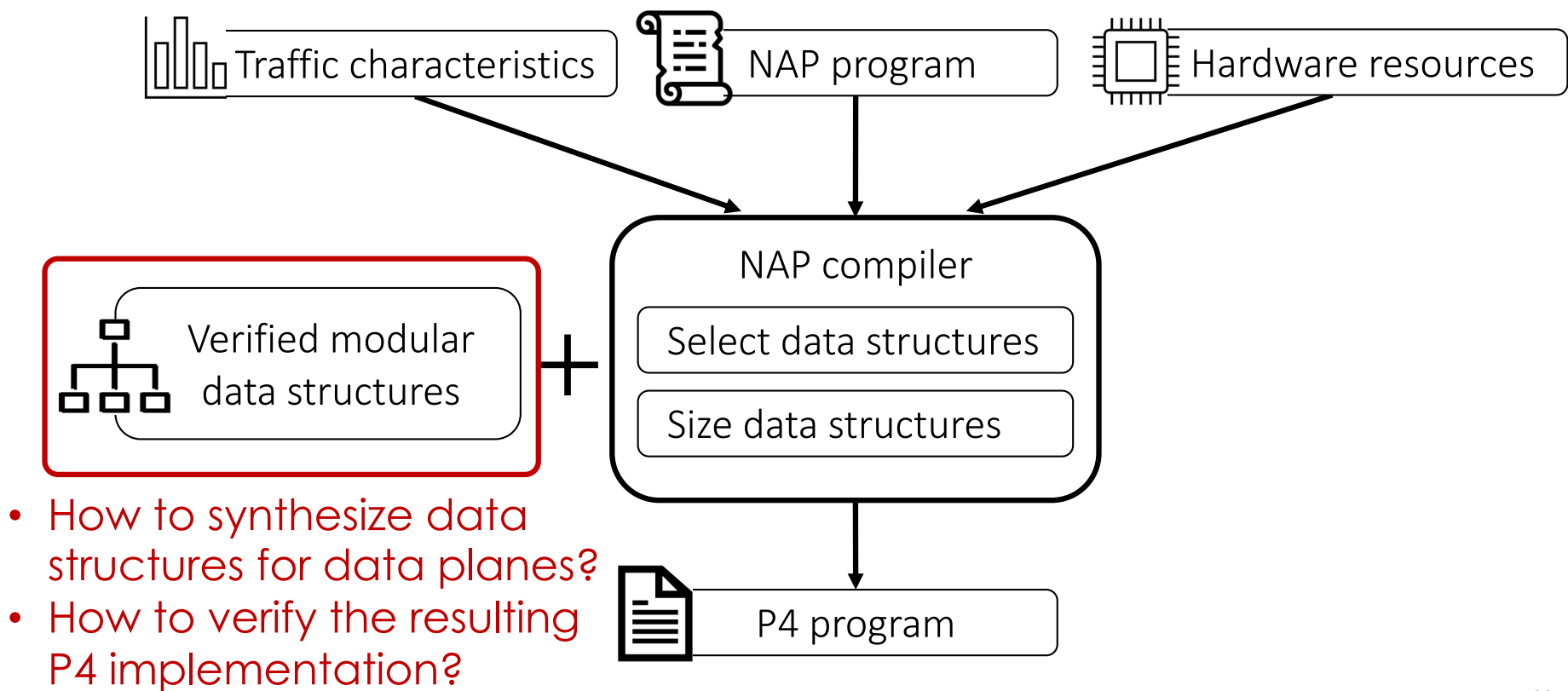
- **Generalizability**
- **Simplicity**
- **Fast compilation**
  - All examples compiled to P4 for the Intel Tofino target within 0.1 second

Applications	LoC		Compile Time (s)
	NAP	P4	
Single Dictionary			
Stateful firewall	15	555	0.0055
DNS amplification mitigation	15	582	0.0056
FTP monitoring	20	798	0.0035
Heavy hitter detection	8	595	0.0049
Traffic rate measurement by IP/8	12	466	0.0040
TCP out-of-order monitoring	19	559	0.0043
Multiple Dictionaries			
TCP superspreader detection	20	842	0.0130
TCP SYN flood detection	20	842	0.0130
NetCache	22	802	0.0394

# Advantages of NAP

- NAP is a **domain-specific** language for approximate network control.
- NAP **selects** & **sizes** the right data structures automatically.

# What is still missing?



# Outline

Motivations, challenges & contributions

Network Approximate Programming

Verified modular data structures

P4 formal semantics

Conclusions & future directions

# Adapt data structures correctly

**Problem:** Developers must **design and verify data structures** under strict architectural constraints and the complexity of the P4 language.

## **Limitations:**

- Existing **synthesis frameworks** are:
  - **Monolithic**: P4 programs often rely on a single, tightly coupled control block.
  - **Ad hoc**: Techniques are often tailored to specific data structures.
  - **Unrefreshable**: Designs often lack mechanisms to periodically evict stale data.
- Existing **verification frameworks** are:
  - **Monolithic**: Verification happens directly on low-level P4 code, making proofs brittle and hard to scale or reuse.
  - **Inexpressive**: Properties handled by solvers can only be simple logical formulas.

# Modular synthesis & layered verification

## Approach:

- **Modular synthesis framework:** Decomposes data structures into reusable, constraint-aware modules.
- **Layered verification framework:** Connects high-level specs to low-level P4 code through stepwise refinement.

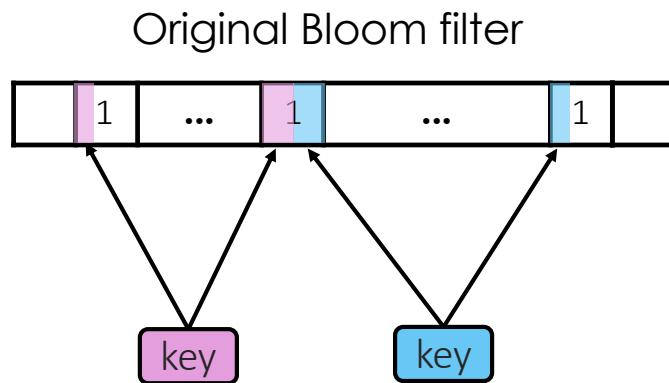
**Insight:** Breaking down monolithic designs and proofs into modular or layered components simplifies both implementation and verification.



# Modular synthesis framework

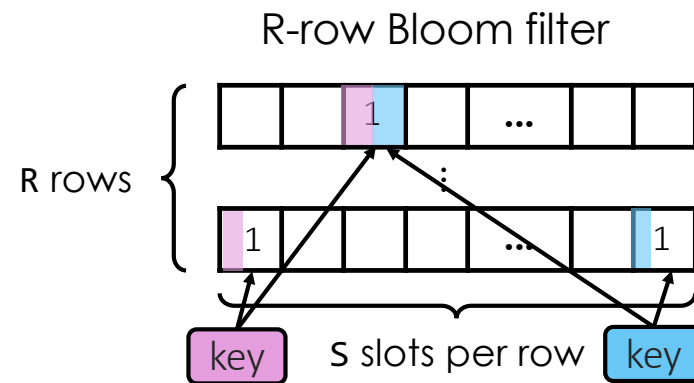
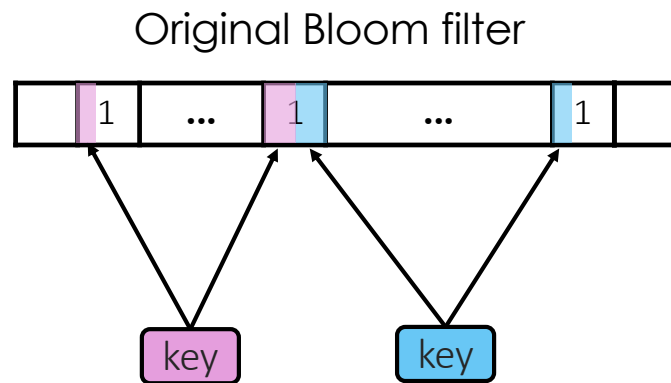
**Modular synthesis framework:** Decomposes data structures into reusable, constraint-aware modules.

- **Restricted state access:** Requires careful allocation of the data structure.
- **Finite number of stages:** Requires a practical line-rate cleaning scheme.



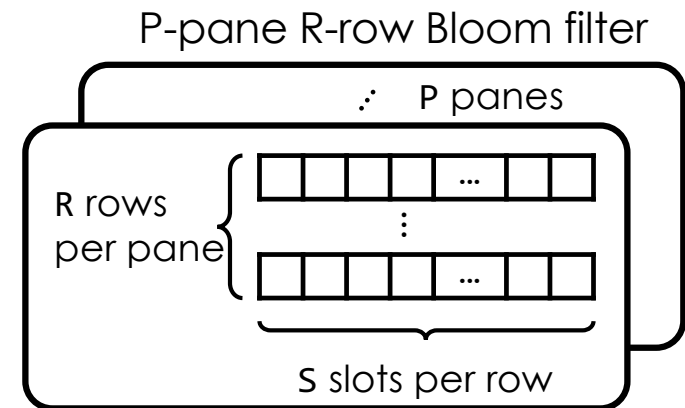
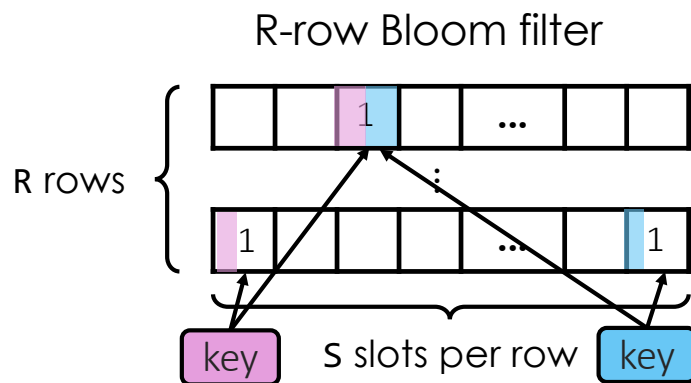
# Row module

- Shard a data structure into **rows** for the staged pipeline.



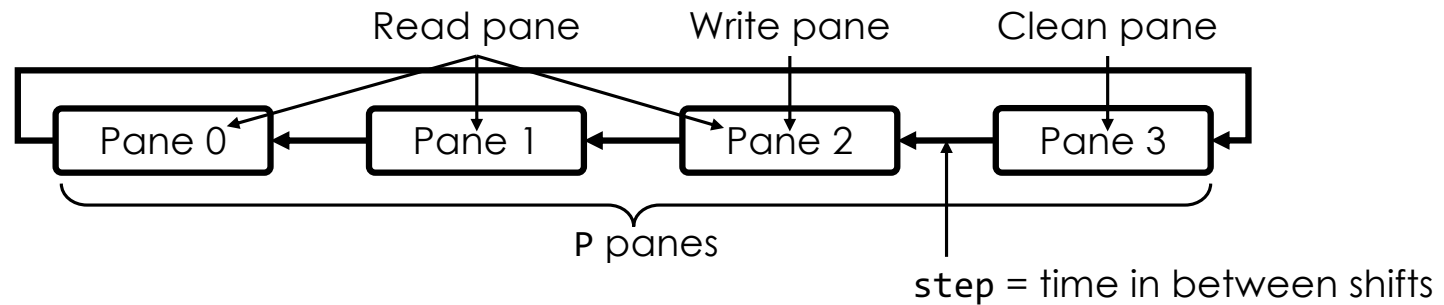
# Pane module

- Shard a data structure into **rows** for the staged pipeline.
- Rotate a data structure by **panes** for the cleaning purpose.



# Rotation Timer

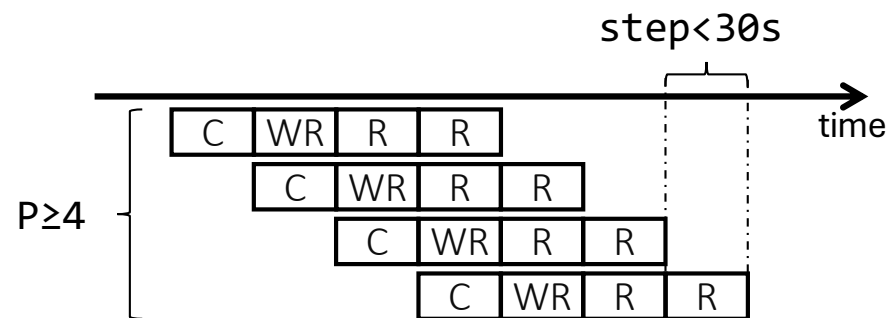
- **Pre-processing** for deciding
  - Rotation timer:
    - Time window length  $\in [(P-2) \cdot \text{step}, (P-1) \cdot \text{step}]$  ( $P \geq 2$ )



# Supporting time windows in NAP

- **Pre-processing** for deciding
  - Rotation timer:
    - Time window length  $\in [(P-2) \cdot \text{step}, (P-1) \cdot \text{step}]$  ( $P \geq 2$ )
    - Supporting tumbling and sliding windows
    - Two parameters: number of panes & step

`within(sec(60), sec(90))`

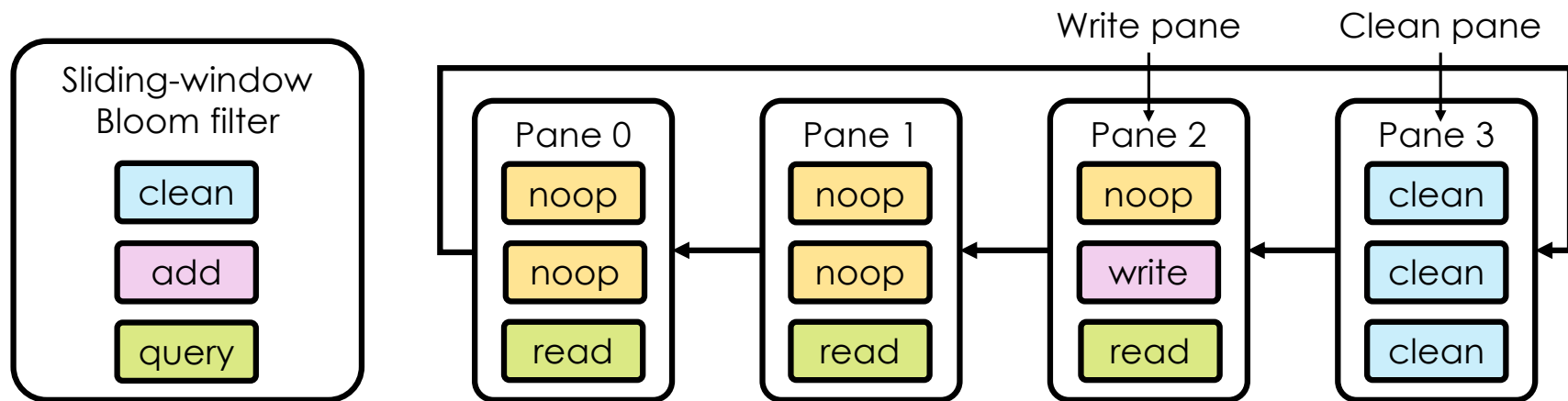


# State operations

- **Pre-processing** for deciding

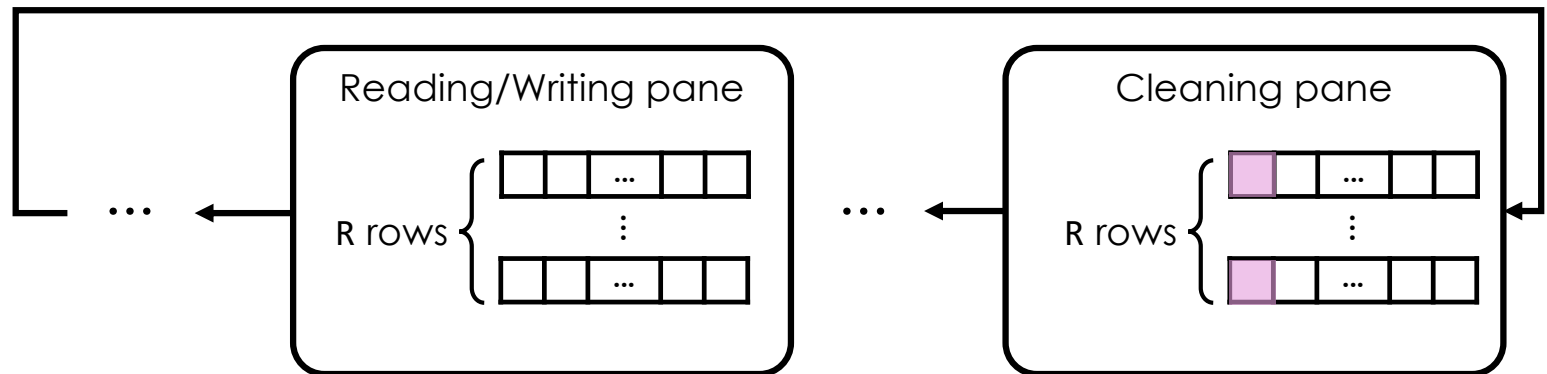
- Rotation timer

- Operations: clean write read noop



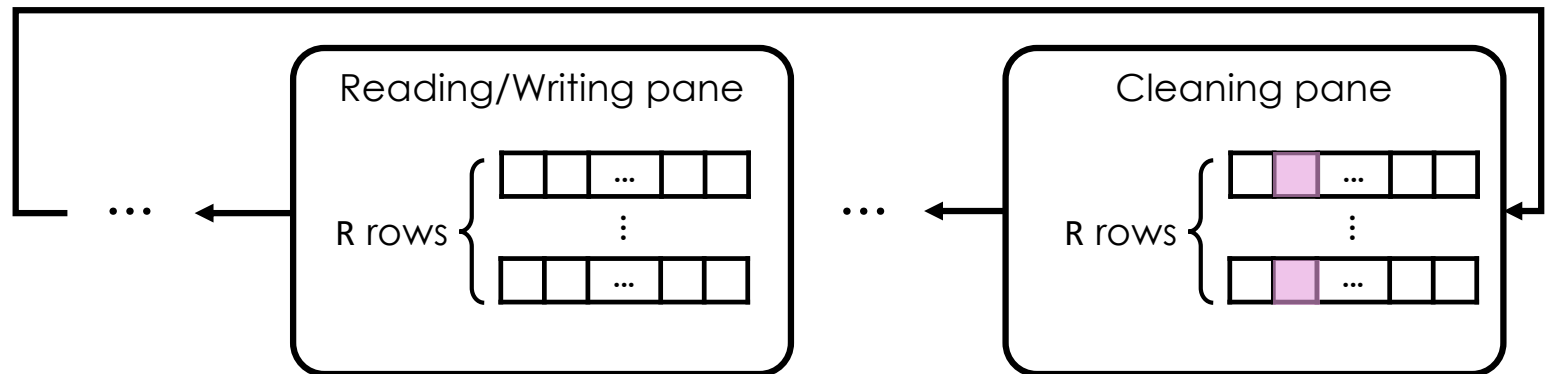
# Computing indexes

- **Pre-processing** for deciding
  - Rotation timer
  - Operations
  - Indexes: **Incremental indexes** for cleaning



# Computing indexes

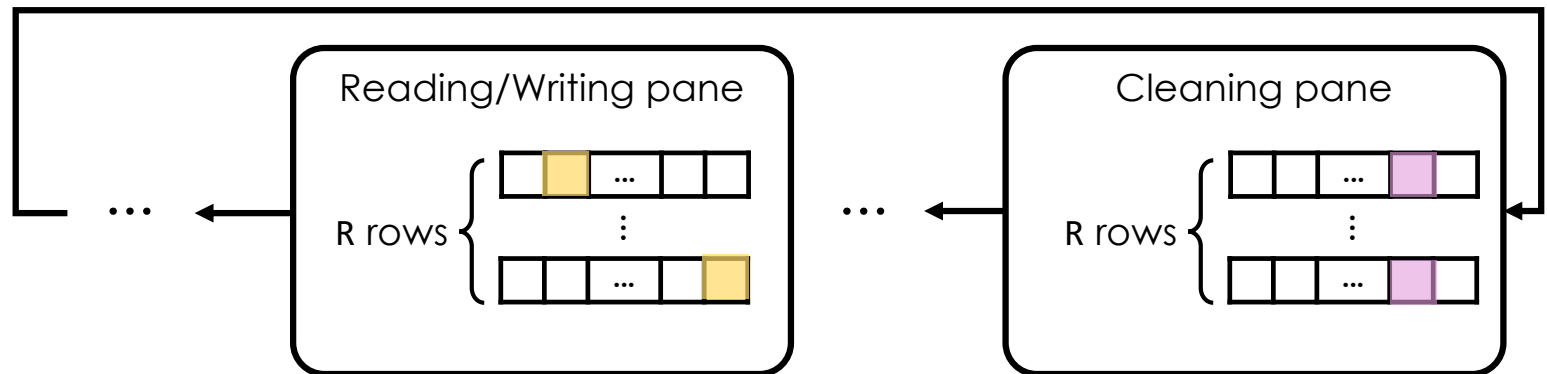
- **Pre-processing** for deciding
  - Rotation timer
  - Operations
  - Indexes: **Incremental indexes** for cleaning





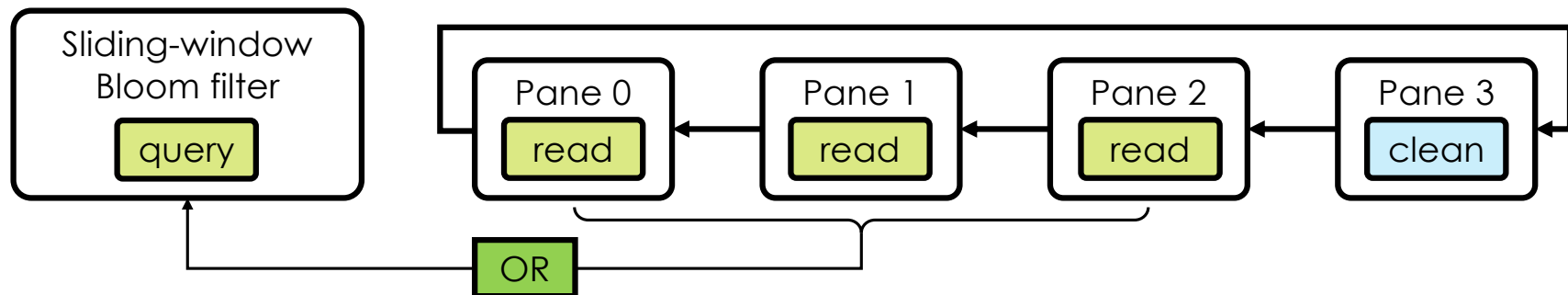
# Computing indexes

- **Pre-processing** for deciding
  - Rotation timer
  - Operations
  - Indexes: **Incremental indexes** for cleaning & **hash indexes** for access



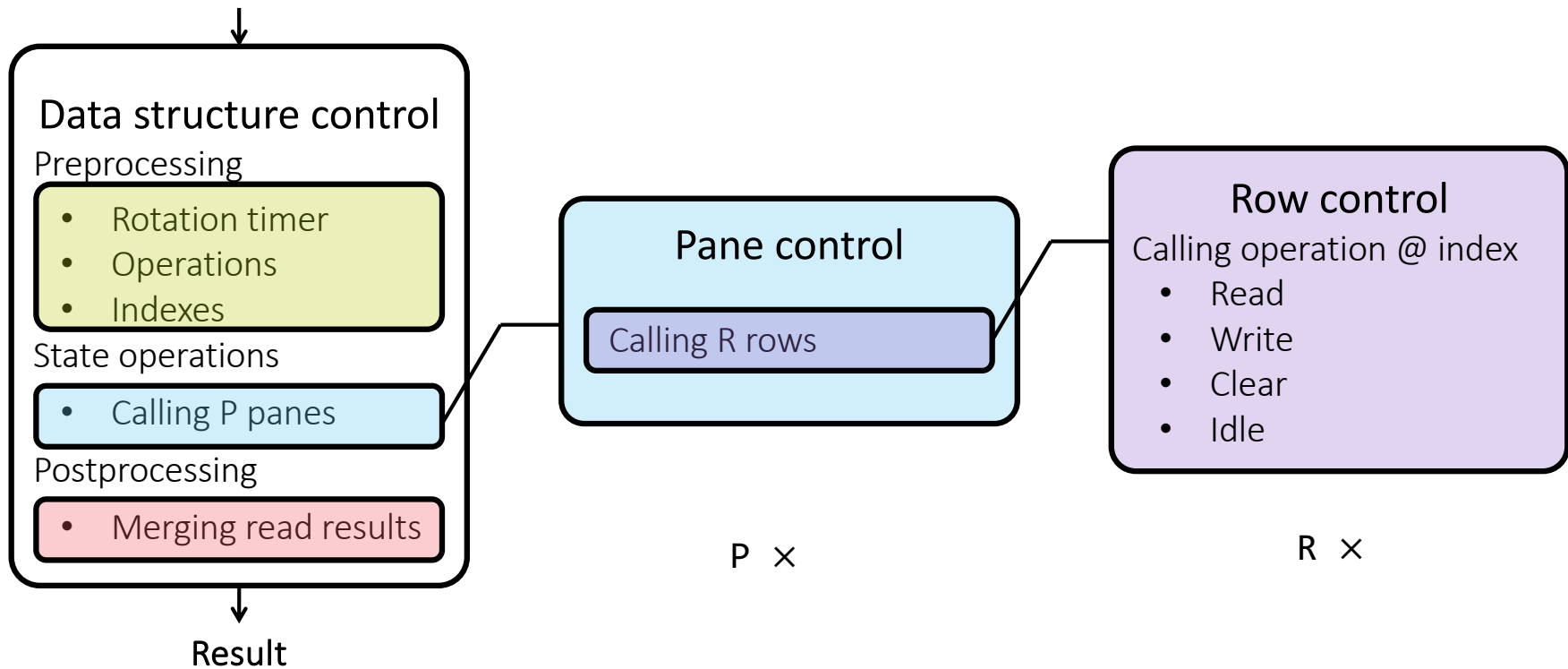
# Merging operation results

- **Pre-processing** for deciding
  - Rotation timer
  - Operations
  - Indexes
- **State operations**: calling operation @ the given index
- **Post-processing** for merging results



# A modular data structure template

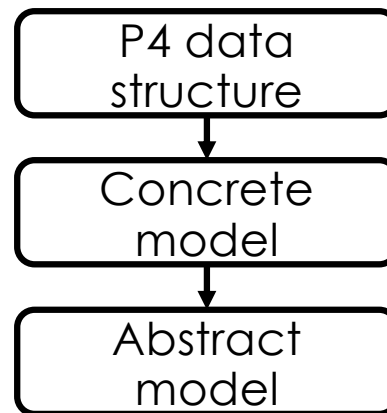
Timestamp, Method, Params



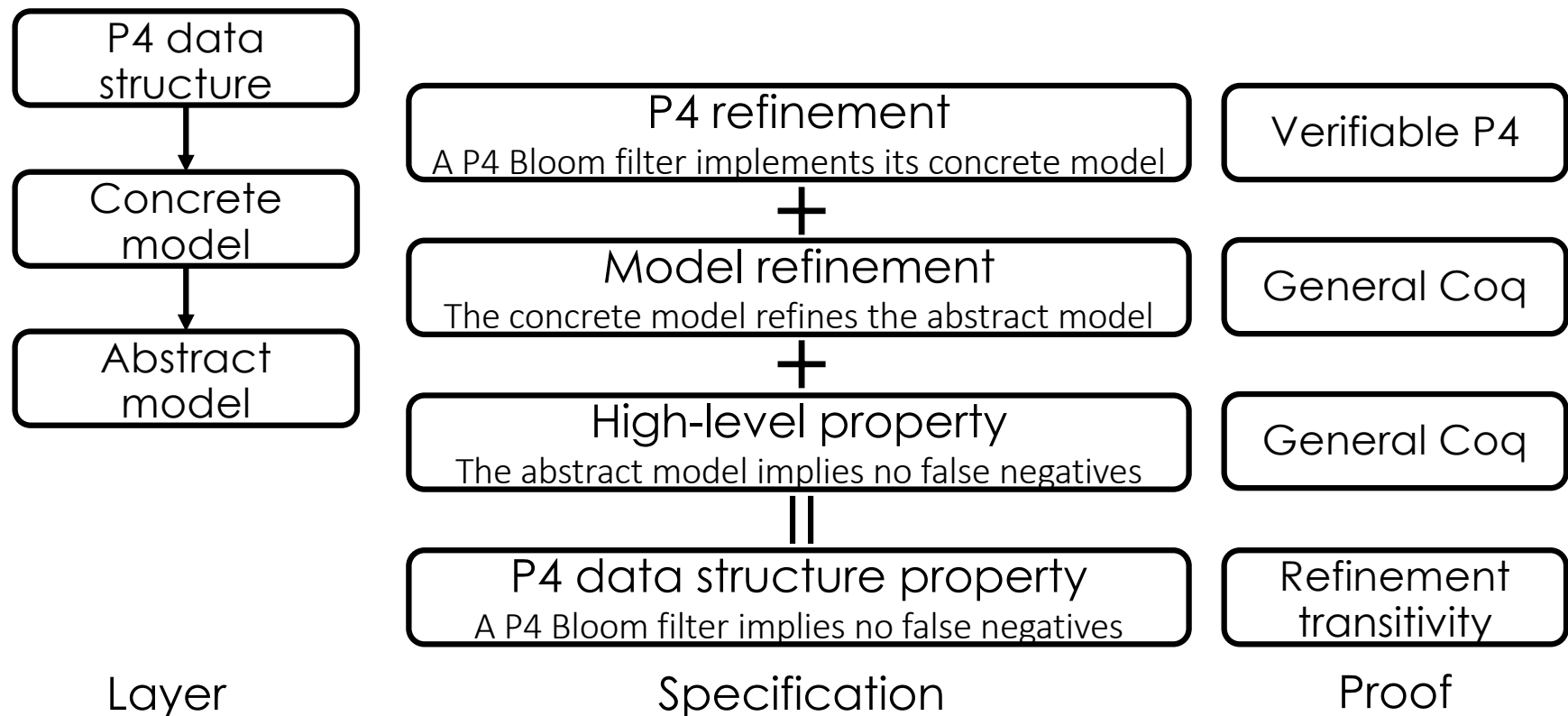
# Layered verification framework

**Layered verification framework:** Connects high-level specs to low-level P4 code through stepwise refinement.

- **Concrete functional model:** For P4 program verification
- **Abstract functional model:** For property verification



# Verification step by step



# Concrete model

**Concrete model:** a low-level functional model that closely mirrors P4 code.

- Modular structure:
  - rows, panes & data structures
  - add, query, clean
- Architecture-aware designs:
  - fixed-width state
  - rotation timer

```
Concrete model
Parameter (S R P step).
Definition row := listn bool S.
Definition pane := listn row R.
Record sbf := mk_sbf
{ sbf_panes : listn pane P;
  sbf_clean_index : Z;
  sbf_timer : bool * Z }
Definition update_timer ..
Definition sbf_add ..
Definition sbf_query ..
Definition sbf_clean ..
```

# Abstract model

**Abstract model:** a high-level functional model for property specification.

- Architecture-aware designs abstracted away
  - Explicit time window
  - Actual inserted elements
  - Abstract cleaning
- Validity assumption:
  - dense flow check

## No False Negative Property

For any valid abstract `sbf`, if an element is added at time `t`, then querying the `sbf` for that element at any time `t'` within the **window length lower bound** returns true.

## Abstract model

```
Parameter (S R P step).  
Definition sbf := option sbf_core.  
Record sbf_core := mk_sbf  
{ sbf_panes : list (list Element)  
  time_next_step : Z;  
  time_last_clean : Z;  
  num_clean : Z }.  
  
Definition packet_arrives ..  
Definition sbf_add ..  
Definition sbf_query ..  
Definition sbf_clean ..
```

# Advantages of modular synthesis

**Generalizability:** systematically supporting many data structures

**Proof organization:** reusable verification of rows and panes

**Code readability:** making P4 code easier to read, learn, and modify



# Advantages of layered verification

**Generalizability:** systematically supporting many data structures

**Maintainability:** localized verification efforts

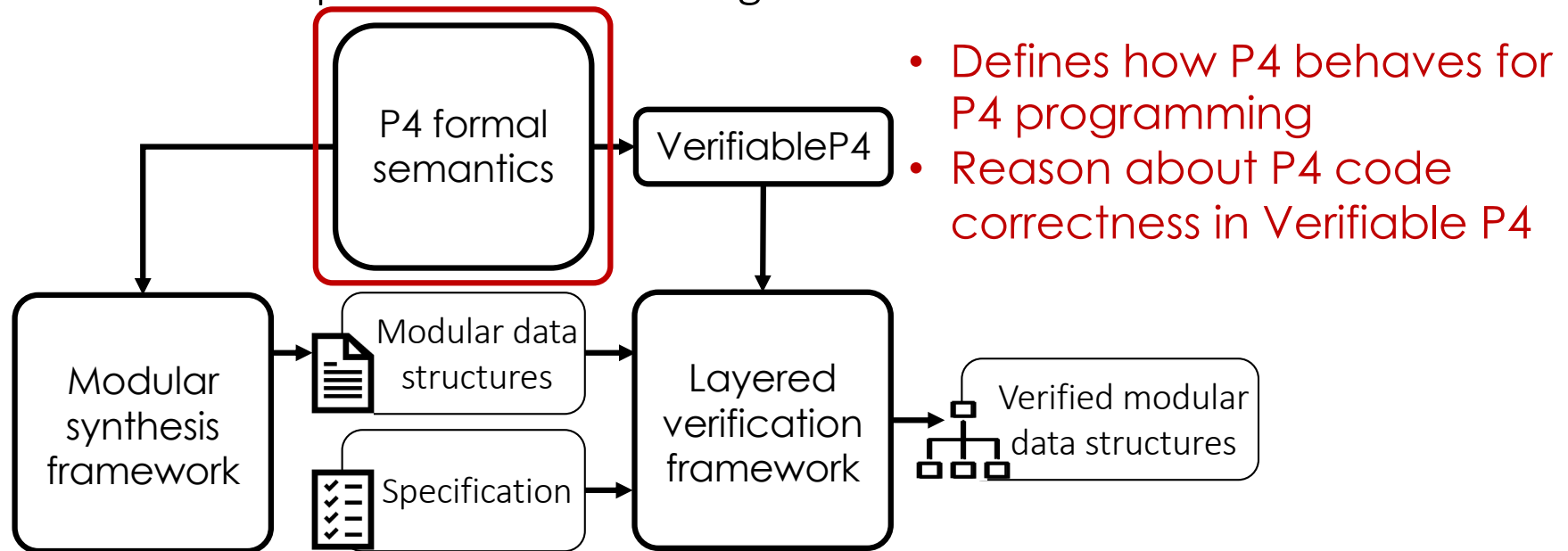
- High-level properties
- Functional models
- Implementation strategies
- Target architecture

**Separation of expertise:** facilitating collaboration

- Data structure expertise
- P4 expertise

# What is missing?

So far, we've built frameworks for synthesis and verification. But one essential component is still missing: **a formal semantics for P4.**



# Outline

Motivations, challenges & contributions

Network Approximate Programming

Verified modular data structures

**P4 formal semantics**

Conclusions & future directions

# Laying semantic foundation for P4

**Problem:** P4 lacks rigorous, mechanized semantics, leaving developers with ambiguous, informal specifications for programming & verification.

**Limitations:** Petr4 semantics underlies a **P4 interpreter** — an executable model of programs.

	<b>Petr4 semantics</b>
<b>Main goal</b>	Faithful to program execution
<b>Mechanization</b>	Pen-and-paper
<b>Nondeterminism</b>	Single execution outcome
<b>Design choices</b>	Borrows from functional languages

# P4 formal semantics

**Problem:** P4 lacks rigorous, mechanized semantics, leaving developers with ambiguous, informal specifications for programming & verification.

**Limitations:** Petr4 semantics underlies a **P4 interpreter** — an executable model of programs.

**Approach:** Our semantics underlies **Verifiable P4** — an interactive verification system.

	<b>Petr4 semantics</b>	<b>Our semantics</b>
<b>Main goal</b>	Faithful to program execution	Captures specs & data-plane behavior
<b>Mechanization</b>	Pen-and-paper	Mechanized in Coq
<b>Nondeterminism</b>	Single execution outcome	All possible execution outcomes
<b>Design choices</b>	Borrows from functional languages	Domain-specific semantics

# Domain-specific semantics

**Insight:** Bounded by **inherent constraints** of programmable data planes, P4 is a **domain-specific language**; so is our **semantics**.

- To allocate the scarce hardware resources optimally, P4 requires **static allocation during compilation**, leading to **a two-phase semantics**.
- Given the finite number of stages, P4 has **no recursion or loop constructs**, leading to **a big-step operational semantics**.
- Given the architecture-dependent stateful behavior, P4 is **a target-specific language**, leading to **target-specific state semantics modules**.

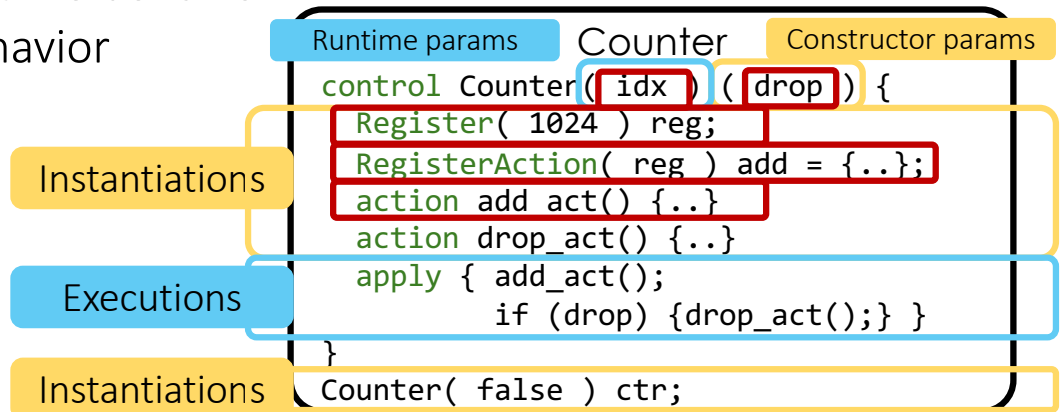
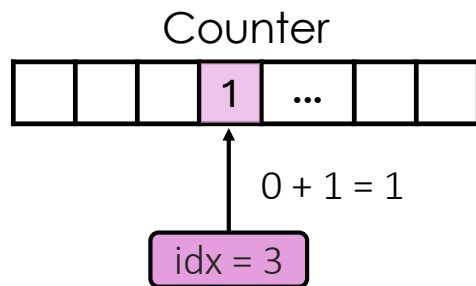
# Two-phase semantics

P4 compiler allocates data plane resources **statically** during instantiation:

- To optimize resource utilization
- To ensure high throughput

P4 language can be naturally split into two phases; so is P4 semantics:

- **Instantiation phase:** compile-time behavior
- **Execution phase:** runtime behavior



# Two-phase semantics

P4 compiler allocates data plane resources **statically** during instantiation:

- To optimize resource utilization
- To ensure high throughput

P4 language can be naturally split into two phases; so is P4 semantics:

- Instantiation phase: compile-time behavior
  - Static locations: decide where information live
  - Static initialization: fill locations with initial runtime information
  - Static instantiation: fill locations with compile-time known information
- Execution phase: runtime behavior

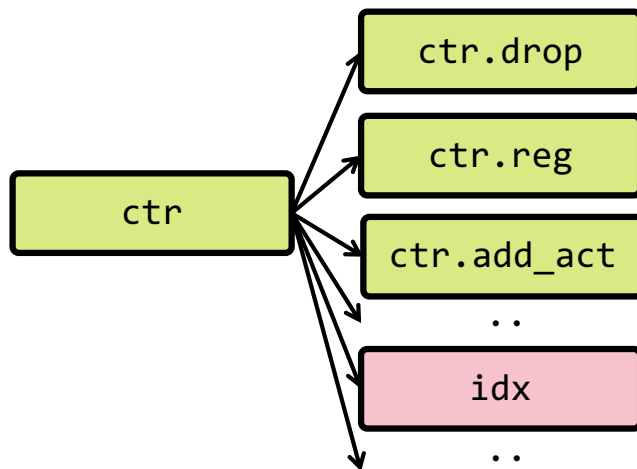
Phase separation leads to a simple semantics faithful to P4 specifications.



# Instantiation: static locations

Instantiation phase assigns **static locations** to all P4 entities.

- **Globally unique path:** uniquely identify every P4 entity under a hierarchy.
- **Locally unique locator:** remove the common prefix for runtime variables.

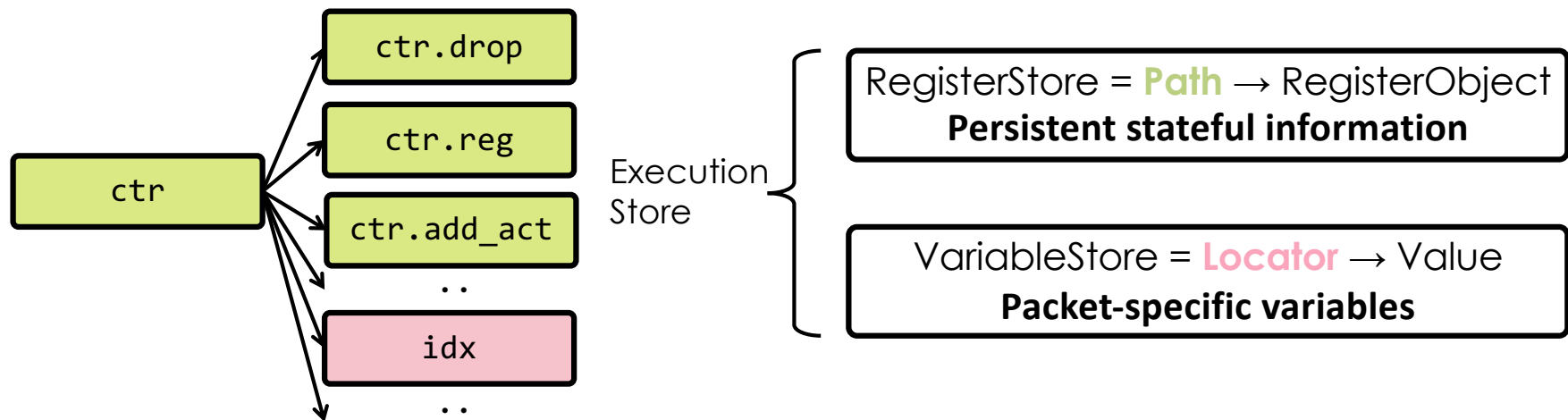


```
Counter
control Counter( idx ) ( drop ) {
  Register( 1024 ) reg;
  RegisterAction( reg ) add = {..};
  action add_act() {..}
  action drop_act() {..}
  apply { add_act();
          if (drop) {drop_act();} }
}
Counter( false ) ctr;
```

# Instantiation: static initialization

Instantiation phase **statically initializes** all registers.

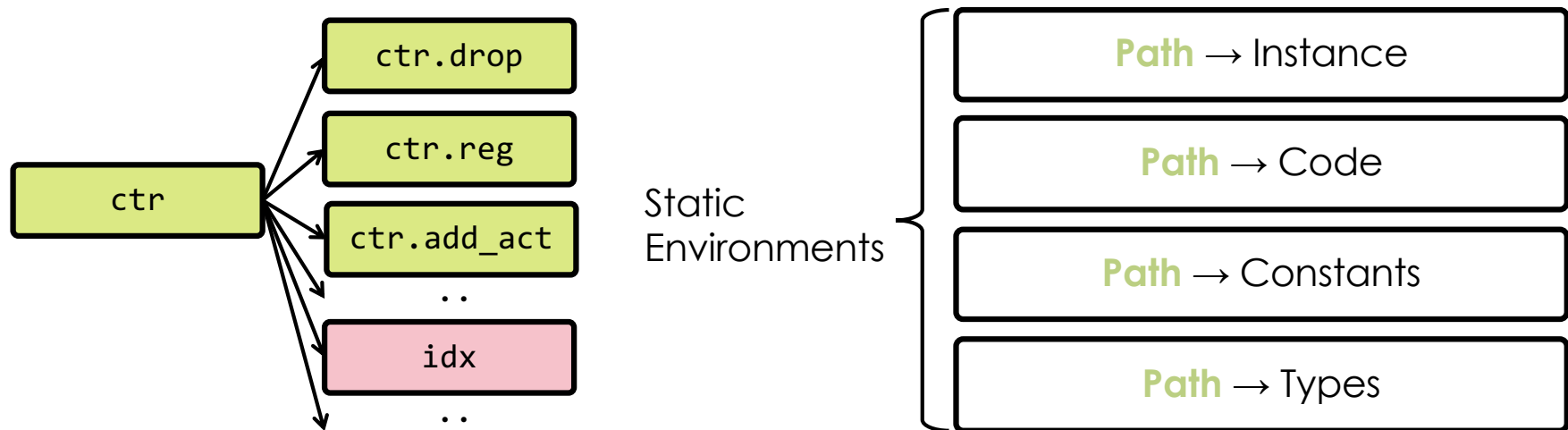
**Execution store** contains all runtime information.



# Instantiation: static instantiation

Instantiation phase **statically instantiates** all instances.

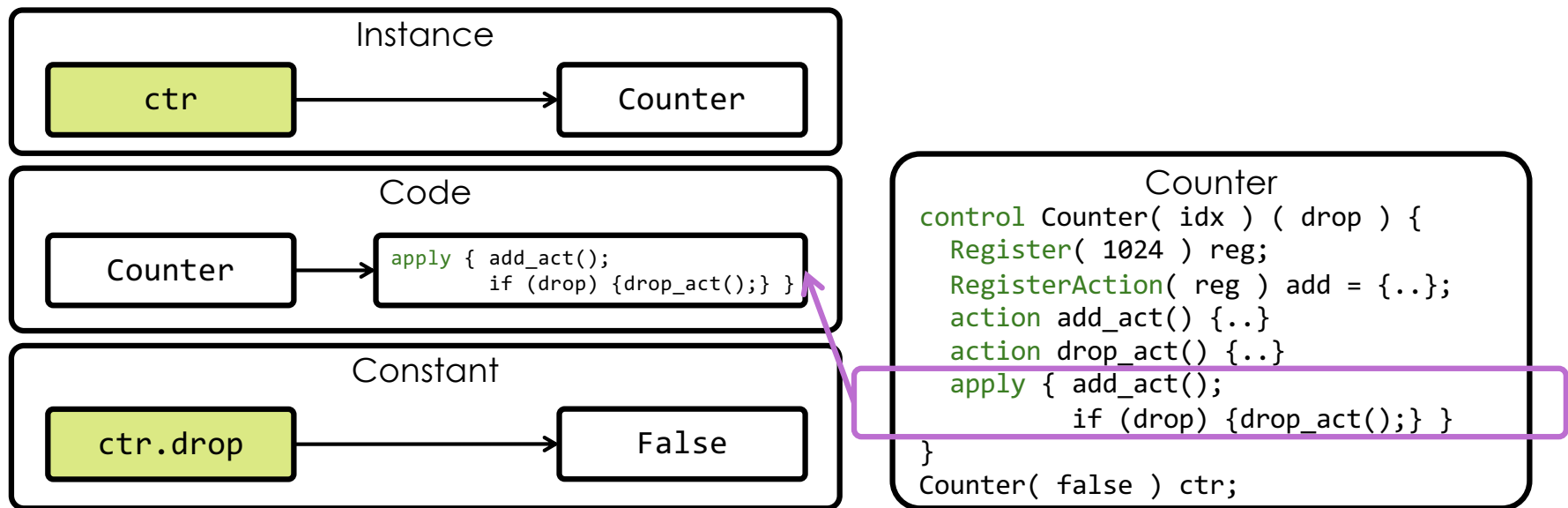
**Static environments** contains all compile-time known information.



# Instantiation: static instantiation

Instantiation phase **statically instantiates** all instances.

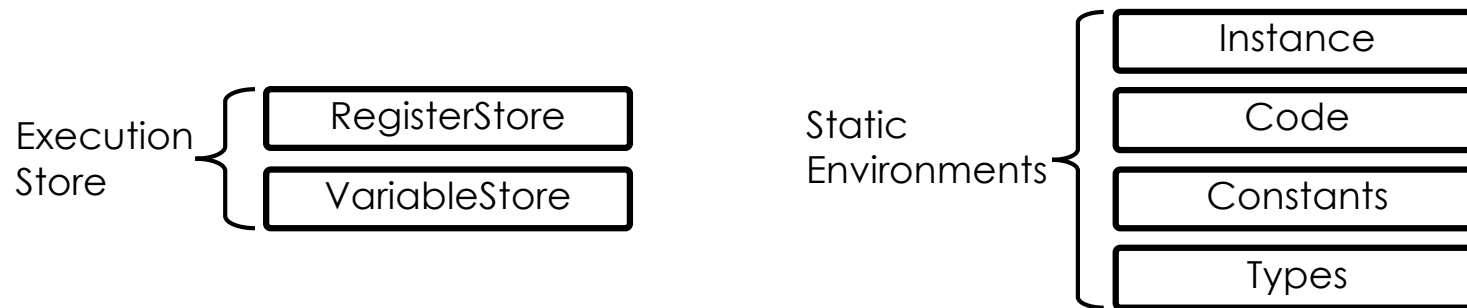
**Static environments** contains all compile-time known information.



# Instantiation phase summary

Instantiation phase evaluates **declarations** for:

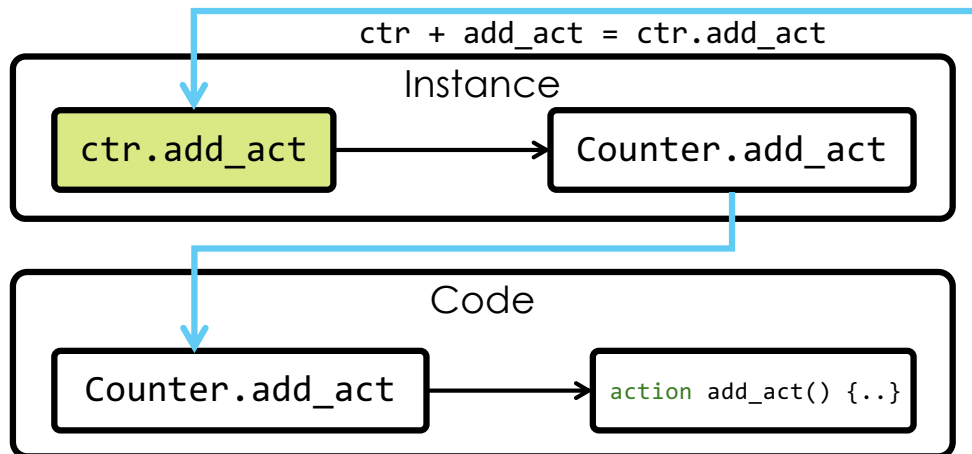
- **Static locations:** decide where information live
- **Static initialization:** fill locations with initial runtime information
- **Static instantiation:** fill locations with compile-time known information



Given initialized **execution store** & generated **static environments**, execution phase evaluates **statements** to simulate **runtime behavior**.

# Execution

Given initialized **execution store** & generated **static environments**, execution phase evaluates **statements** to simulate **runtime behavior**.



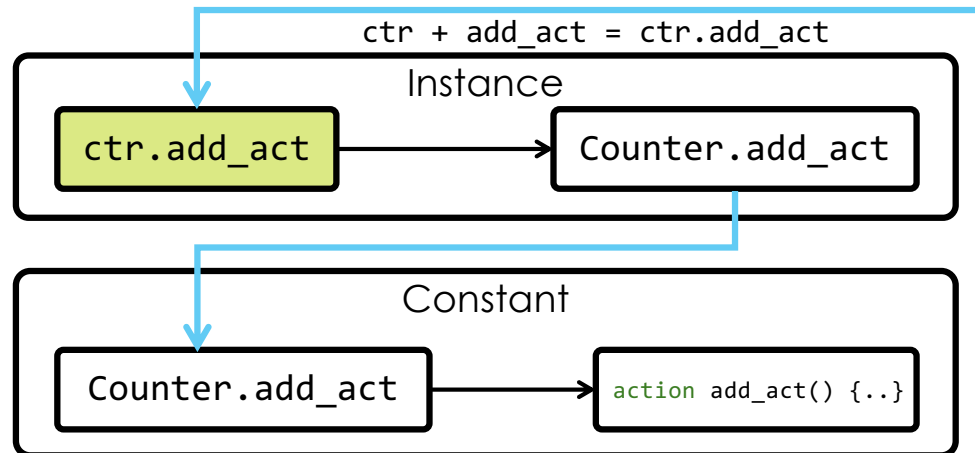
Semantic rule for method call

- **Lookup:** Globally unique path -> code
- **Execution:** Execute code definition

```
Counter
control Counter( idx ) ( drop ) {
  Register( 1024 ) reg;
  RegisterAction( reg ) add = {...};
  action add_act() {...}
  action drop_act() {...}
  apply { add_act();
          if (drop) {drop_act();} }
}
Counter( false ) ctr;
```

# Semantic rules

69 semantic rules formalize P4 behavior.



Semantic rule for method call

- **Lookup:** Globally unique path -> code
- **Execution:** Execute code definition

```
Counter
control Counter( idx ) ( drop ) {
  Register( 1024 ) reg;
  RegisterAction( reg ) add = {...};
  action add_act() {...}
  action drop_act() {...}
  apply { add_act();
          if (drop) {drop_act();} }
}
Counter( false ) ctr;
```

# Domain-specific semantics

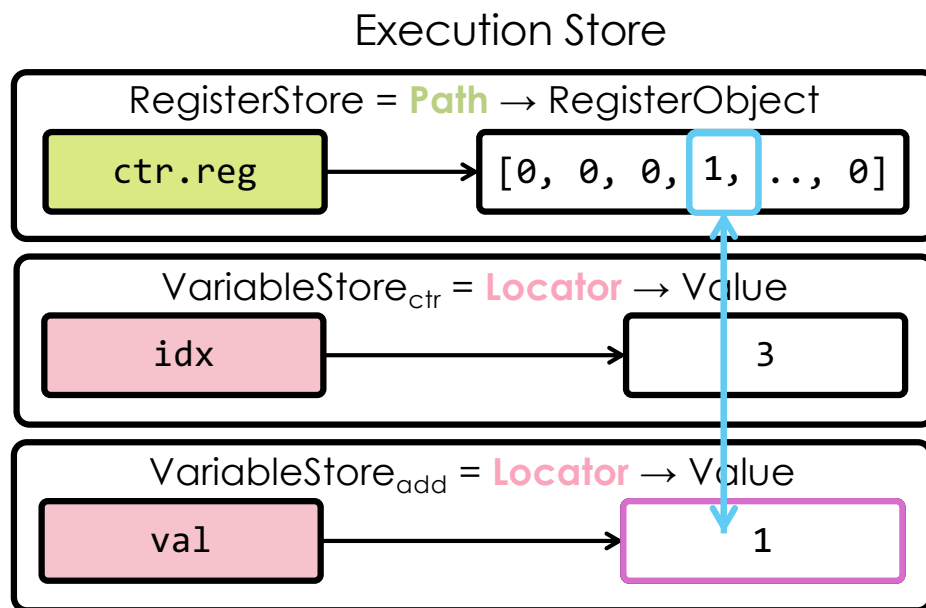
**Insight:** Bounded by **inherent constraints** of programmable data planes, P4 is a **domain-specific language**; so is our **semantics**.

- To allocate the scarce hardware resources optimally, P4 requires **static allocation during compilation**, leading to **a two-phase semantics**.
- Given the finite number of stages, P4 has **no recursion or loop constructs**, leading to **a big-step operational semantics**.
- Given the architecture-dependent stateful behavior, P4 is **a target-specific language**, leading to **target-specific state semantics modules**.



# Target-specific state semantics

State semantics is **target-specific**.



Semantic rule for register action

- Read: read `reg[idx]` into `val` in `StackFrameadd`.
- Modify: execute the user-defined `apply` method.
- Write: write `val` to `reg[idx]` in `RegisterStore`.

```
Counter
control Counter(idx ) ( drop ) {
  Register( 1024 ) reg;
  RegisterAction( reg ) add = {
    void apply( val ) {
      value = value + 1; } };
  action add_act() {
    add.execute(key); }
  .. }
Counter( false ) ctr;
```

# Domain-specific semantics

**Insight:** Bounded by **inherent constraints** of programmable data planes, P4 is a **domain-specific language**; so is our **semantics**.

- To allocate the scarce hardware resources optimally, P4 requires **static allocation during compilation**, leading to **a two-phase semantics**.
- Given the finite number of stages, P4 has **no recursion or loop constructs**, leading to **a big-step operational semantics**.
- Given the architecture-dependent stateful behavior, P4 is **a target-specific language**, leading to **target-specific state semantics modules**.

# Domain-specific semantics

**Insight:** Bounded by **inherent constraints** of programmable data planes, P4 is a **domain-specific language**; so is our **semantics**.

- To allocate the scarce hardware resources optimally, P4 requires **static allocation during compilation**, leading to **a two-phase semantics**.
- Given the finite number of stages, P4 has **no recursion or loop constructs**, leading to **a big-step operational semantics**.
- Given the architecture-dependent stateful behavior, P4 is **a target-specific language**, leading to **target-specific state semantics modules**.

# Debugging language specifications

- Rigorous formalization of domain-specific P4 semantics uncovers **ambiguities, errors, and inconsistencies** in specifications & compilers.
  - 23 issues discussed with the P4 language design working group.
  - 17 fixes adopted.

# Advantages of P4 formal semantics

- Rigorous formalization of domain-specific P4 semantics uncovers **ambiguities, errors, and inconsistencies** in specifications & compilers.
  - 23 issues discussed with the P4 language design working group.
  - 17 fixes adopted.
- Many issues could have been avoided with **formal semantics** when designing the language.

Category	Issues	Status
Expression	Concatenation is missing from the operations on the bit type.	Released
Instantiation	Instantiation should not be a statement.	Released
Table	Default action should be set as NoAction when undefined.	Released

# Advantages of P4 formal semantics

- Rigorous formalization of domain-specific P4 semantics uncovers **ambiguities, errors, and inconsistencies** in specifications & compilers.
  - 23 issues discussed with the P4 language design working group.
  - 17 fixes adopted.
- P4 faces challenges in balancing language **generality** and **domain-specific focus**.

Category	Issues	Status
Expression	Implicit conversions of lists, tuples, structs & headers are not specified.	Stalled
Function	Abstract extern methods open multiple back doors, e.g., allowing recursion and accessing nonlocal variables.	Stalled
Name	Name duplication and name shadowing are undefined.	Stalled

# Outline

Motivations, challenges & contributions

P4 formal semantics

Verifiable modular data structures

Network Approximate Programming

Conclusions & future directions

# Conclusions

To realize verifiable traffic control in the data plane, we present

- **Network Approximate Programming Language**  
Automating data structure selection and sizing
- **Verifiable Modular Data Structures**  
Hardware-compliant data structures with correctness guarantees
- **Formal Semantics for P4**  
Building a solid foundation for P4 programming & reasoning



# Looking ahead

## **Broader impact:**

- Promote abstractions for programmable networks.
- Advance verification for real-world P4 programs.
- Bridge the gap between programming languages and network control.

## **Future directions:**

- Broaden NAP for richer dictionary classes and optimization strategies.
- Integrate with distributed network control.
- Extend verification to more targets and data structures.

# Thank you so much!



Backup slides

# Programming state in P4

P4 is a domain-specific language for expressing packets processing on the programmable data planes.

- Low-level and complicated
- Specialized language constructs
- Ambiguous and buggy specification

Q: What does a P4 program mean?

Q: What is the default action by default?

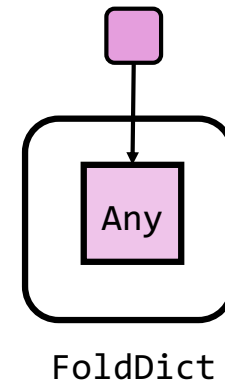
A: Not in the specification.

Match-Action Table

```
table routing {  
  key = { hdr.ipv4.dstAddr : lpm; }  
  actions = { drop; route; }  
  const entries = {..}  
  size = 2048;  
}
```

# Value state machine in dictionaries

- **Key:** flow identifier
- **Value:** stateful information
- **Operations:**
  - Create<key>(parameters)
  - Add(key)
  - Query(key)
- **Dictionary Class:** value updates
- **Parameters:**
  - **Error direction:** inclusion approximation
  - **Time window:** temporal approximation
  - **Value state machine**



# Compiler: configure time window

- **Goal:**

- `within(lo, hi)`:  
Sliding window of length  $\in [lo, hi]$
- `since(intv)/last(int)`:  
Tumbling windows are degenerate cases.

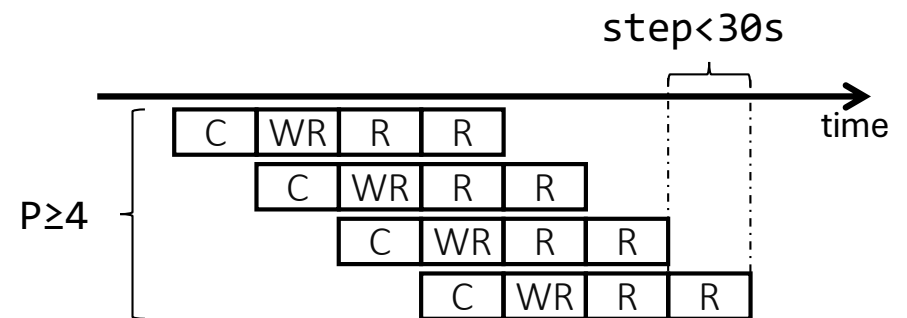
- **Synthesis framework:**

- A sliding window of length  $\in [(P-2) \cdot \text{step}, (P-1) \cdot \text{step}]$  ( $P \geq 2$ )

- **Time constraints:**

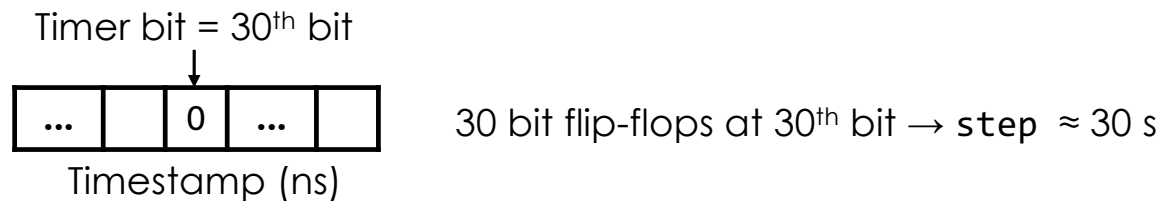
- $P \geq lo/(hi - lo) + 2$

```
type key = {int eip; int iip}  
ExistDict<key> IDset =  
    ExistDict.create (over,  
        within(sec(60), sec(90)),  
        ExistDict())
```



# Data structure pipeline

- **Pre-processing** for deciding
  - Rotation timer:
    - Time window length  $\in [(P-2) \cdot \text{step}, (P-1) \cdot \text{step}]$  ( $P \geq 2$ )
    - Supports tumbling and sliding windows
    - Flexible step size



# Dense flow

- We need a “dense enough” packet flow to properly clean the state:
  - Catch all bit flops at the timer bit to **rotate panes** on time.
  - **Increment cleaning index** to clean a pane completely.
- We use a **packet generator** to maintain the minimum packet rate.

$$\text{Rate (pkt/ns)} = \max(1/2^{\text{tb\_pos}}, S/\text{step})$$



# Concrete model

**Concrete model:** low-level functional model that closely mirrors P4 code.

- The concrete model, defined in Coq, is **fully parameterized**.
- The data structure, defined in P4, **hardcode** parameters.

## P4 data structure

```
control Row( key ) {  
  Register( S ) reg;  
  RegisterAction( reg ) add = {..};  
}  
control Pane( key ) {  
  Row() row_1;  
  Row() row_2;  
}  
control SBF( key ) {  
  Pane() pane_1;  
  Pane() pane_2;  
}
```

## Concrete model

```
Parameter (S R P step).  
Definition row := listn bool S.  
Definition pane := listn row R.  
Record sbf := mk_sbf  
  { sbf_panes : listn pane P;  
    sbf_clean_index : Z;  
    sbf_timer : bool * Z }  
  
Definition update_timer ..  
Definition sbf_add ..  
Definition sbf_query ..  
Definition sbf_clean ..
```

# Verification-aware programming

VerifiableP4 allows proving properties of approximate data structures.

- Verification takes efforts.
- P4 program should be amenable to verification.

## Modular

```
control Row( key ) {  
  ..  
}  
control BloomFilter( key ) {  
  Row() row_1;  
  Row() row_2;  
}  
control Ingress(..) {  
  BloomFilter() bf;  
}  
Switch(ig = Ingress()) main;
```

## Flattened

```
control Ingress(..) {  
  Register( 1024 ) bf_row_1_reg;  
  RegisterAction( bf_row_1_reg ) add = {..};  
  ..  
  Register( 1024 ) bf_row_1_reg;  
  RegisterAction( bf_row_1_reg ) add = {..};  
  ..  
}  
Switch(ig = Ingress()) main;
```

# Proof for P4 refinement

Semi-modular verification:

- Specifications & proofs for a row can be replayed for all row instances.
- Specifications & proofs for a pane can be replayed for all pane instances.

Verifying a sliding-window Bloom filter (LoC)

Object	P4 code	Concrete functional model	Function spec.	P4 proof
Row	53	85	165	140
Pane	22	62	235	140
Filter	341	333	858	1579

# Abstract model

**Abstract model:** high-level functional model for property specification.

## Abstract model

```
Parameter (S R P step).
Definition sbf := option sbf_core.
Record sbf_core := mk_sbf
{ sbf_panes : list (list Element)
  time_next_step : Z;
  time_last_clean : Z;
  num_clean : Z }.

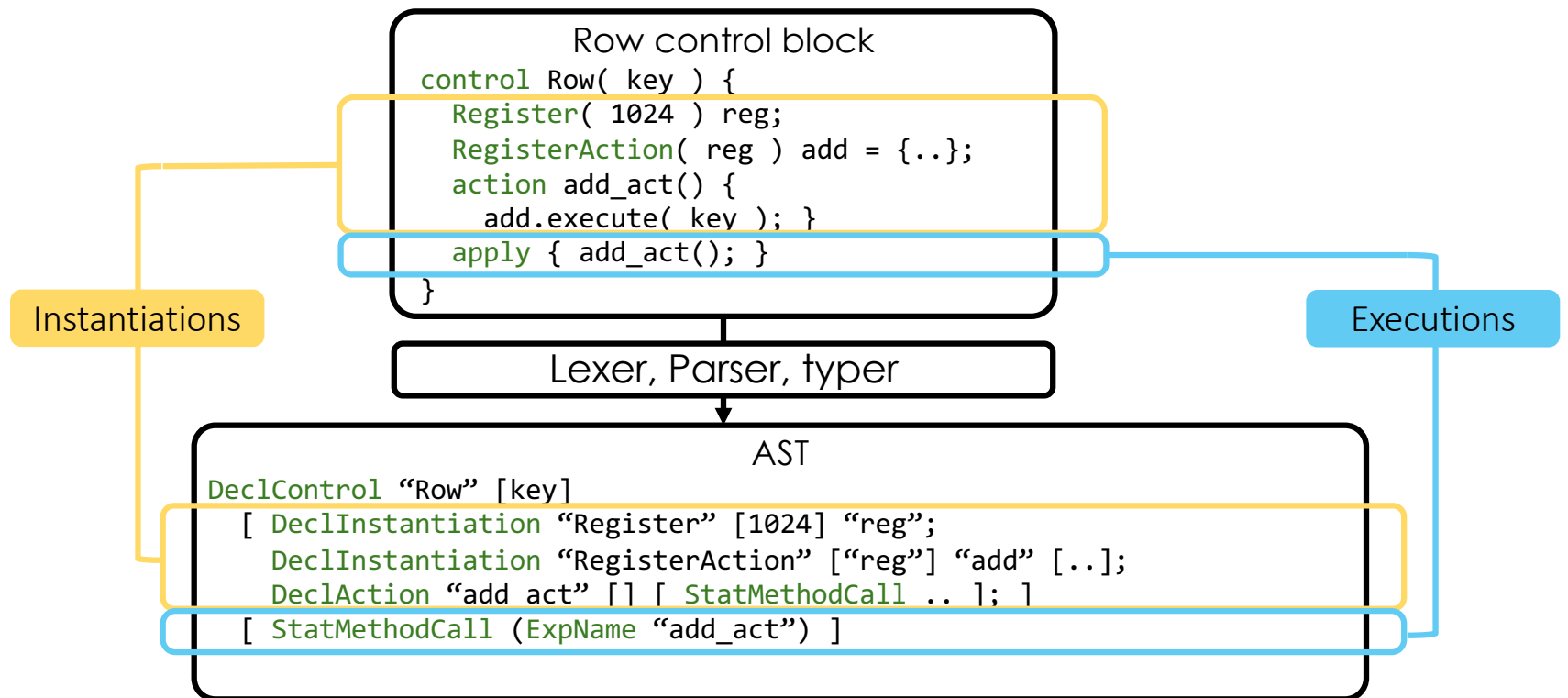
Definition packet_arrives ..
Definition sbf_add ..
Definition sbf_query ..
Definition sbf_clean ..
```

## No False Negative Property

```
Definition window_lo := (P-2)*step.
Lemma no_false_neg_lemma :
  forall sbf t t' e,
    valid_by sbf t ->
      t <= t' <= t + window_lo ->
        sbf_query (sbf_add sbf t e) t' e = true.
```

For any valid abstract `sbf`, if an element `e` is added at time `t`, then querying the `sbf` for that element at any time `t'` within the window length lower bound returns true.

# Abstract syntax tree



# Why not Petr4's solution?

Petr4 intermingles instantiation & execution.

- **Local environment:** name -> location
- **Global store:** location -> value & closures
- **Closure:** local environment + code definition

```
Env_ctr = [ "drop" → 200, "idx" → 400, ...,  
            "add act" → 600 ]  
Store = [ 200 → false, 400 → 3, ..., Values  
Closures 600 → clos(Env_add_act, Code_add_act) ]
```

```
Counter  
control Counter( idx ) ( drop ) {  
  Register( 1024 ) reg;  
  RegisterAction( reg ) add = {..};  
  action add_act() {..}  
  action drop_act() {..}  
  apply { add_act();  
          if (drop) {drop_act();} }  
}  
Counter( false ) ctr;
```

# Why not Petr4's solution?

Petr4 intermingles instantiation & execution.

- **Local environment:** name -> location
- **Global store:** location -> value & closures
- **Closure:** local environment + code definition

Env\_ctr

Store =

Closures

Dynamic locations:

- Keeping track of fresh locations
- Adding a layer of indirection

```
{  
.  
};  
  
} }
```

# Why not Petr4's solution?

Petr4 intermingles instantiation & execution.

- **Local environment:** name -> location
- **Global store:** location -> value
- **Closure:** local environment + code definition

```
Env_ctr = [ "drop" → 200, "idx" → 400, ...,  
            "reg" → "ctr.reg", Register  
            "add_act" → 600 ]  
Store = [ 200 → false, 400 → 3, ...,  
          "ctr.reg" → [0, 0, .., 0], Initialization @ 1st packet  
          600 → clos(Env_add_act, Code_add_act) ]
```

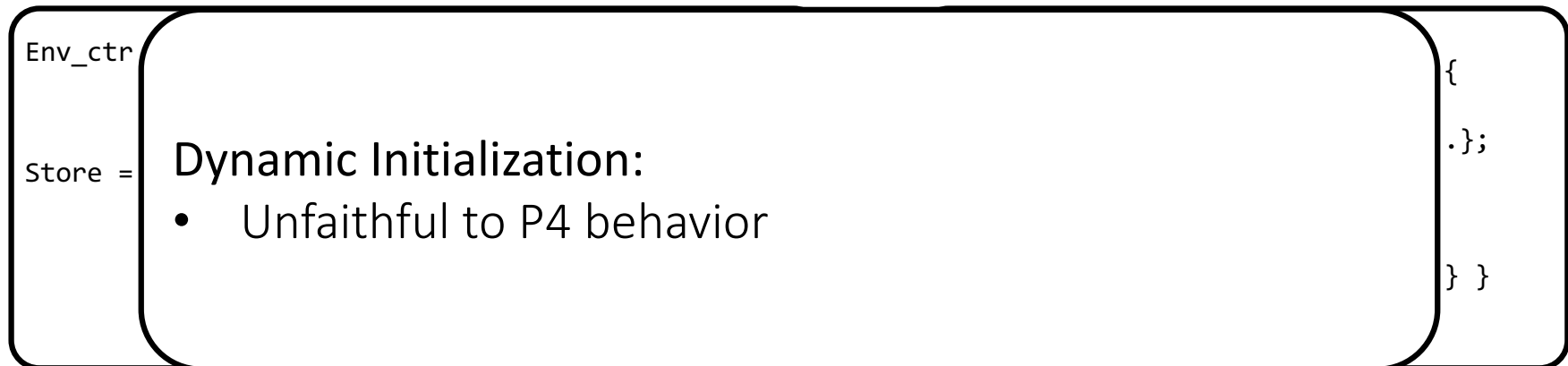
```
Counter  
control Counter( idx ) ( drop ) {  
  Register( 1024 ) reg;  
  RegisterAction( reg ) add = {..};  
  action add_act() {..}  
  action drop_act() {..}  
  apply { add_act();  
          if (drop) {drop_act();} }  
}  
Counter( false ) ctr;
```



# Why not Petr4's solution?

Petr4 intermingles instantiation & execution.

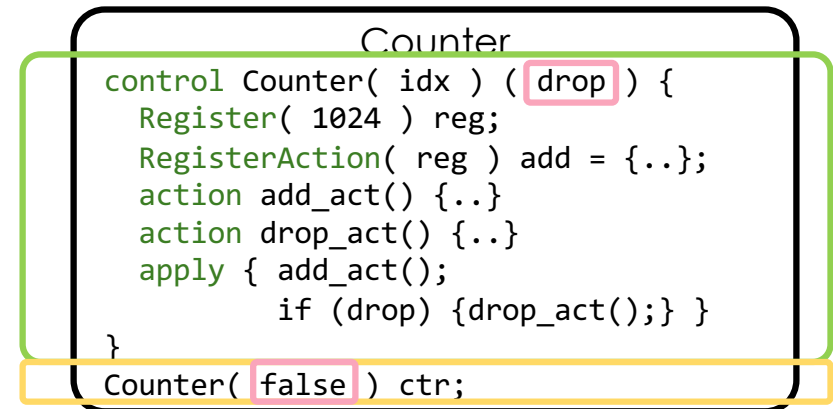
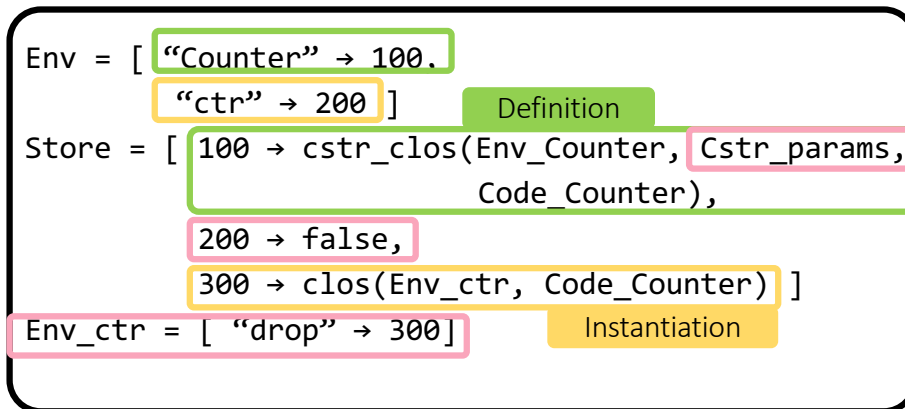
- **Local environment:** name -> location
- **Global store:** location -> value
- **Closure:** local environment + code definition



# Why not Petr4's solution?

Petr4 intermingles instantiation & execution.

- **Local environment:** name -> location
- **Global store:** location -> value
- **Closure:** local environment + code definition



# Why not Petr4's solution?

Petr4 intermingles instantiation & execution.

- **Local environment:** name -> location
- **Global store:** location -> value
- **Closure:** local environment + code definition



# Why not Petr4's solution?

Petr4 borrows from functional languages:

- **Local environment**: name -> location
- **Global store**: location -> value
- **Closure**: local environment + code definition

Petr4 mixes instantiation with execution, adding unnecessary complexity:

- **Dynamic** locations
- **Dynamic** initialization
- **Dynamic** instantiation



These are all **static** in our P4 semantics, happening in the **instantiation phase**.

# Takeaway: why two phases?

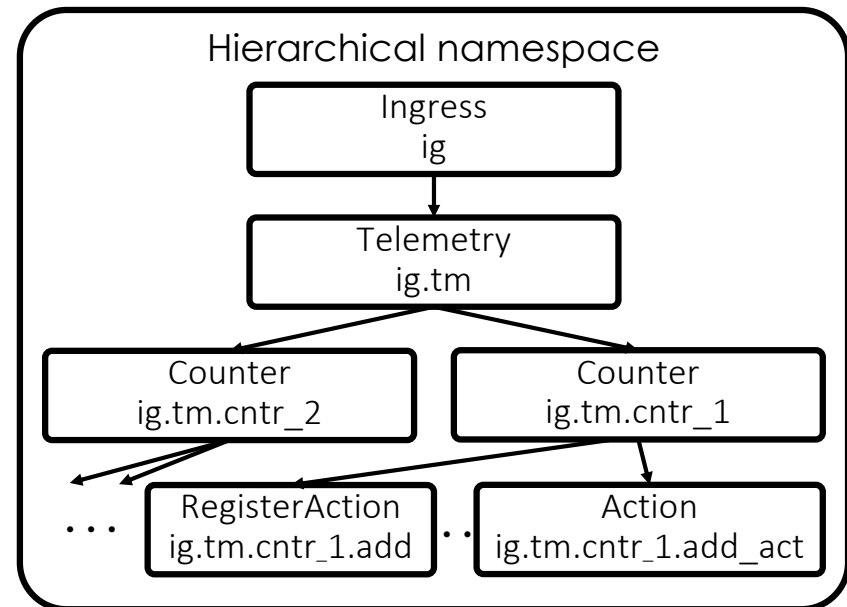
- VerifiableP4's semantics is built upon Petr4, but with phase distinction.
  - **Static environment:** path  $\rightarrow$  static object
  - **Program state:** path  $\rightarrow$  value/register object
- Petr4's semantics mixes two phases:
  - **Global storage:** dynamic locations  $\rightarrow$  values
  - **Local environment:** currently visible names  $\rightarrow$  dynamic locations
- Benefits of two phases:
  - Faithful representation of static compilation behavior in P4 specifications.
  - Straightforward stateful semantics & reasoning

# Instantiation: static locations

Instantiation phase generates a **static environment** mapping from **globally unique paths** to **static instances/code/values/types**.

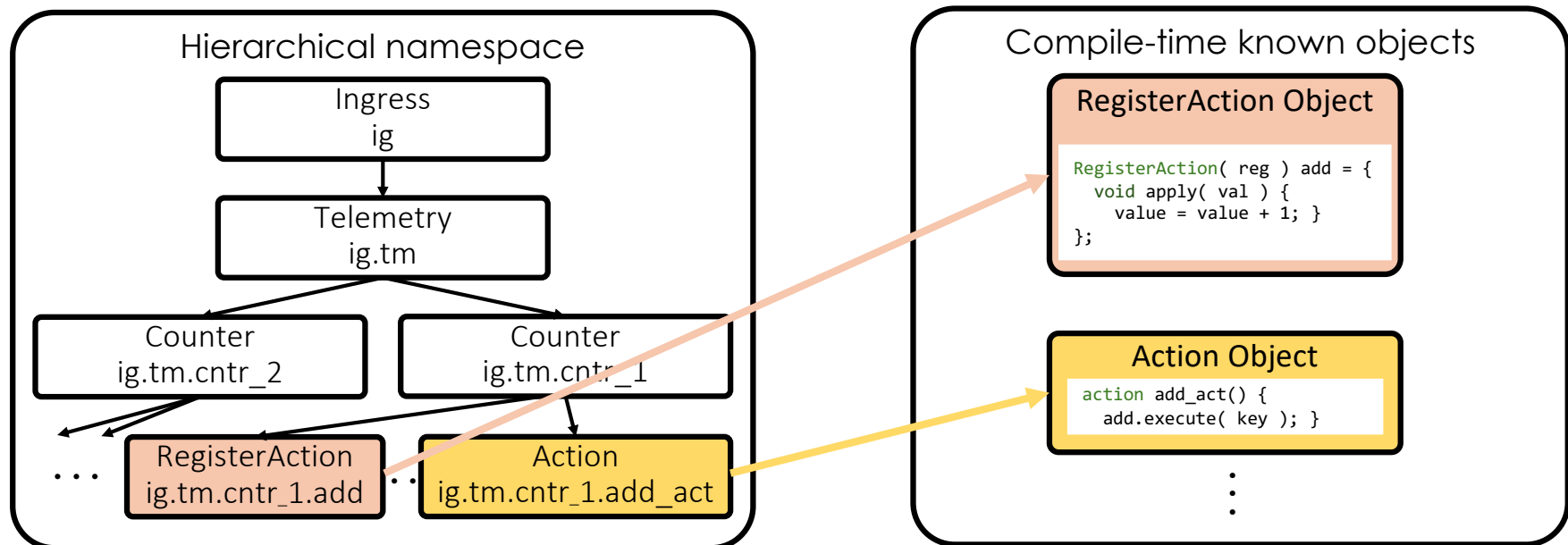
Instantiations in a telemetry system

```
control Counter( key ) ( drop ) {  
  Register( 1024 ) reg;  
  RegisterAction( reg ) add = {..};  
  action add_act() {..}  
  action drop_act() {..}  
}  
control telemetry( key ) {  
  Counter( false ) ctr_tcp;  
  Counter( true ) ctr_udp;  
}  
control Ingress(..) {  
  telemetry() tm;  
}
```



# Instantiation

Instantiation phase generates a **static environment** mapping from **globally unique paths** to **static instances/code/values/types**.



# Program logic: specification

- Program logic provides a formal system to **specify** and **verify** program properties based on semantics.
- Simplified **functional model** of Row: SRow
- **Function specification** of row\_1.apply(key):
  - **Precondition:** row\_1 represents srow
  - **Postcondition:** row\_1 represents srow after key is inserted

```
PATH row_1 MOD Null [row_1]
WITH (srow : SRow) (key : Z) (_:  $0 \leq \text{key} < \text{num\_slots}$ ),
  PRE (ARG [key], MEM [ ], EXT [row_repr row_1 srow])
  POST (RET Null, ARG [ ], MEM [ ], EXT [row_repr row_1 (srow_insert srow key)])
```



# Program logic: verification

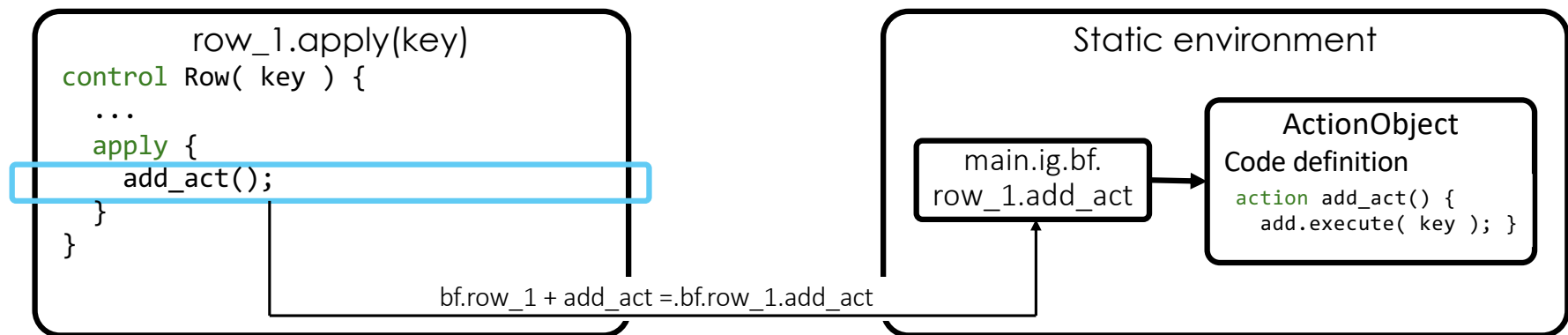
- Program logic provides a formal system to **specify** and **verify** program properties based on semantics.
- To prove that `row_1.apply(key)` satisfies its **function specification**:
  - Create a symbolic program state as described by the precondition;
  - Apply **program logic rules** in forward mode;
  - Proves the resulting program state implies the postcondition.
- Example program logic rule for assignment:

$$\frac{\text{semantics rule for expression} \quad \Gamma, p, \vec{P} \vdash \text{exp} \Downarrow v}{\Gamma, p \vdash \underbrace{\{\text{MEM } \vec{P}, \text{EXT } \vec{Q}\}}_{\text{precondition}} \underbrace{x@(\text{inst } p') := \text{exp}}_{\text{assignment statement}} \underbrace{\{\text{MEM } \vec{P}[p' \rightarrow v], \text{EXT } \vec{Q}\}}_{\text{postcondition}}}$$

# Execution

Execution phase simulates **run-time behavior** over **program state** based on **static environment**.

- Semantic rule for method call:
  - Current path + object name = fully qualified name
  - Static environment: fully qualified name -> object



# Execution

Execution phase simulates **run-time behavior** over **program state** based on **static environment**.

- ProgramState := StackFrame × RegisterStore
- **Packet-specific variables:** StackFrame := Path → Value
- **Persistent stateful information:** RegisterStore := Path → RegisterObject

Execution of a row instance

```
control Row( key ) {  
  ...  
  apply {  
    add_act();  
  }  
}
```

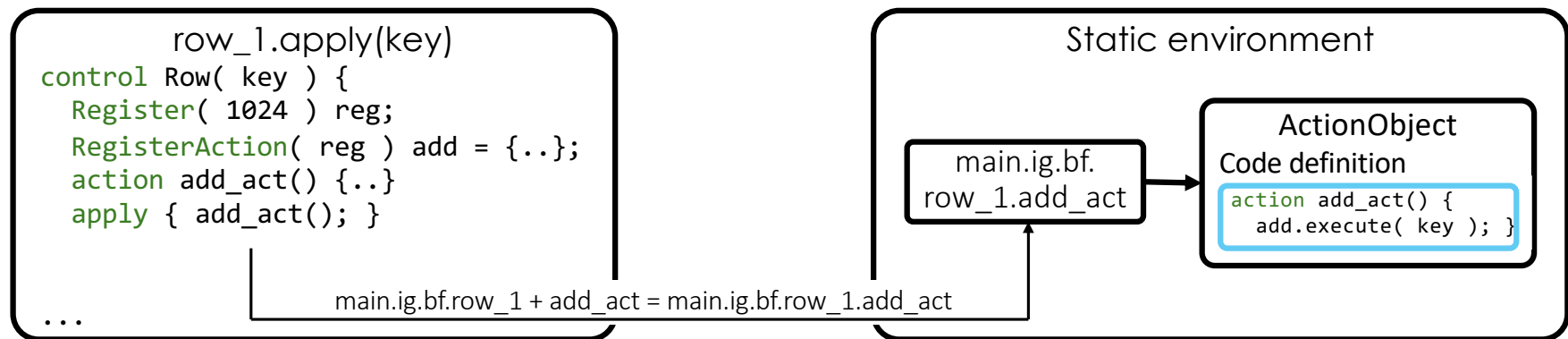
Program state

```
StackFrame := [“main.ig.bf.row_1.key” → 3]  
RegisterStore := [“main.ig.bf.row_1.reg” →  
  [0, 1, 0, 0, ..., 0, 0] ]
```

# Execution

Execution phase simulates **run-time behavior** over **program state** based on **static environment**.

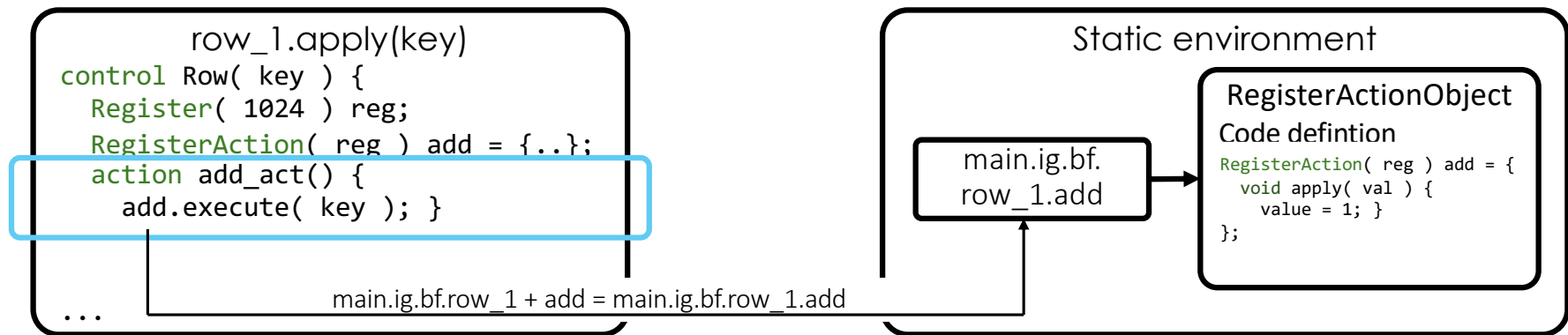
- Semantic rule for method call:
  - Current path + object name = fully qualified name
  - Static environment: fully qualified name -> object
  - Execute the object over the ProgramState



# Execution

Execution phase simulates **run-time behavior** over **program state** based on **static environment**.

- Semantic rule for method call: ← Rules can be recursively used.
  - Current path + object name = fully qualified name
  - Static environment: fully qualified name -> object
  - Execute the object over the ProgramState



# Execution

Execution phase simulates **run-time behavior** over **program state** based on **static environment**.

- Semantic rule for method call:
  - Current path + object name = fully qualified name
  - Static environment: fully qualified name -> object
  - Execute the object over the ProgramState

