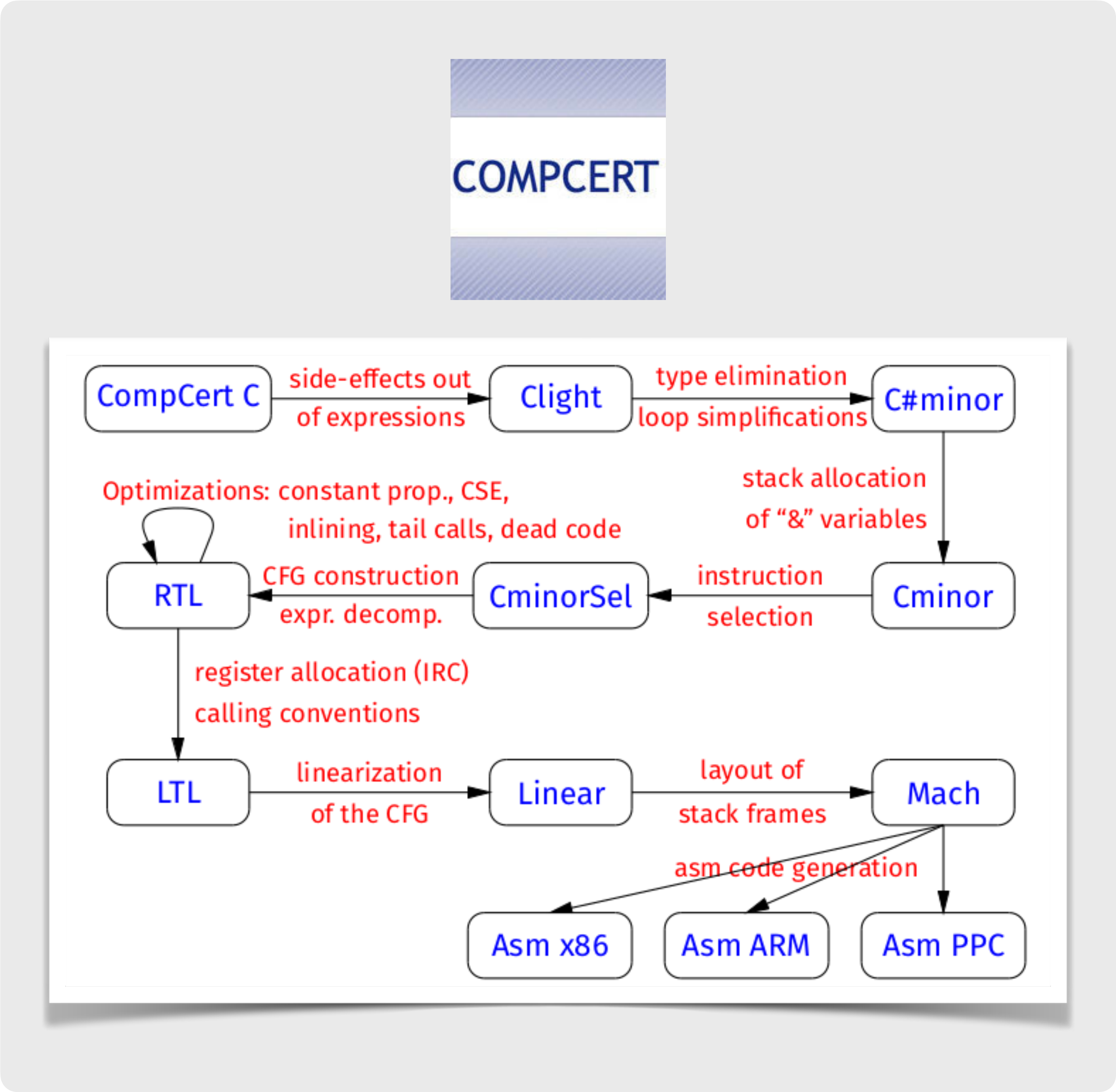
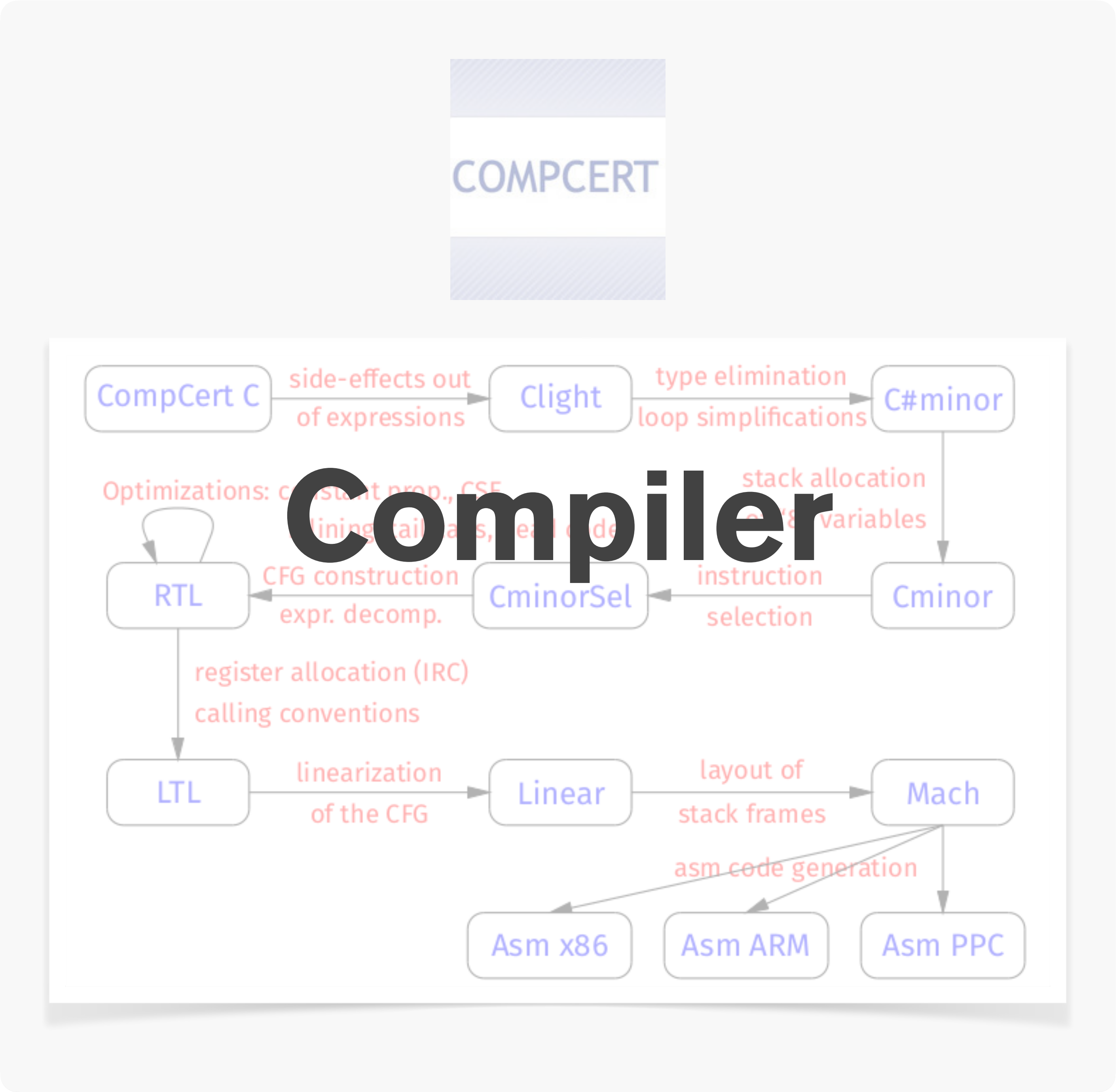
CS 4110
Victory Lap

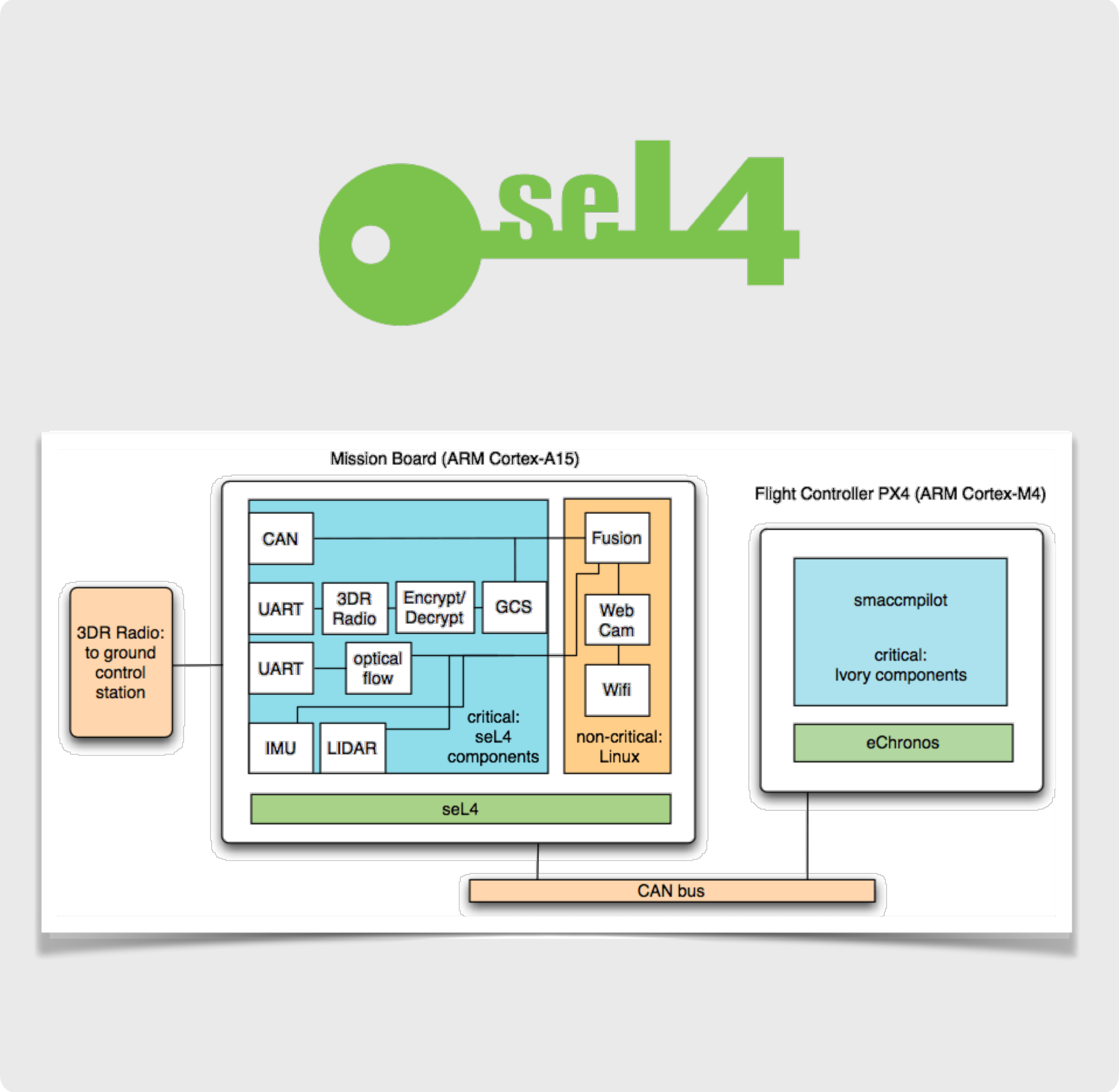
Nate Foster
Cornell

A “golden age” of formal methods

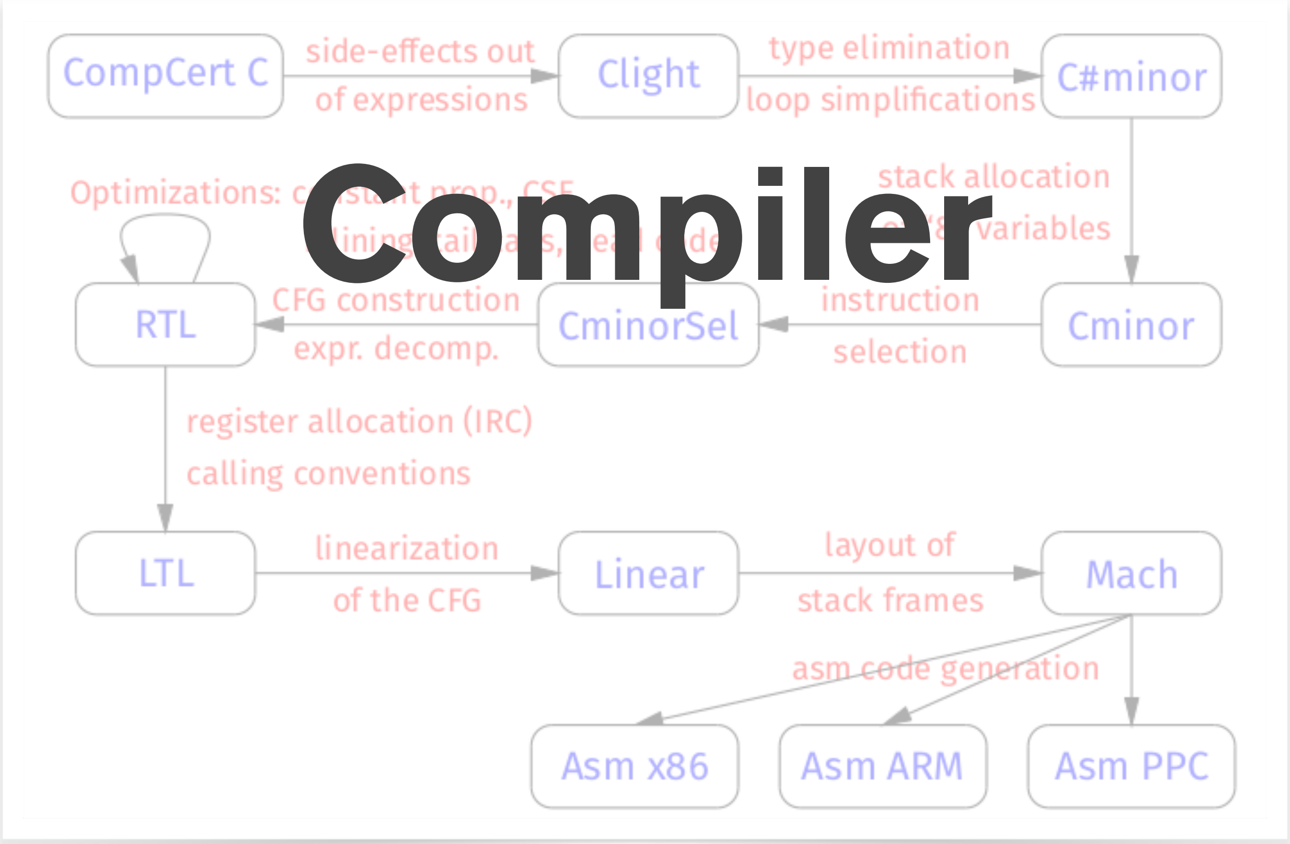


A “golden age” of formal methods

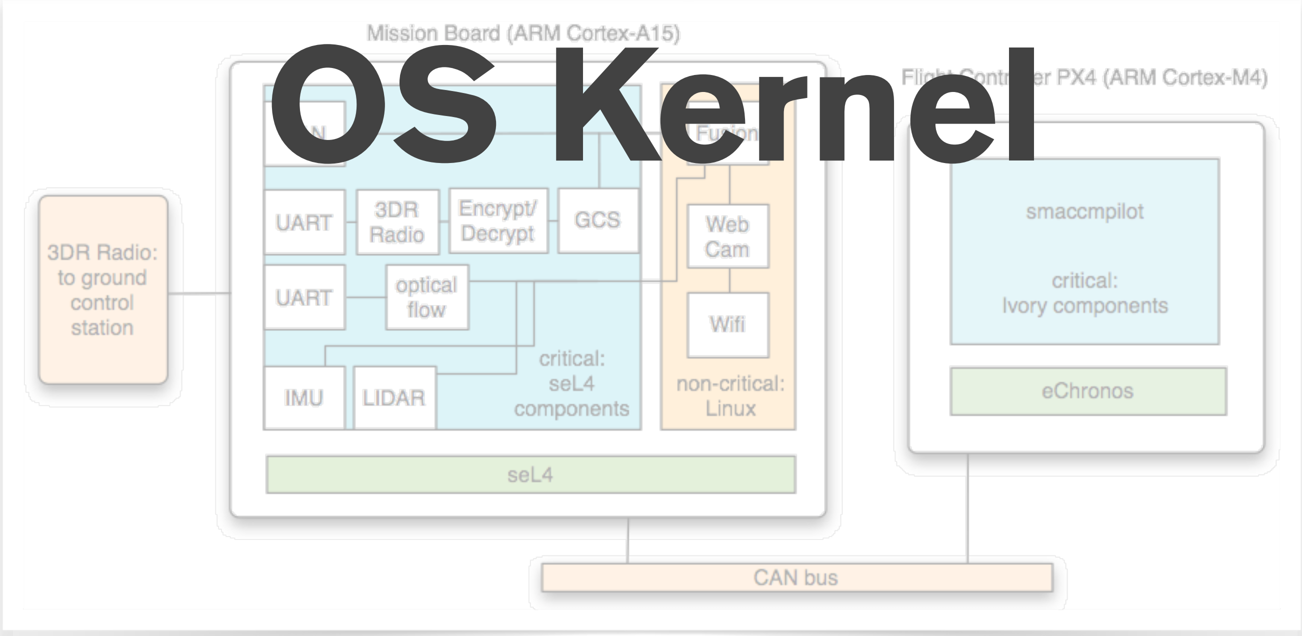




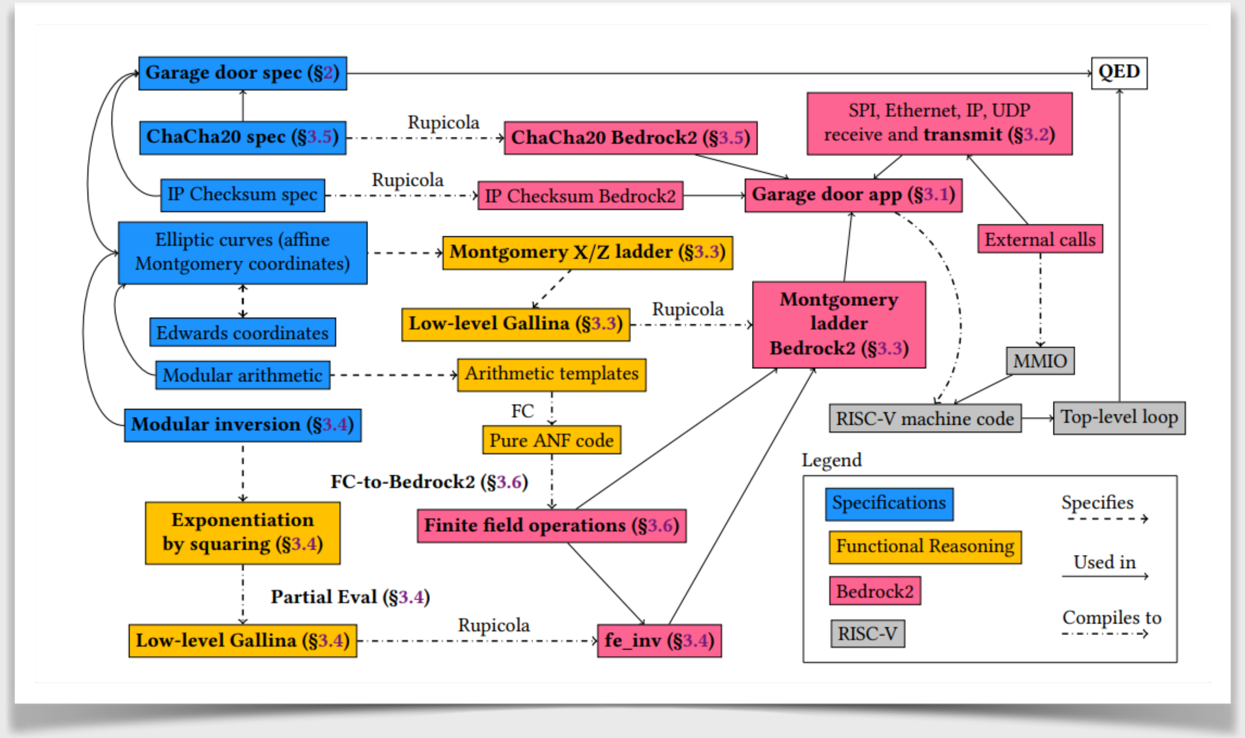
A “golden age” of formal methods



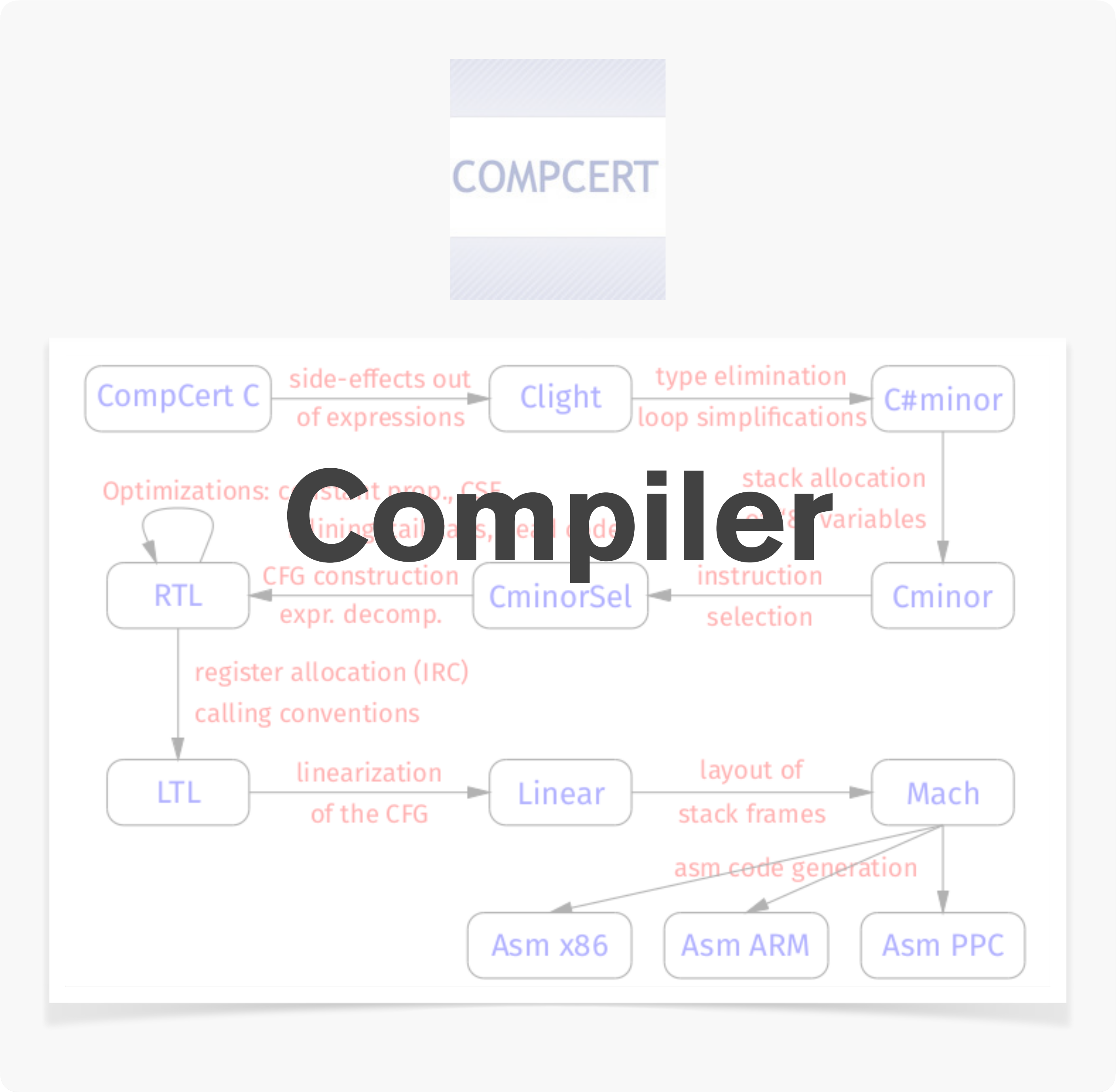
Compiler

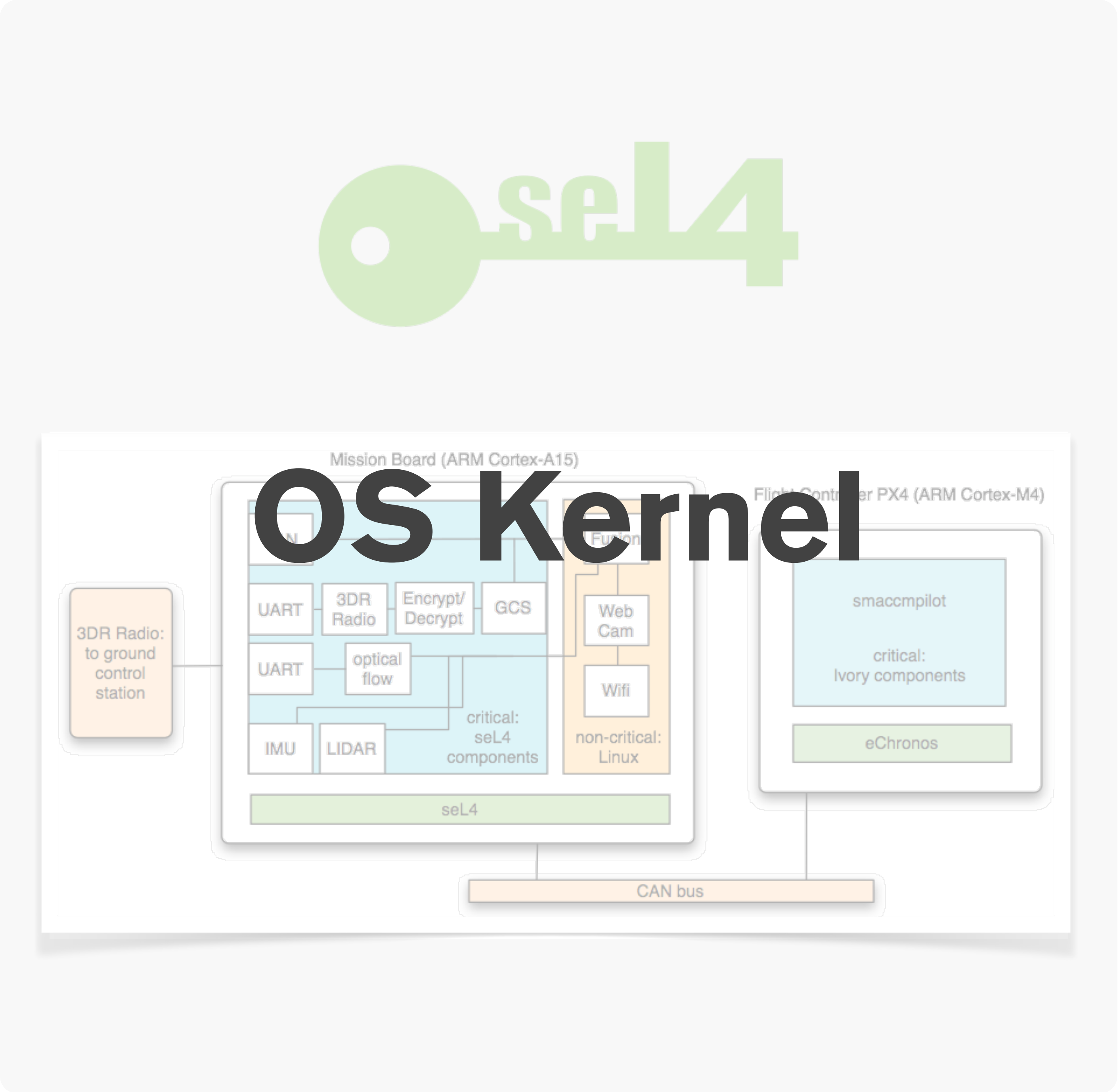


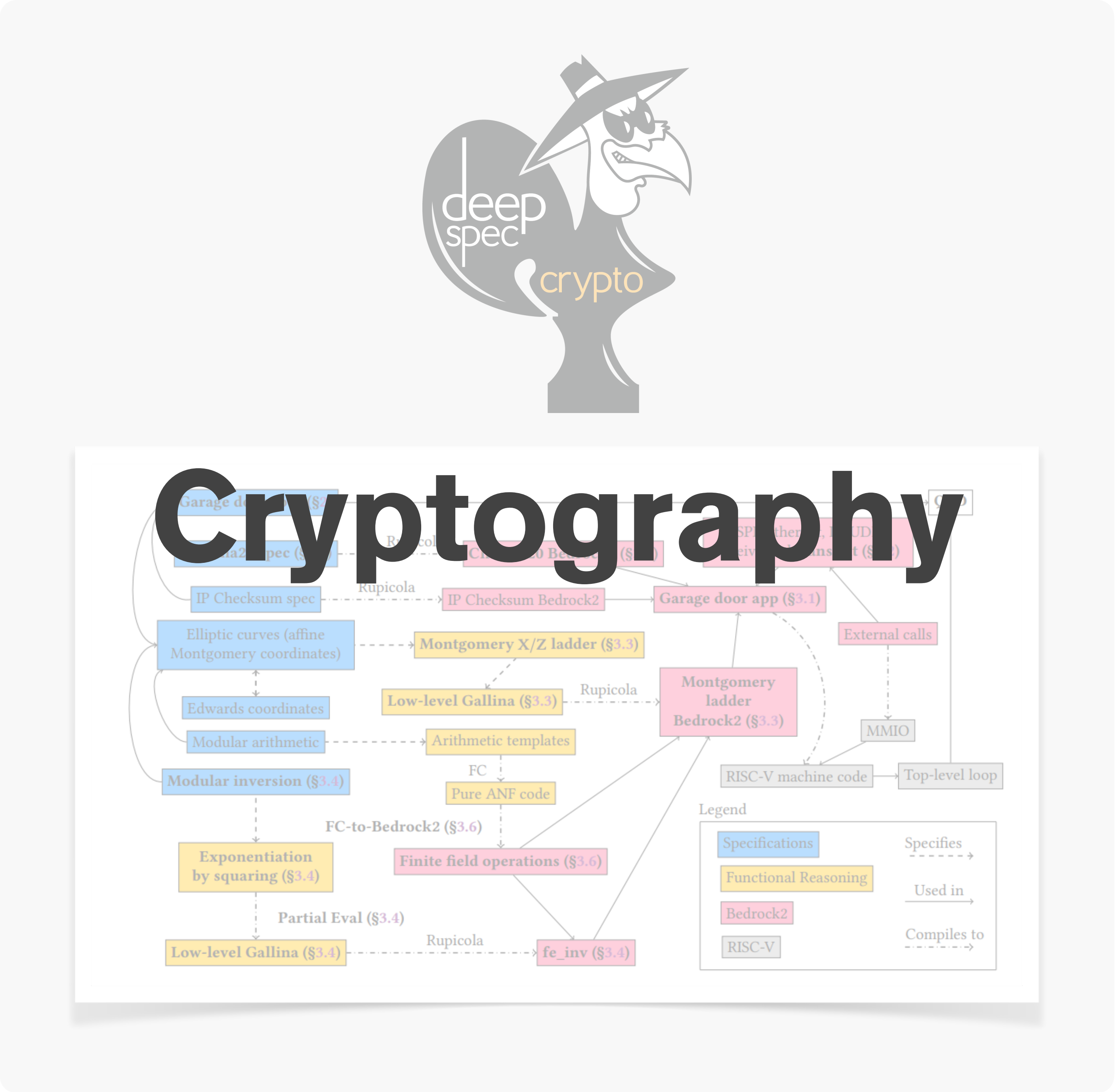
OS Kernel



A “golden age” of formal methods







What about networks?

THE DESIGN PHILOSOPHY OF THE DARPA INTERNET PROTOCOLS

David D. Clark

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Ma. 02139

Abstract

The Internet protocol suite, TCP/IP, was first proposed fifteen years ago. It was developed by the Defense Advanced Research Projects Agency (DARPA), and has been used widely in military and commercial systems. While there have been papers and specifications that describe how the protocols work, it is sometimes difficult to deduce from these why the protocol is as it is. For example, the Internet protocol is based on a connectionless or datagram mode of service. The motivation for this has been greatly misunderstood. This paper attempts to capture some of the early reasoning which shaped the Internet protocols.

1. Introduction

For the last 15 years[1], the Advanced Research Projects Agency of the U.S. Department of Defense has been developing a suite of protocols for packet switched networking. These protocols, which include the Internet Protocol (IP), and the Transmission Control Protocol (TCP), are now U.S. Department of Defense standards for internetworking, and are in wide use in the commercial networking environment. The ideas developed in this effort have also influenced other protocol suites, most importantly the connectionless configuration of the ISO protocols[2, 3, 4].

While specific information on the DOD protocols is fairly generally available[5, 6, 7], it is sometimes difficult to determine the motivation and reasoning which led to the design.

In fact, the design philosophy has evolved considerably from the first proposal to the current standards. For example, the idea of the datagram, or connectionless service, does not receive particular emphasis in the first paper, but has come to be the defining characteristic of the protocol. Another example is the layering of the architecture into the IP and TCP layers. This seems basic to the design, but was also not a part of the original proposal. These changes in the Internet design arose through the repeated pattern of implementation and testing that occurred before the standards were set.

The Internet architecture is still evolving. Sometimes a new extension challenges one of the design principles, but in any case an understanding of the history of the design provides a necessary context for current design extensions. The connectionless configuration of ISO protocols has also been colored by the history of the Internet suite, so an understanding of the Internet design philosophy may be helpful to those working with ISO.

This paper catalogs one view of the original objectives of the Internet architecture, and discusses the relation between these goals and the important features of the protocols.

2. Fundamental Goal

What about networks?

THE DESIGN PHILOSOPHY OF THE DARPA INTERNET PROTOCOLS

David D. Clark

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Ma. 02139

Abstract

The Internet protocol suite, TCP/IP, was first proposed fifteen years ago. It was developed by the Defense Advanced Research Projects Agency (DARPA), and has been used widely in military and commercial systems. While there have been papers and specifications that describe how the protocols work, it is sometimes difficult to deduce from these why the protocol is as it is. For example, the Internet protocol is based on a connectionless or datagram mode of service. The motivation for this has been greatly misunderstood. This paper attempts to capture some of the early reasoning which shaped the Internet protocols.

1. Introduction

For the last 15 years[1], the Advanced Research Projects Agency of the U.S. Department of Defense has been developing a suite of protocols for packet switched networking. These protocols, which include the Internet Protocol (IP), and the Transmission Control Protocol (TCP), are now U.S. Department of Defense standards for internetworking, and are in wide use in the commercial networking environment. The ideas developed in this effort have also influenced other protocol suites, most importantly the connectionless configuration of the ISO protocols[2, 3, 4].

While specific information on the DOD protocols is fairly generally available[5, 6, 7], it is sometimes difficult to determine the motivation and reasoning which led to the design.

In fact, the design philosophy has evolved considerably from the first proposal to the current standards. For example, the idea of the datagram, or connectionless service, does not receive particular emphasis in the first paper, but has come to be the defining characteristic of the protocol. Another example is the layering of the architecture into the IP and TCP layers. This seems basic to the design, but was also not a part of the original proposal. These changes in the Internet design arose through the repeated pattern of implementation and testing that occurred before the standards were set.

The Internet architecture is still evolving. Sometimes a new extension challenges one of the design principles, but in any case an understanding of the history of the design provides a necessary context for current design extensions. The connectionless configuration of ISO protocols has also been colored by the history of the Internet suite, so an understanding of the Internet design philosophy may be helpful to those working with ISO.

This paper catalogs one view of the original objectives of the Internet architecture, and discusses the relation between these goals and the important features of the protocols.

2. Fundamental Goal

"While **tools to verify logical correctness are useful**, both at the specification and implementation stage, **they do not help with the severe problems that often arise related to performance.**"

Evolution of networks

Conventional Networks

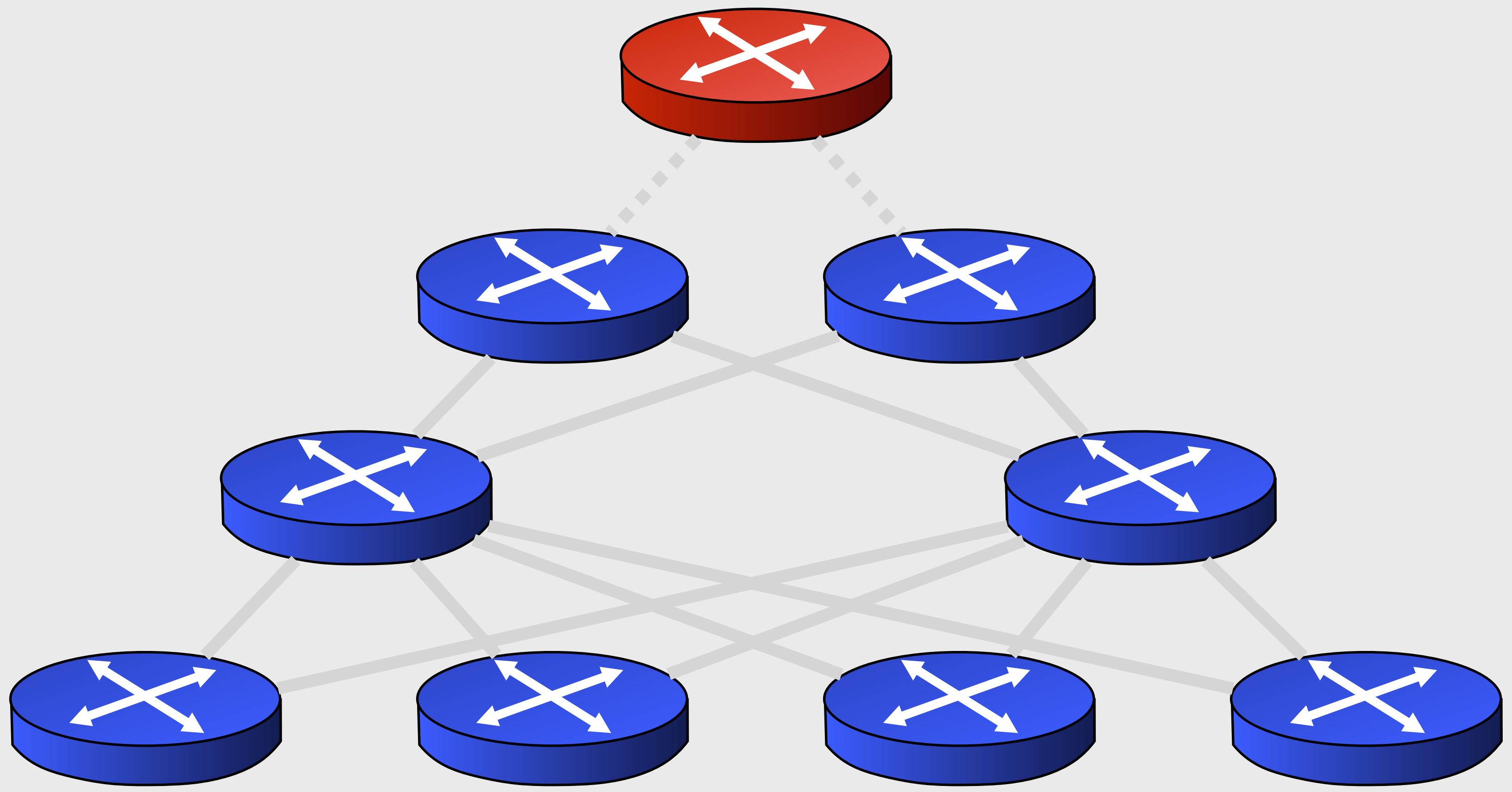
- Vertically integrated
- Fixed protocols
- Vendors write the software

Evolution of networks

Conventional Networks

- Vertically integrated
- Fixed protocols
- Vendors write the software

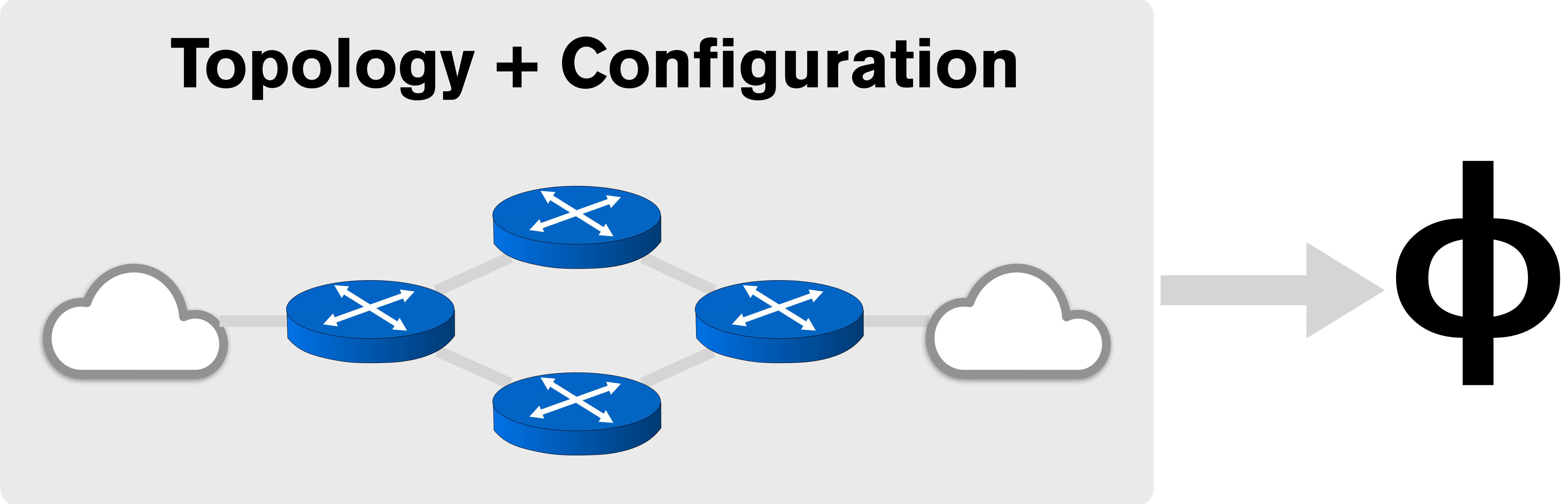
Modern Networks



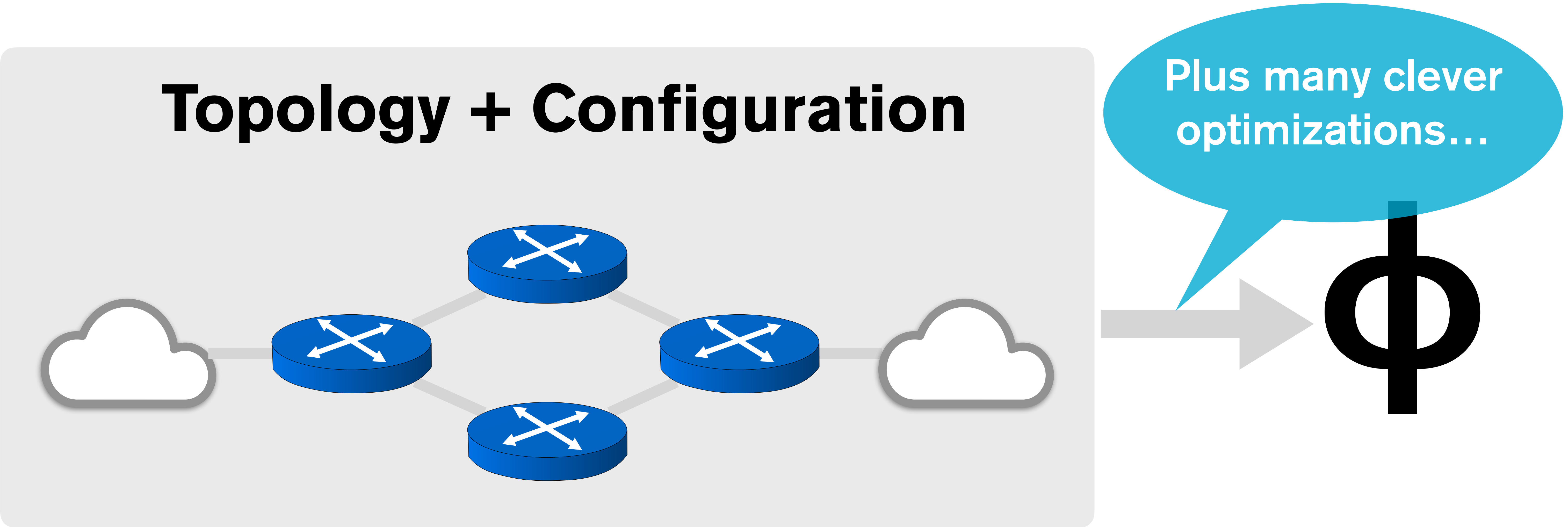
- Disaggregated
- Programmable
- Owners write the software

Status quo: solver-based approach

Status quo: solver-based approach

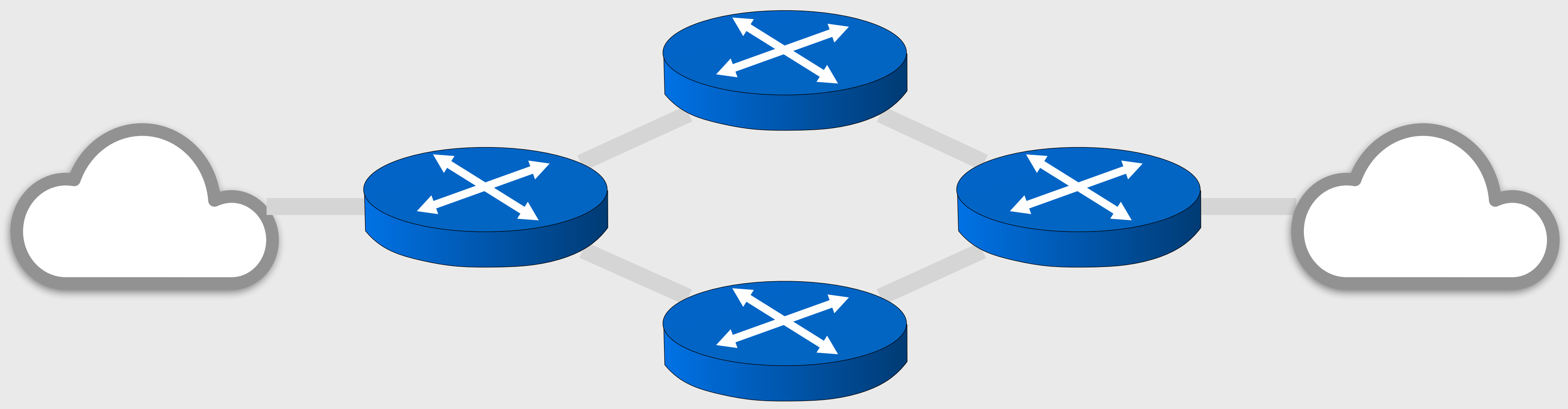


Status quo: solver-based approach



Status quo: solver-based approach

Topology + Configuration



Plus many clever optimizations…

→ ϕ

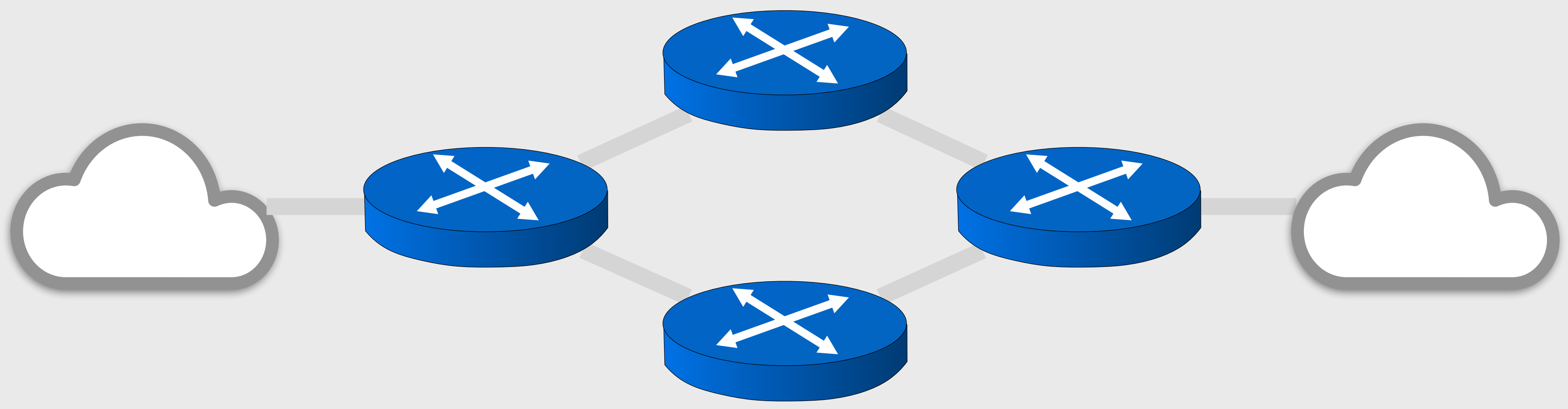
Specification

"The network should be free of forwarding loops, and every packet should be delivered to its destination"

→ ψ

Status quo: solver-based approach

Topology + Configuration



Plus many clever optimizations…

Φ

Specification

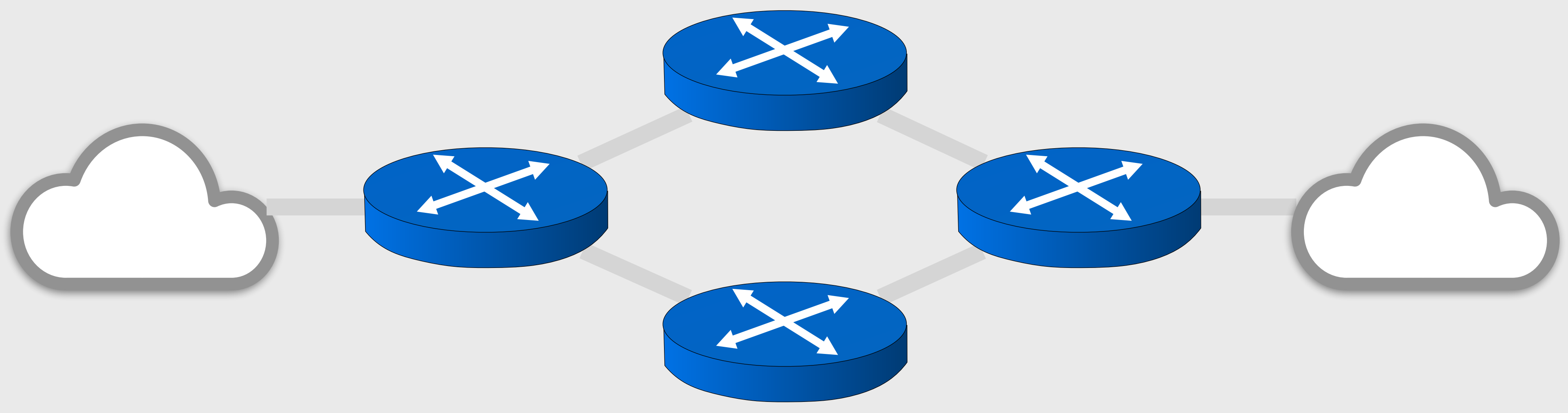
"The network should be free of forwarding loops, and every packet should be delivered to its destination"

Ψ

SMT Solver

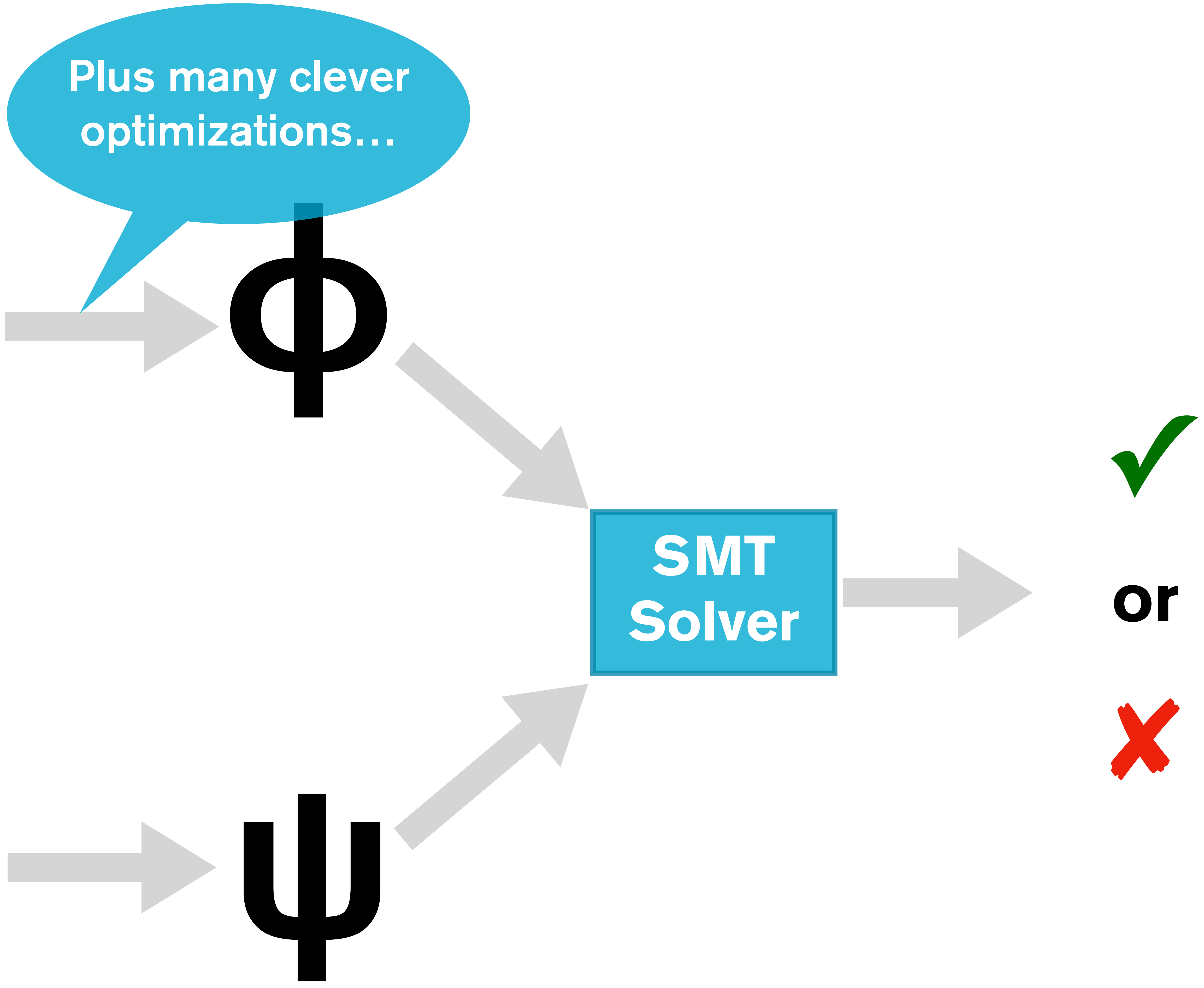
Status quo: solver-based approach

Topology + Configuration

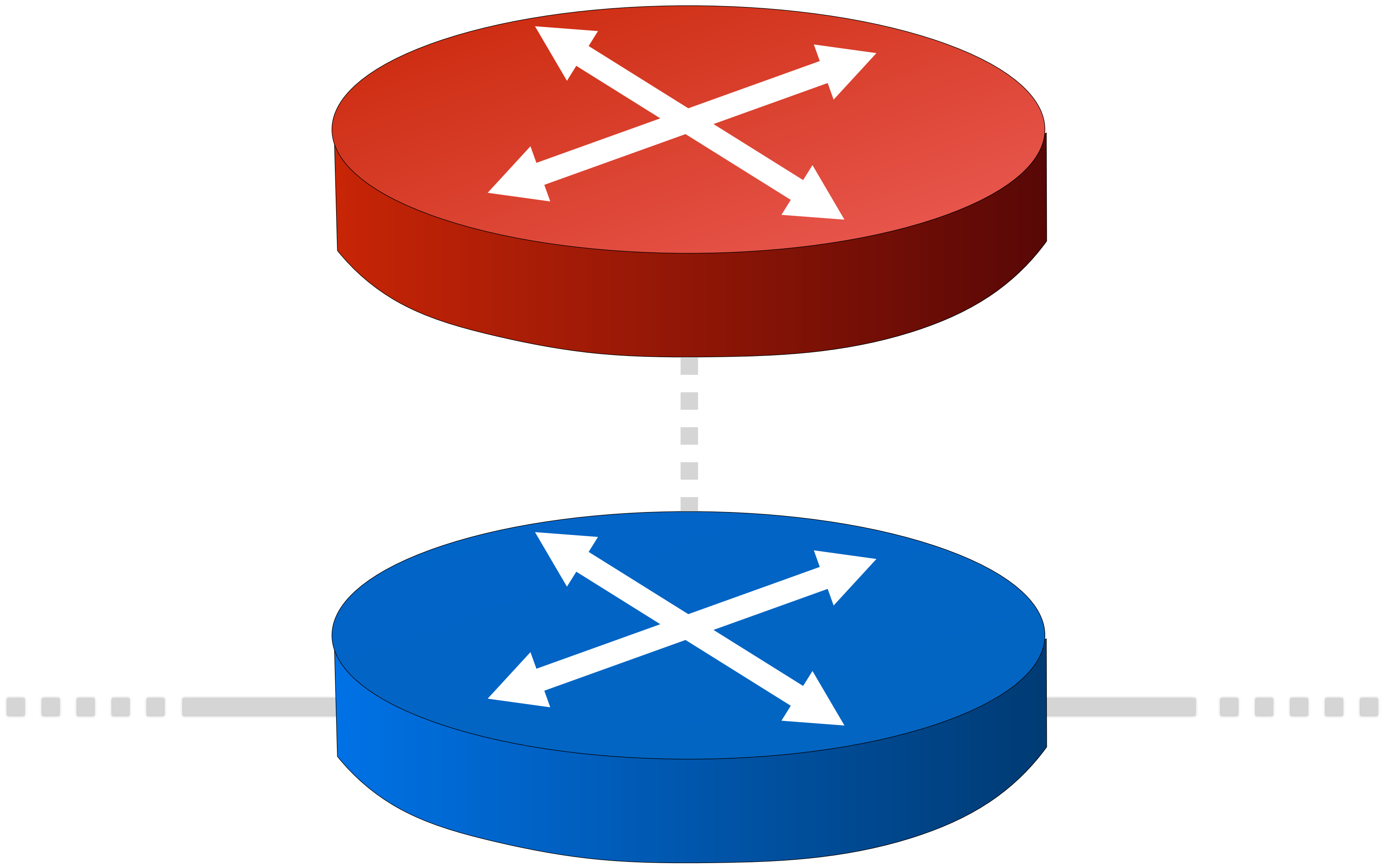


Specification

“The network should be free of forwarding loops, and every packet should be delivered to its destination”

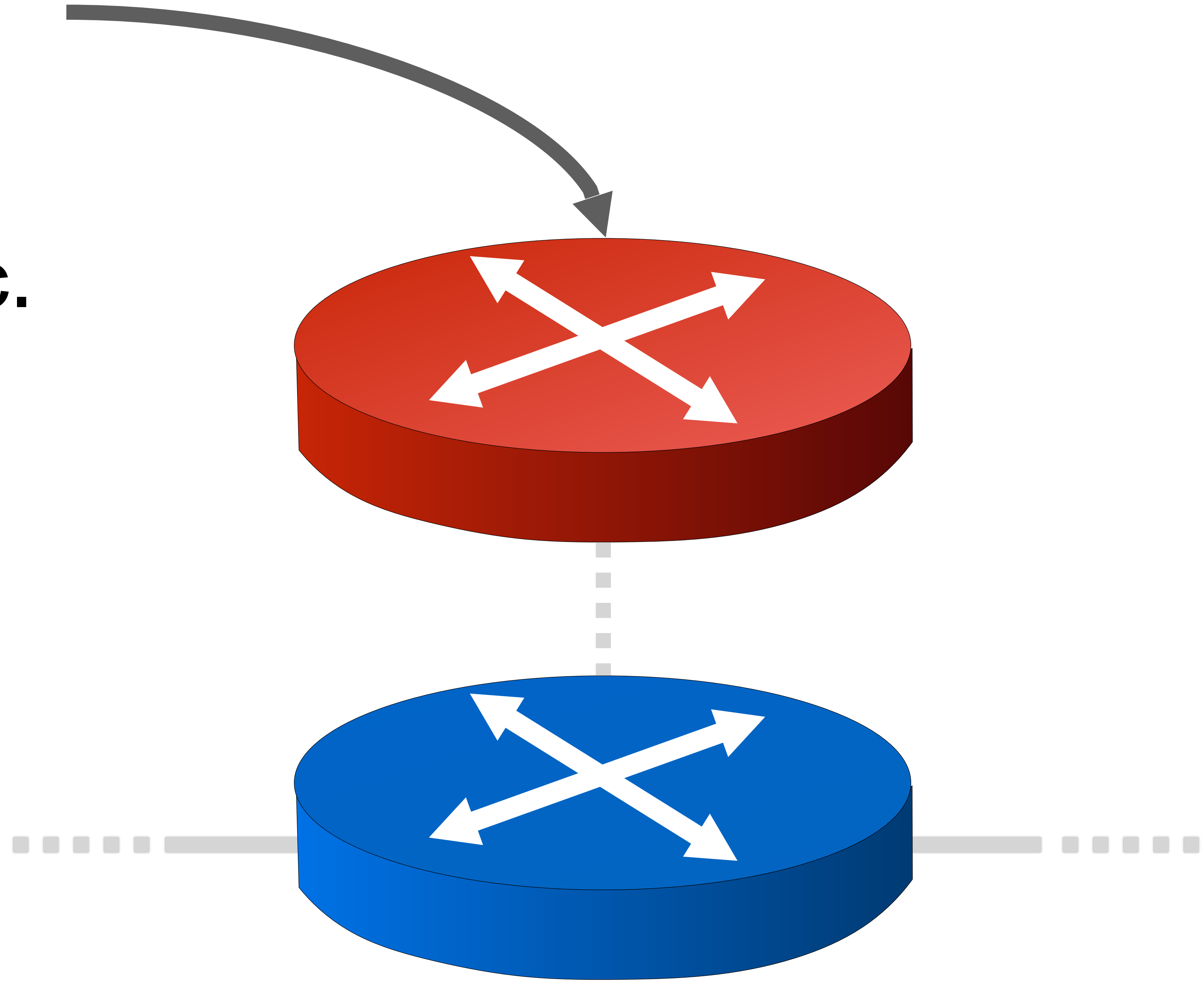


Networking terminology



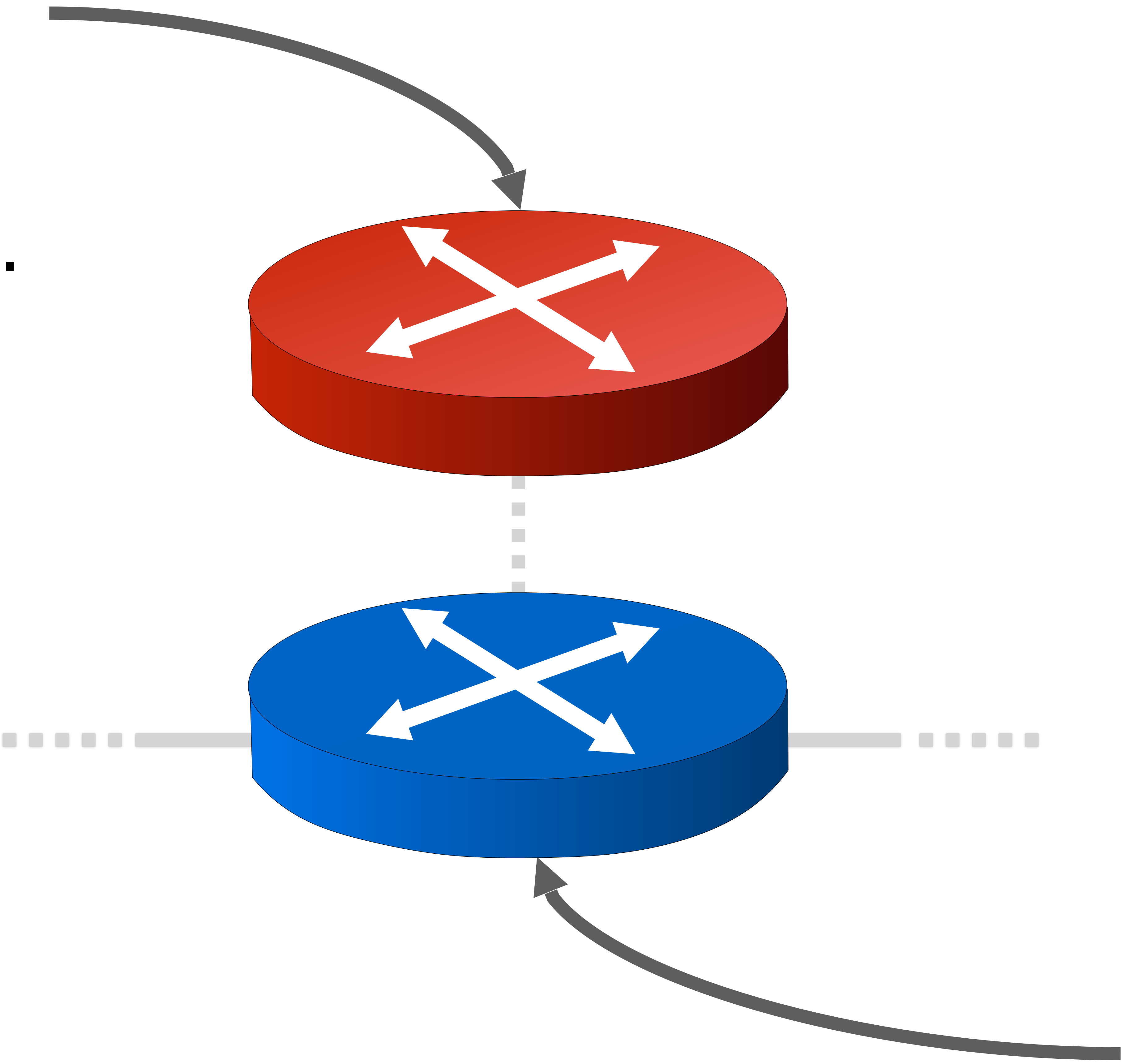
Networking terminology

Control Plane
discovers topology,
computes routes,
manages policy, etc.



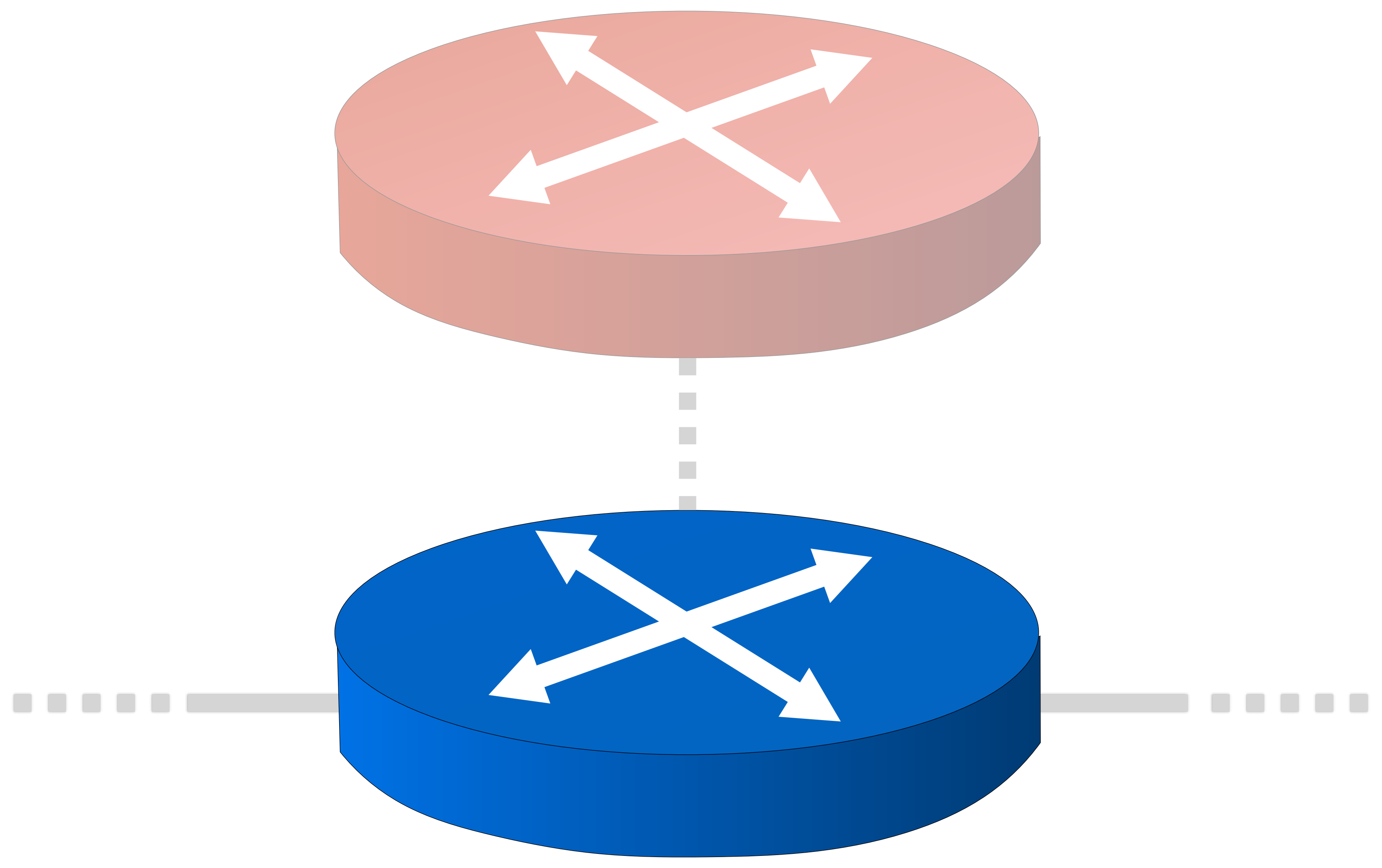
Networking terminology

Control Plane
discovers topology,
computes routes,
manages policy, etc.

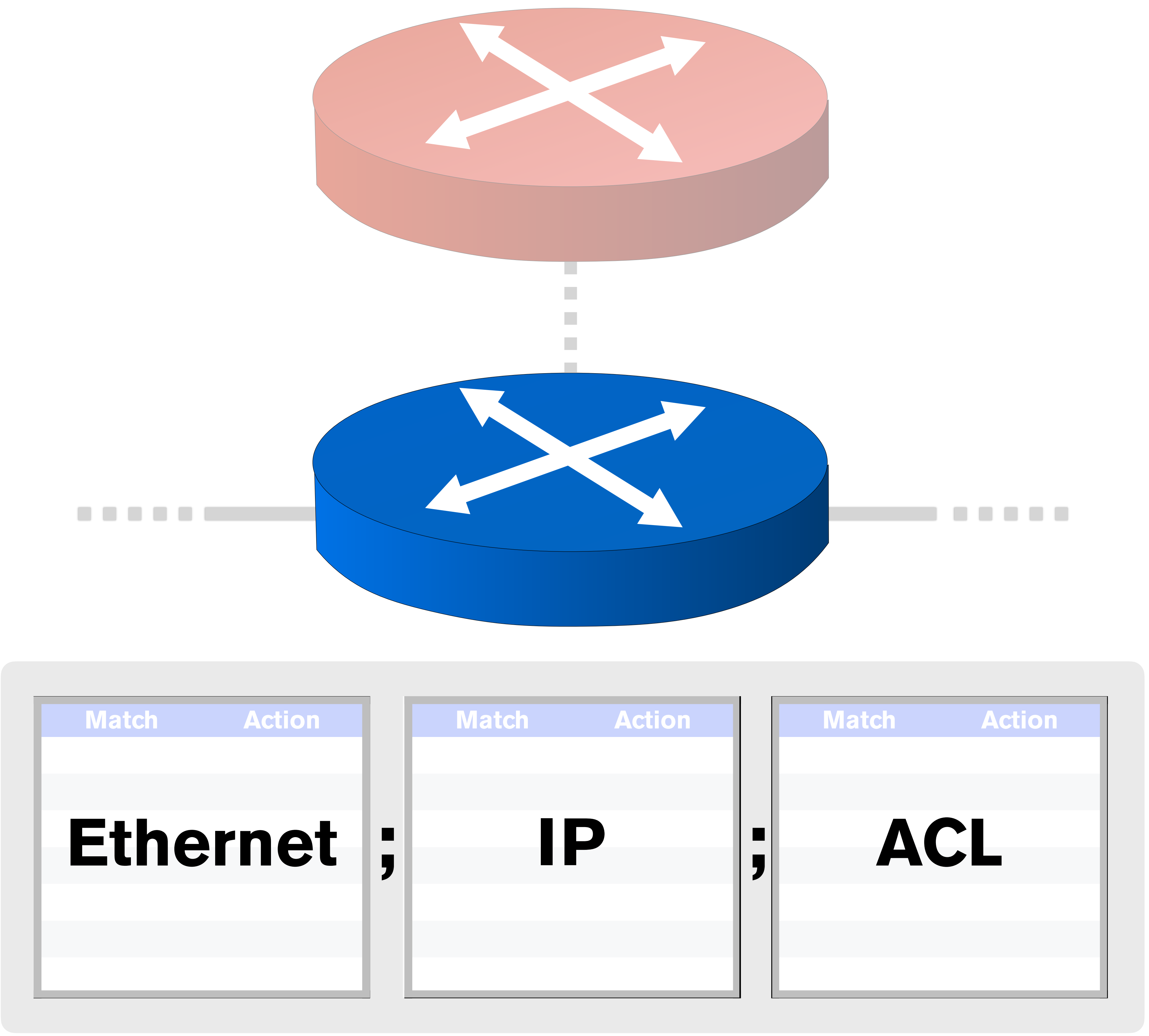


Data plane
forwards packets,
monitors traffic,
enforces access
control, etc.

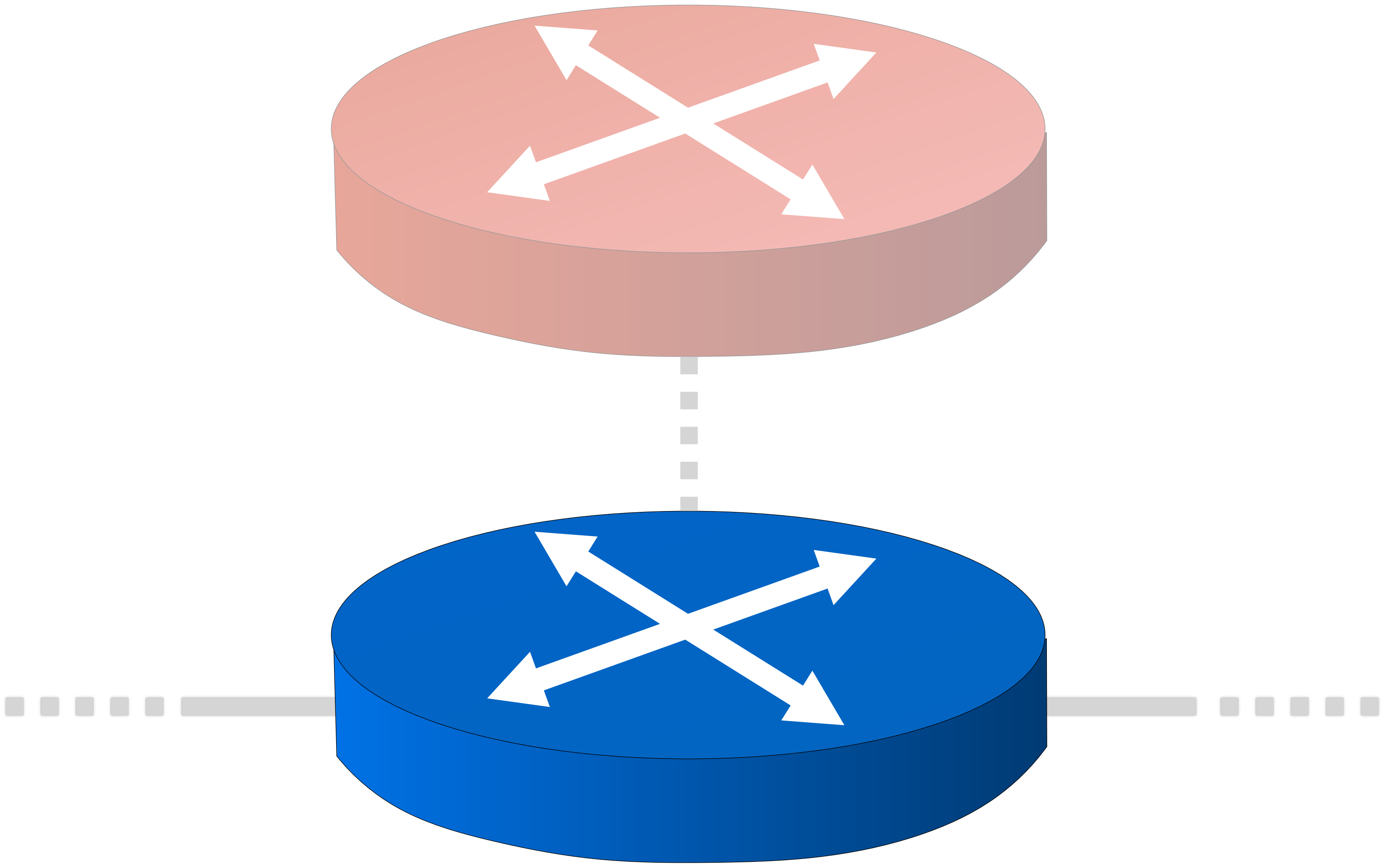
Data Plane

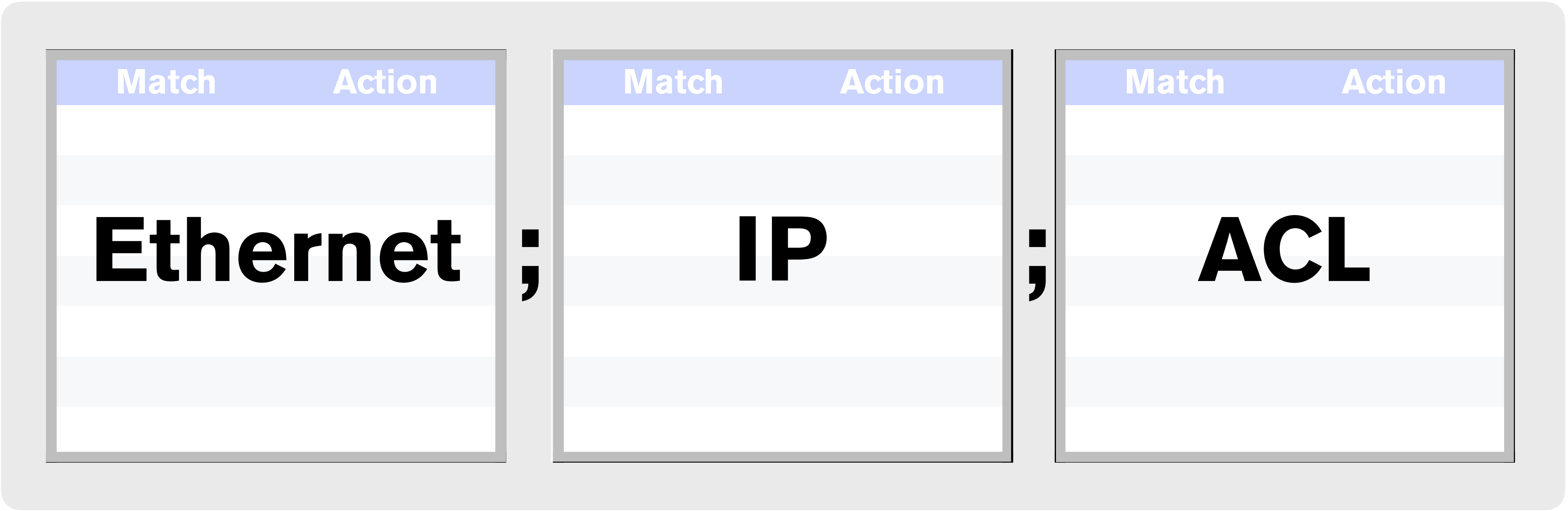


Data Plane



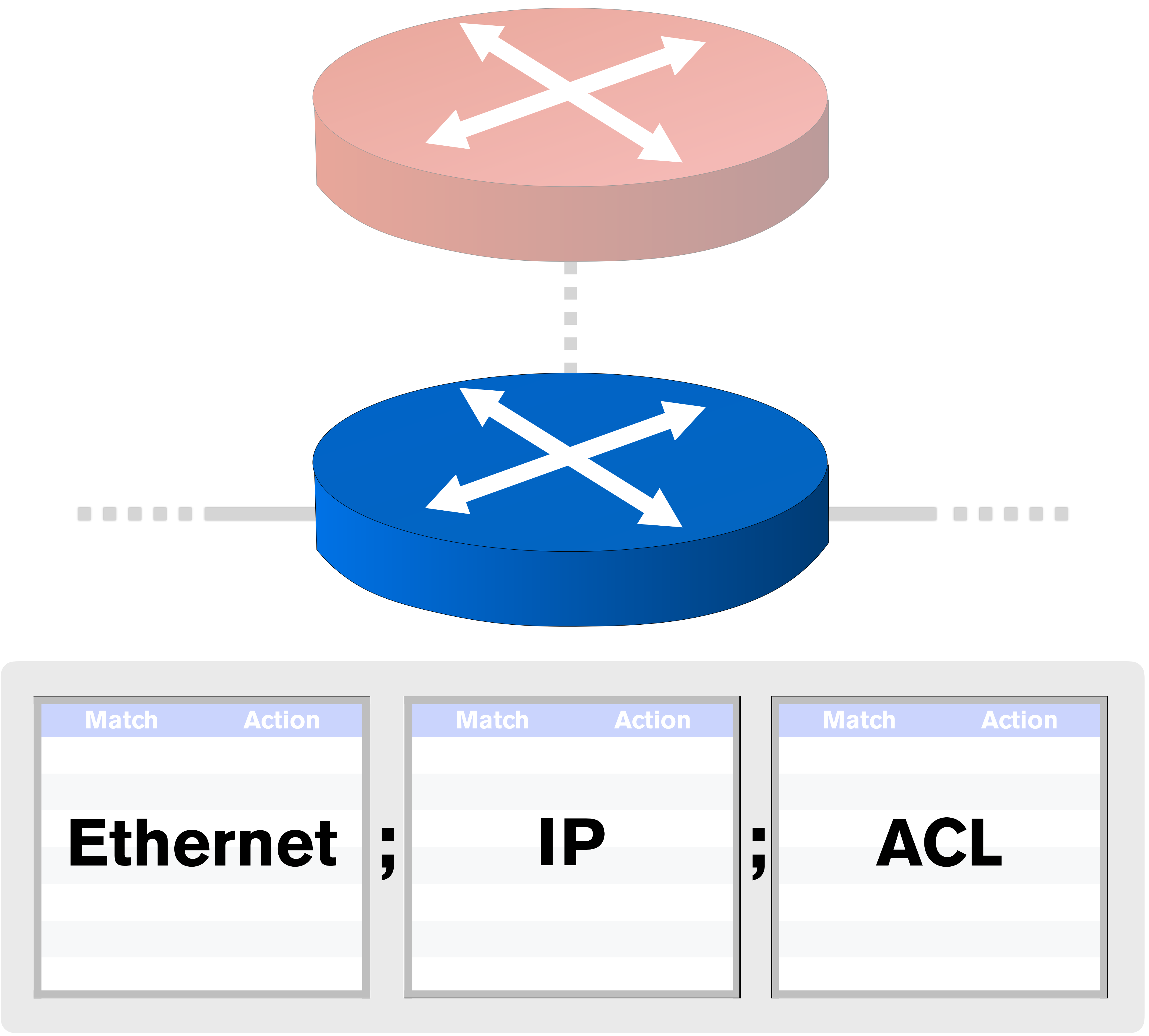
Data Plane





Typically structured as a *pipeline* of *match-action forwarding tables*, in hardware or software

Data Plane

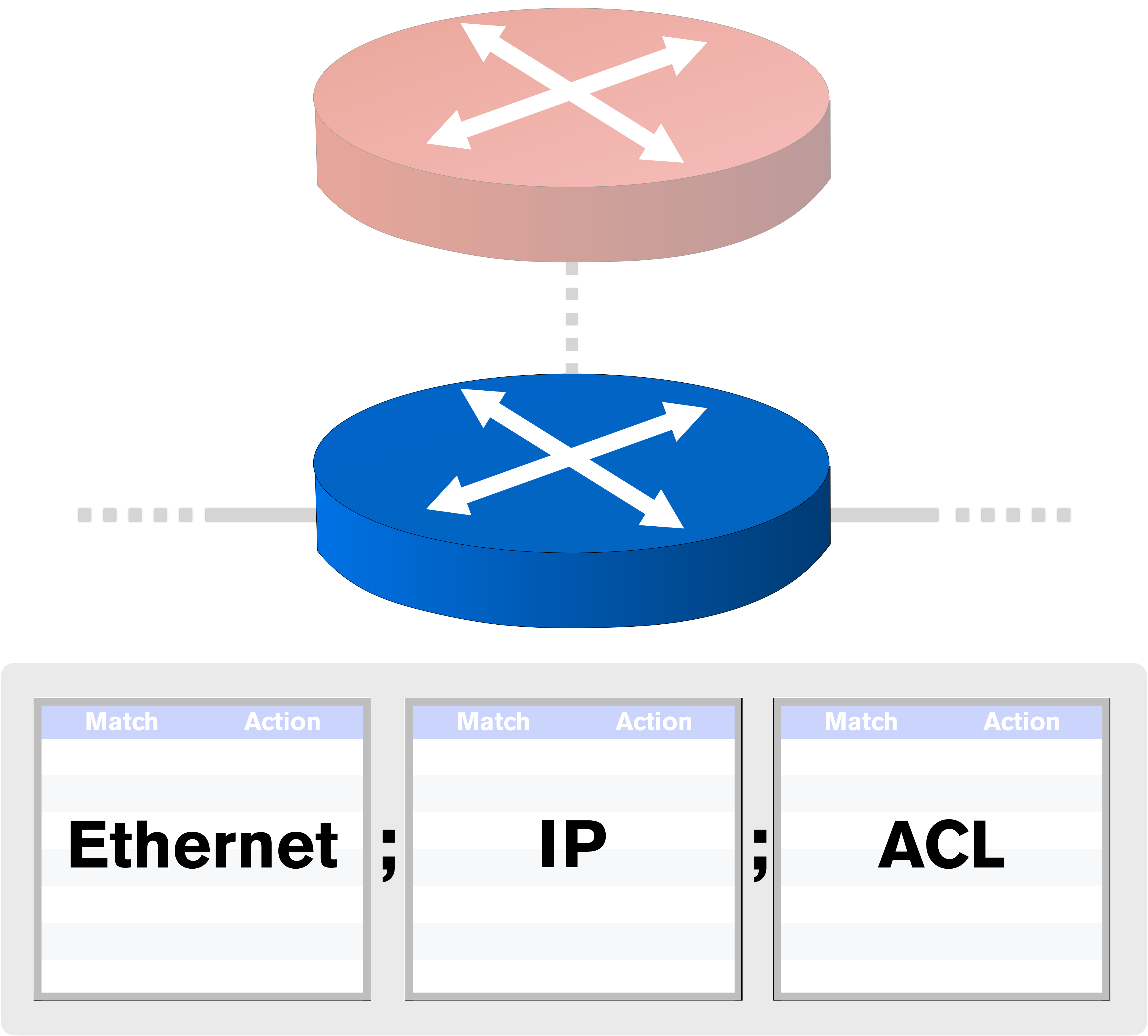


Typically structured as a *pipeline* of *match-action forwarding tables*, in hardware or software

Each table contains *rules* that:

- *Match* on packet headers
- Execute *Actions* that transform, forward, or drop packets

Data Plane



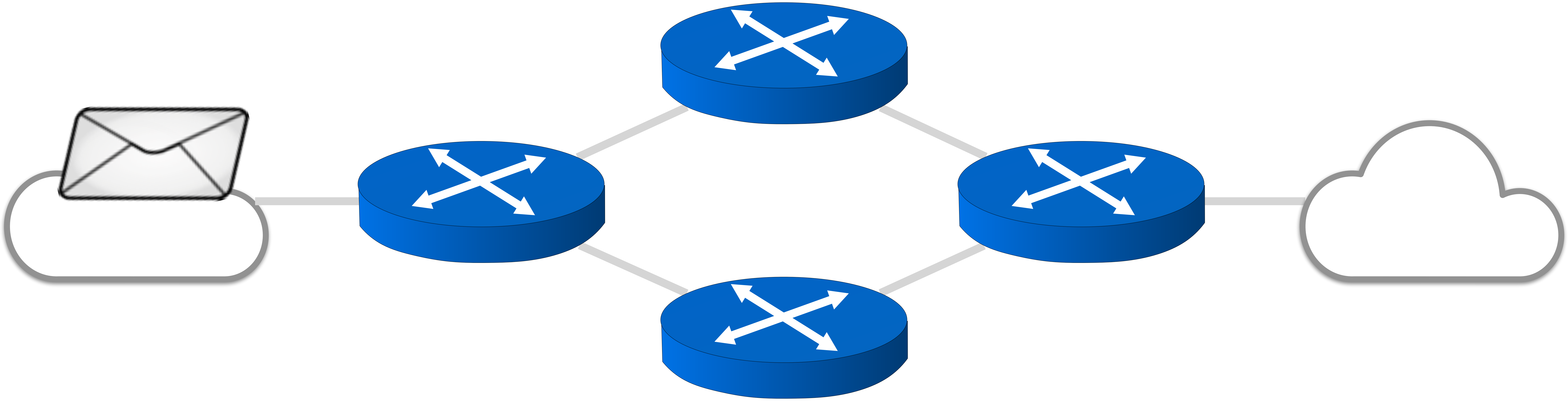
Typically structured as a *pipeline* of *match-action forwarding tables*, in hardware or software

Each table contains *rules* that:

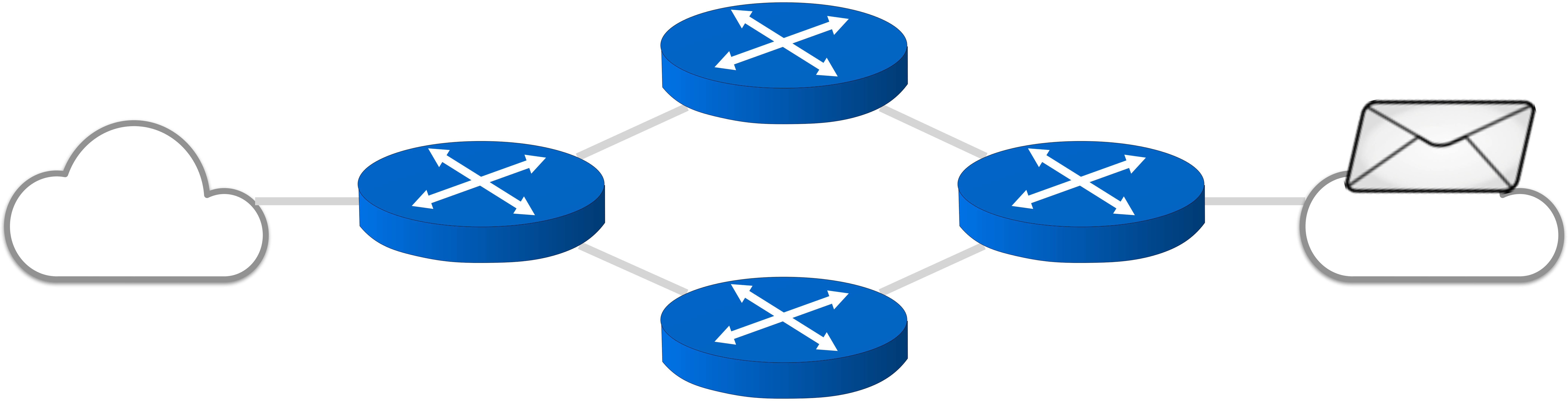
- *Match* on packet headers
- Execute *Actions* that transform, forward, or drop packets

Control plane can *dynamically reconfigure* the network by modifying the rules in tables

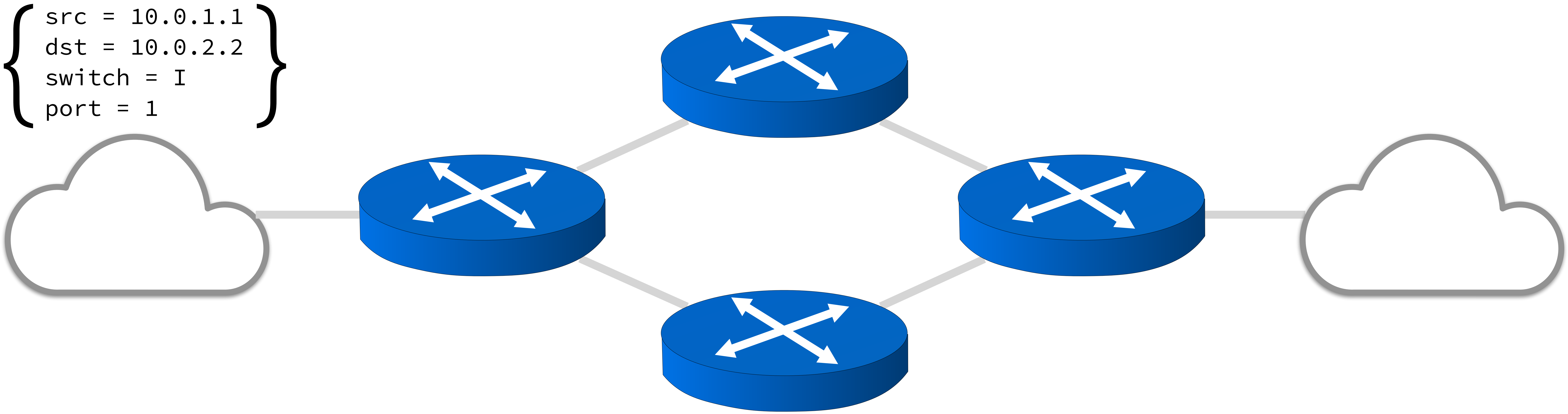
Data Plane Behavior



Data Plane Behavior

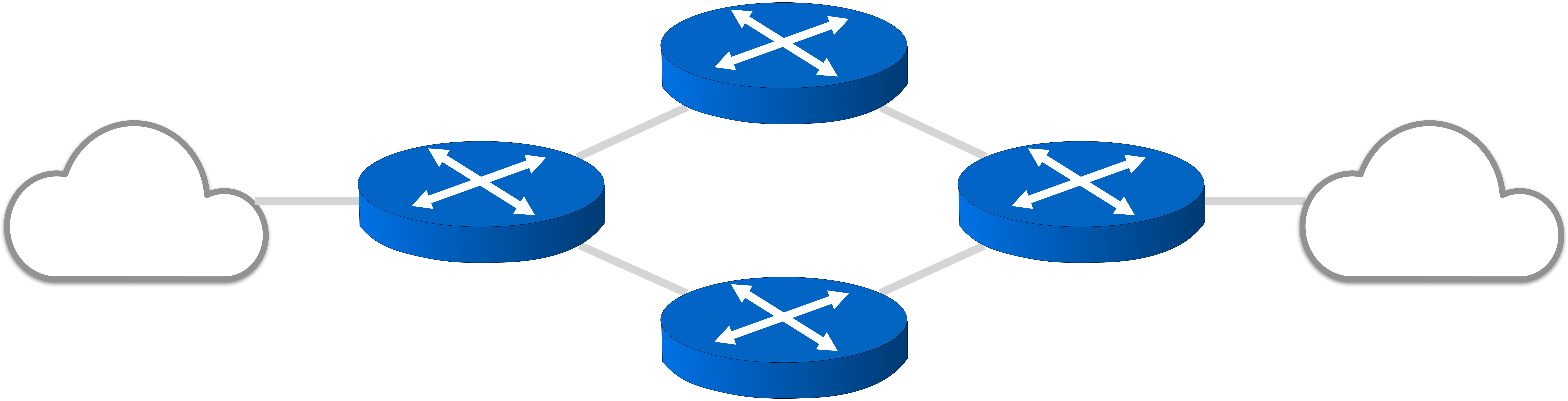


Packets: Records of fixed-width data

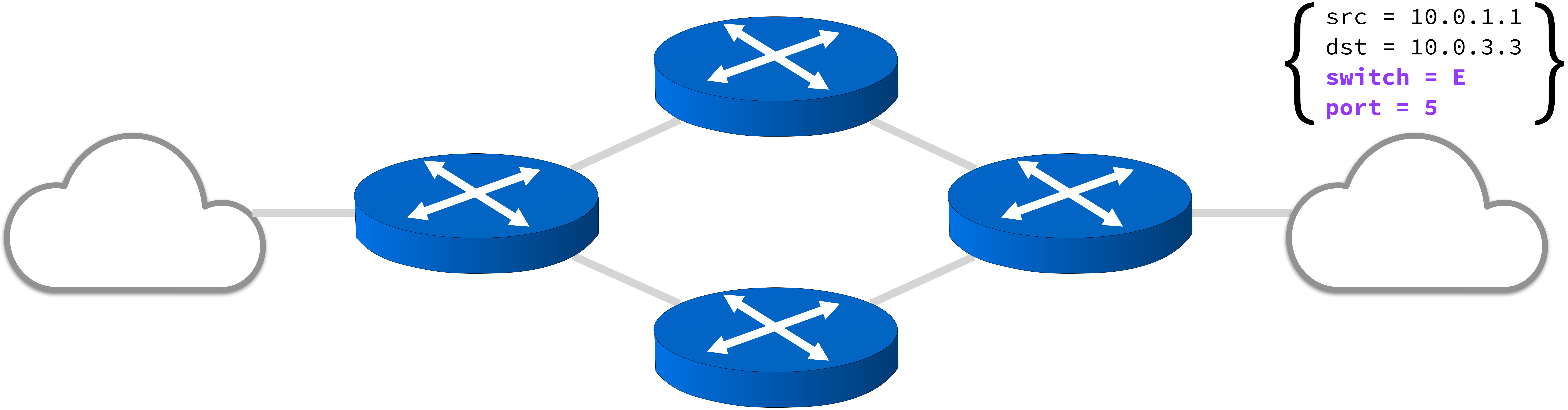


Packets: Records of fixed-width data

```
{
  src = 10.0.1.1
  dst = 10.0.2.2
  switch = A
  port = 2
}
```

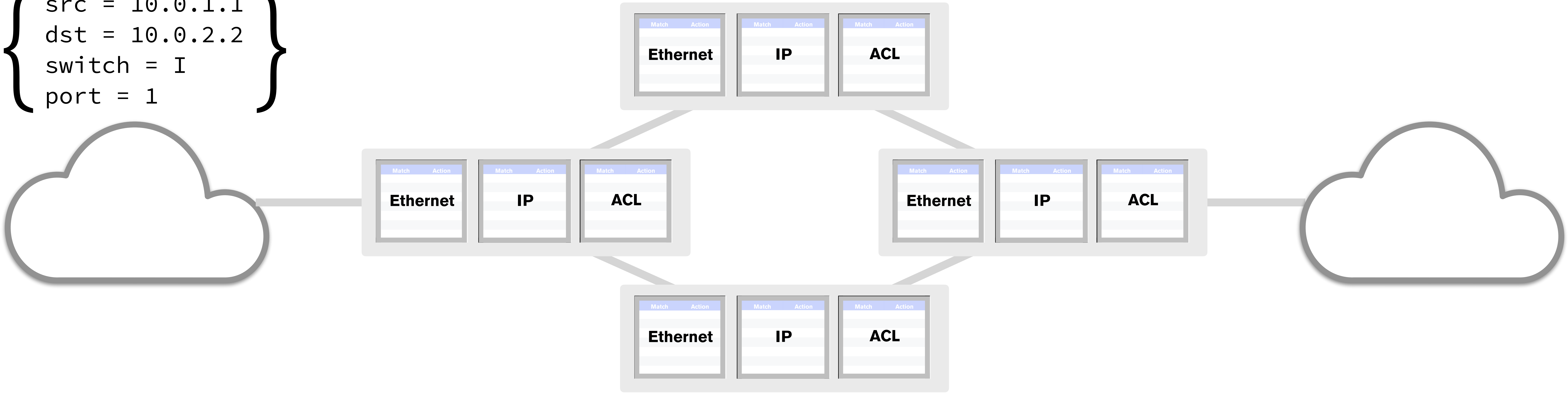


Packets: Records of fixed-width data

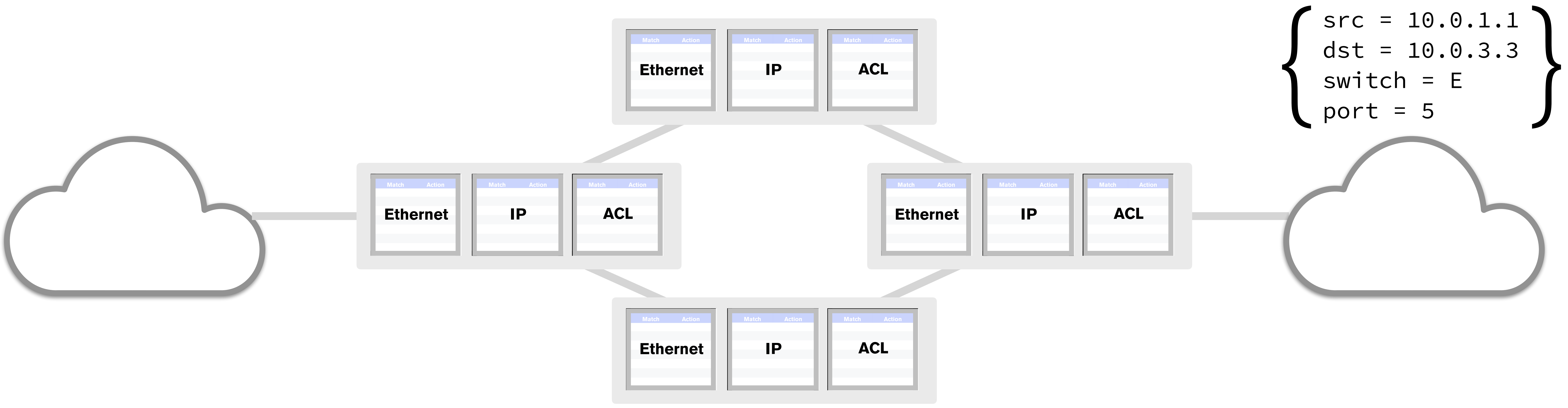


Network: Graphs of pipelines

```
src = 10.0.1.1
dst = 10.0.2.2
switch = I
port = 1
```



Network: Graphs of pipelines



Design considerations

Packet Classification
To model match-action tables, need predicates evaluated packet headers (and other variables)

Design considerations

Packet Classification

To model match-action tables, need predicates evaluated packet headers (and other variables)

Transformations

To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

Design considerations

Packet Classification
To model match-action tables, need predicates evaluated packet headers (and other variables)

Transformations
To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

Modular Composition
To model richer pipelines, need operators that compose smaller programs both conditionally and in sequence

Design considerations

Packet Classification
To model match-action tables, need predicates evaluated packet headers (and other variables)

Transformations
To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

Modular Composition
To model richer pipelines, need operators that compose smaller programs both conditionally and in sequence

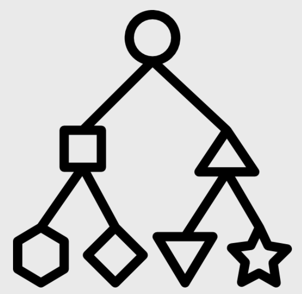
Iteration
Perhaps surprisingly, to model the potentially iterated processing performed via the topology, need general loops

NetKAT syntax

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```

NetKAT predicates

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```

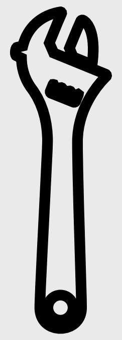


Packet Classification
To model match-action tables, need predicates evaluated packet headers (and other variables)

Match	Action
tcp_dst=22; ip_dst=10.0.1.1	allow
tcp_dst=22	deny
ip_dst=10.0.2.2	allow
ip_dst=10.0.3.3	allow
*	deny

NetKAT transformations

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```



Transformations
To model behavior of switches, need imperative updates on packets and a way to delimit end of processing at a switch

```
A ⟹ B
 ≝
dup; sw = A; sw := B; dup
```

NetKAT composition operators

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```

Modular Composition
To model richer pipelines, need operators that compose smaller programs both conditionally and in sequence

```
if p then q else r
        def
        ===
(p ; q) + (!p ; r)
```

NetKAT composition operators

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```

Note: “;” and “+” are overloaded as predicates

Modular Composition
To model richer pipelines, need operators that compose smaller programs both conditionally and in sequence

```
if p then q else r
        ≝
(p ; q) + (!p ; r)
```

NetKAT iteration

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```

Iteration
Perhaps surprisingly, to model the potentially iterated processing performed via the topology, need general loops

```
while p do q
```

$\overset{\text{def}}{=}$

```
(p ; q)* ; !p
```

NetKAT semantics

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```

Informal: NetKAT programs denote *functions* that take an input packet and produce a set of packet traces (i.e., non-empty lists).

Formal: $[\![p]\!] \in \text{Packet} \rightarrow \mathcal{P}(\text{Packet}^{+})$

NetKAT semantics

```
p, q ::=
  | id
  | drop
  | f = n
  | !p
  | f := n
  | p ; q
  | p + q
  | p*
  | dup
```

Informal: NetKAT programs denote *functions* that take an input packet and produce a set of packet traces (i.e., non-empty lists).

Formal: $[\![p]\!] \in \text{Packet} \rightarrow \mathcal{P}(\text{Packet}^{+})$

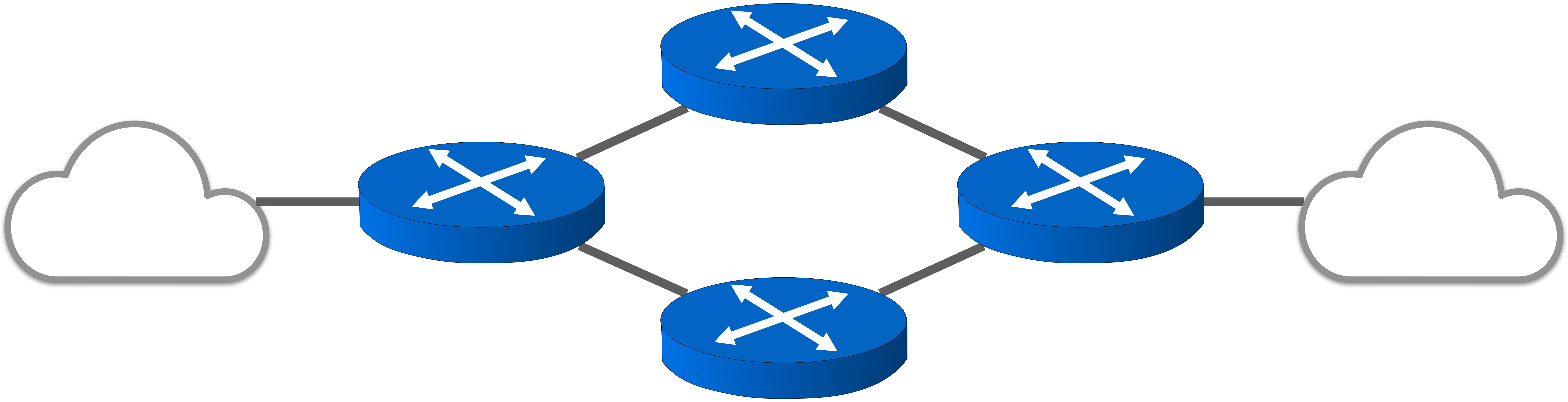
FAQ

Q: *Why a set?*
A: Enables dropping packets + multicast

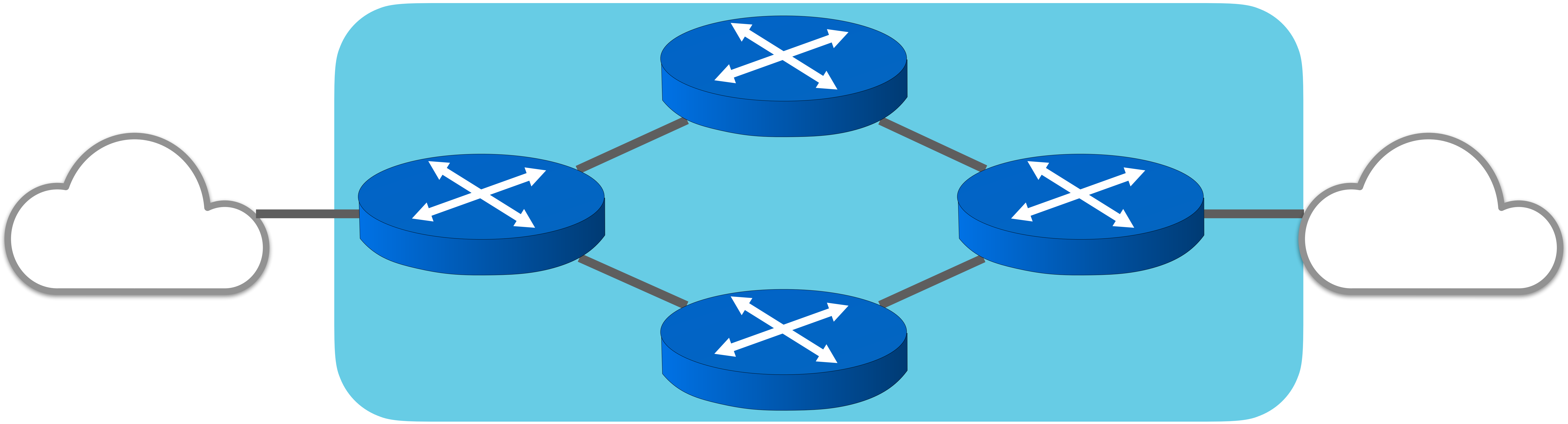
Q: *Why a trace?*
A: Captures end-to-end forwarding path

Q: *Why a function?*
A: Simple model of "mostly stateless" forwarding

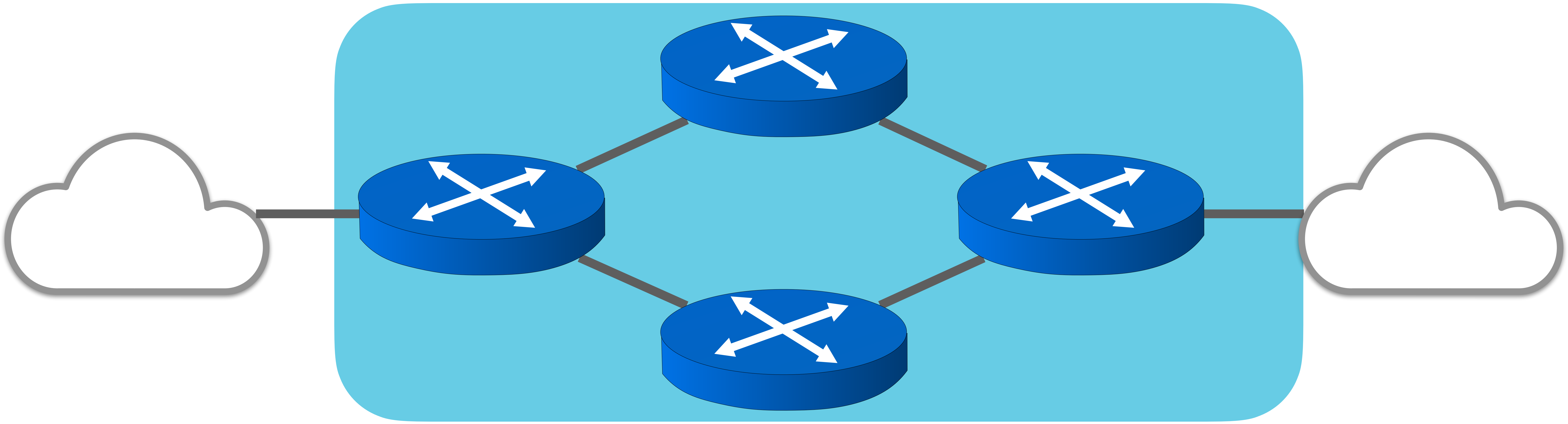
Modeling networks in NetKAT



Modeling networks in NetKAT

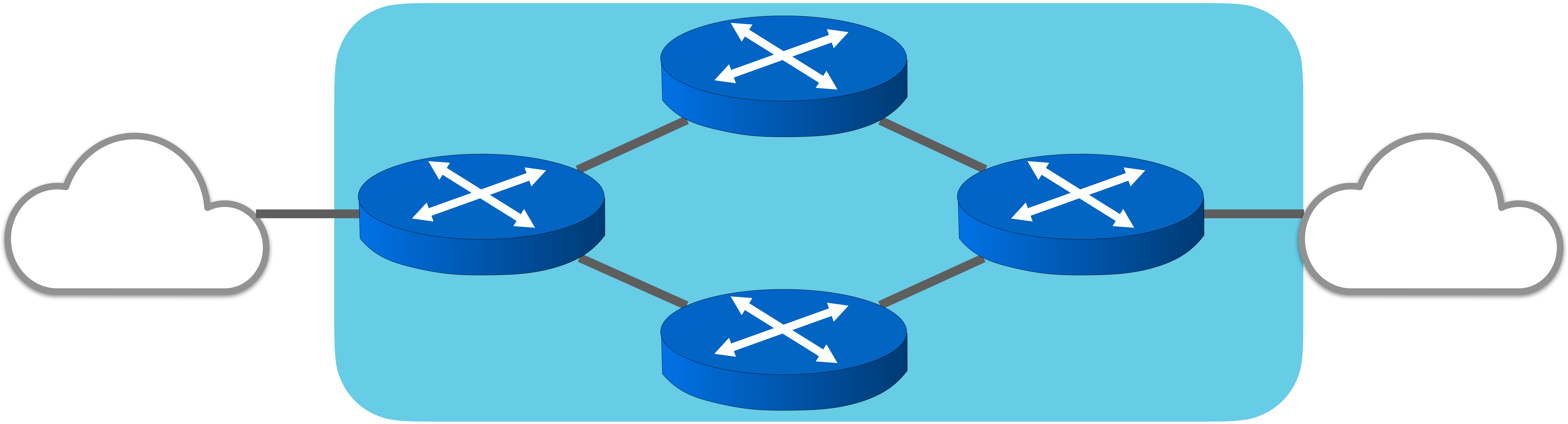


Modeling networks in NetKAT



switch
Models the processing done at each switch

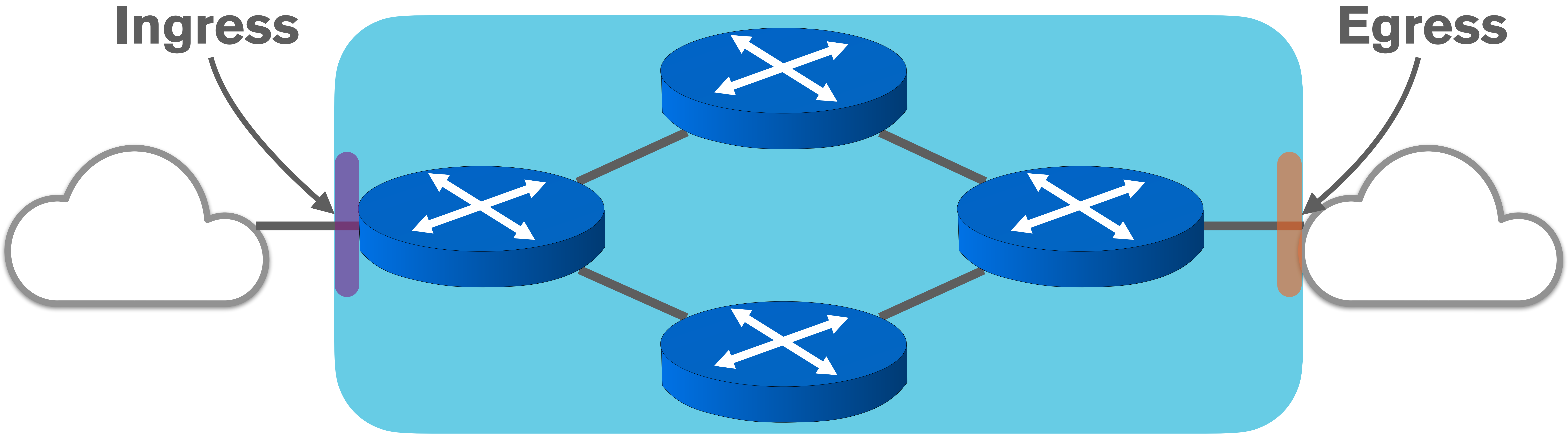
Modeling networks in NetKAT



switch
Models the processing done at each switch

topology
Models the forwarding done by each link

Modeling networks in NetKAT

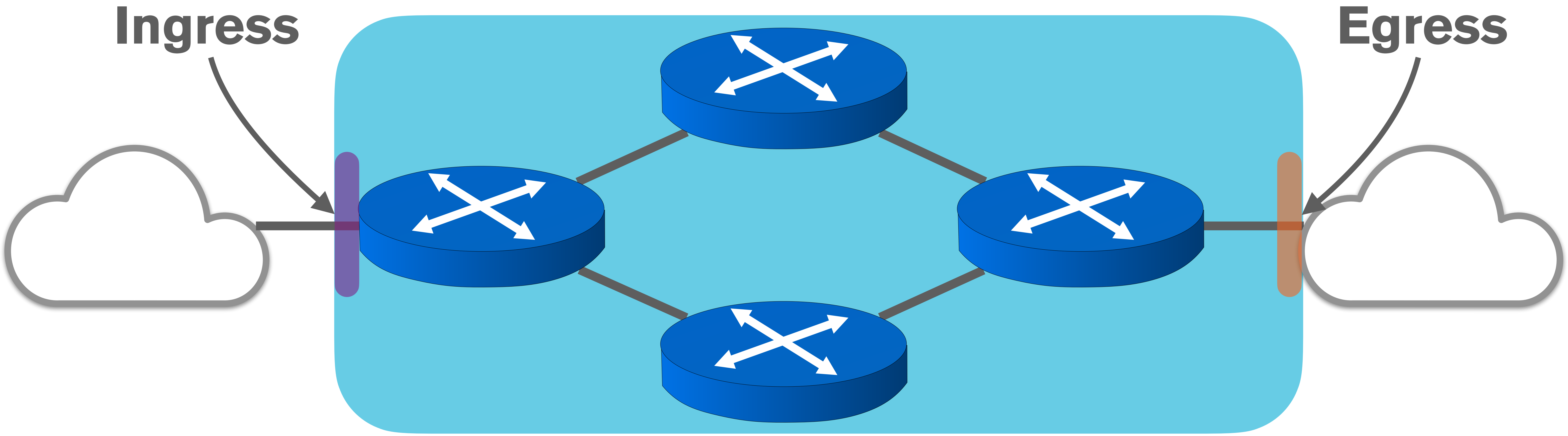


switch
Models the processing done at each switch

topology
Models the forwarding done by each link

ingress & egress
Models the perimeter of the network

Modeling networks in NetKAT



switch
Models the processing done at each switch

topology
Models the forwarding done by each link

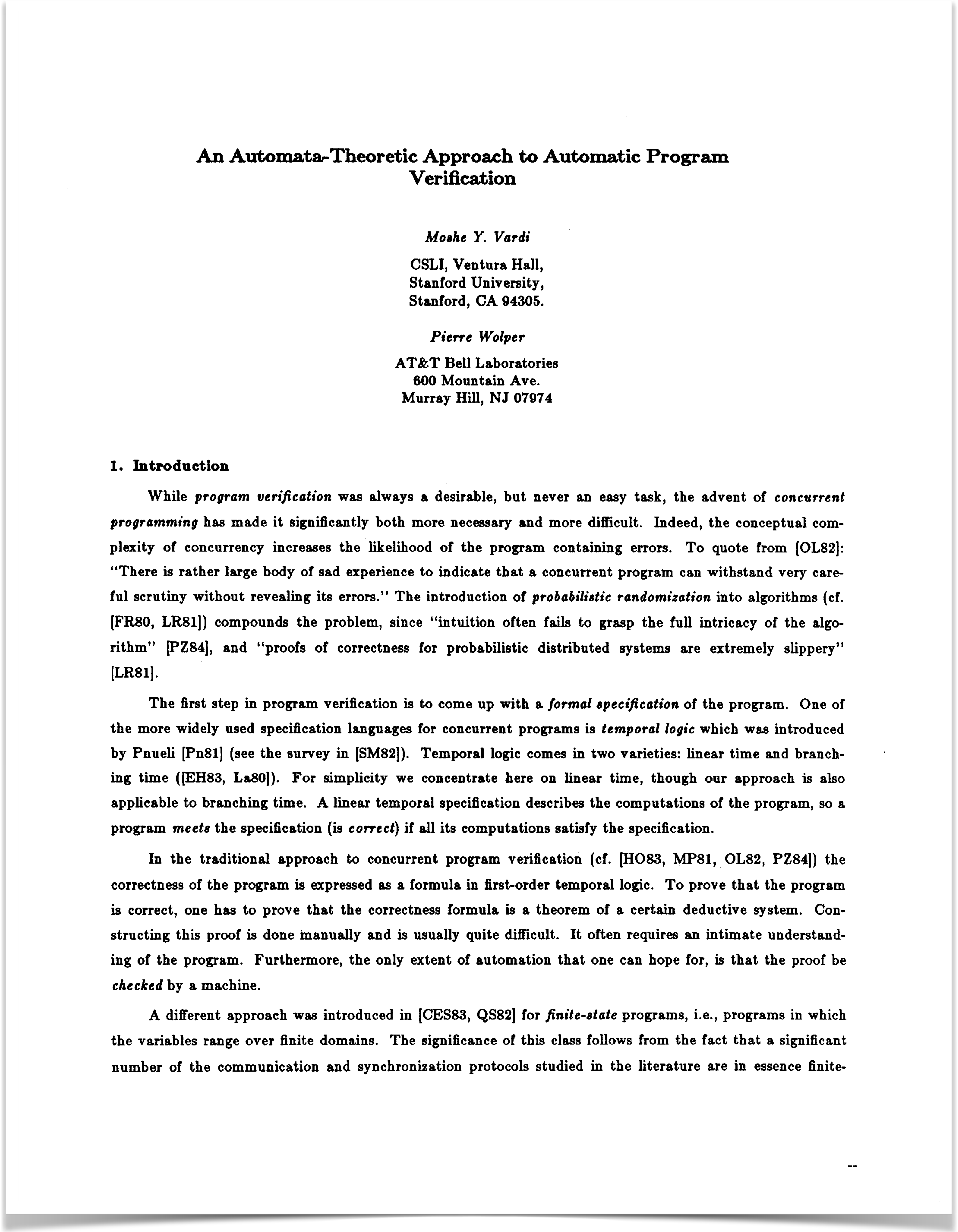
ingress & egress
Models the perimeter of the network

```
ingress ; (switch ; topology)* ; egress
```

Verification via equivalence

Idea: encode the program and its specification in a unified framework, then check equivalence (or inclusion, etc.)

History: A classic approach, pioneered by Vardi & Wolper, and widely used in hardware and software verification


An Automata-Theoretic Approach to Automatic Program Verification

Moshe Y. Vardi

CSLI, Ventura Hall,
Stanford University,
Stanford, CA 94305.

Pierre Wolper

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

1. Introduction

While *program verification* was always a desirable, but never an easy task, the advent of *concurrent programming* has made it significantly both more necessary and more difficult. Indeed, the conceptual complexity of concurrency increases the likelihood of the program containing errors. To quote from [OL82]: "There is rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors." The introduction of *probabilistic randomization* into algorithms (cf. [FR80, LR81]) compounds the problem, since "intuition often fails to grasp the full intricacy of the algorithm" [PZ84], and "proofs of correctness for probabilistic distributed systems are extremely slippery" [LR81].

The first step in program verification is to come up with a *formal specification* of the program. One of the more widely used specification languages for concurrent programs is *temporal logic* which was introduced by Pnueli [Pn81] (see the survey in [SM82]). Temporal logic comes in two varieties: linear time and branching time ([EH83, La80]). For simplicity we concentrate here on linear time, though our approach is also applicable to branching time. A linear temporal specification describes the computations of the program, so a program *meets* the specification (is *correct*) if all its computations satisfy the specification.

In the traditional approach to concurrent program verification (cf. [HO83, MP81, OL82, PZ84]) the correctness of the program is expressed as a formula in first-order temporal logic. To prove that the program is correct, one has to prove that the correctness formula is a theorem of a certain deductive system. Constructing this proof is done manually and is usually quite difficult. It often requires an intimate understanding of the program. Furthermore, the only extent of automation that one can hope for, is that the proof be *checked* by a machine.

A different approach was introduced in [CES83, QS82] for *finite-state* programs, i.e., programs in which the variables range over finite domains. The significance of this class follows from the fact that a significant number of the communication and synchronization protocols studied in the literature are in essence finite-

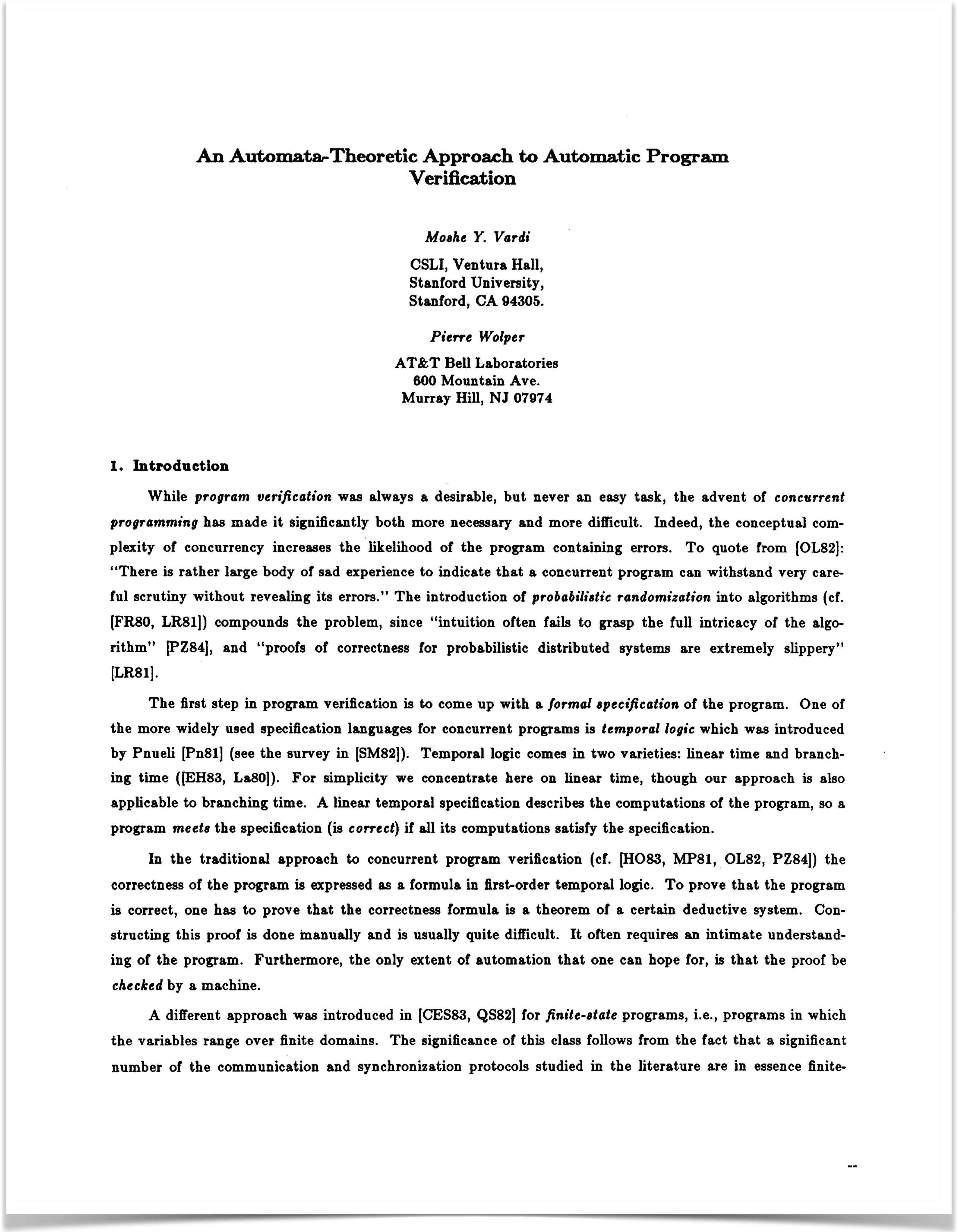
Verification via equivalence

Idea: encode the program and its specification in a unified framework, then check equivalence (or inclusion, etc.)

History: A classic approach, pioneered by Vardi & Wolper, and widely used in hardware and software verification

Questions:

1. Can NetKAT encode useful specifications?
2. Is program equivalence decidable (and if so, can we come up with practical approaches for checking it?)



An Automata-Theoretic Approach to Automatic Program Verification

Moshe Y. Vardi

CSLI, Ventura Hall,
Stanford University,
Stanford, CA 94305.

Pierre Wolper

AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

1. Introduction

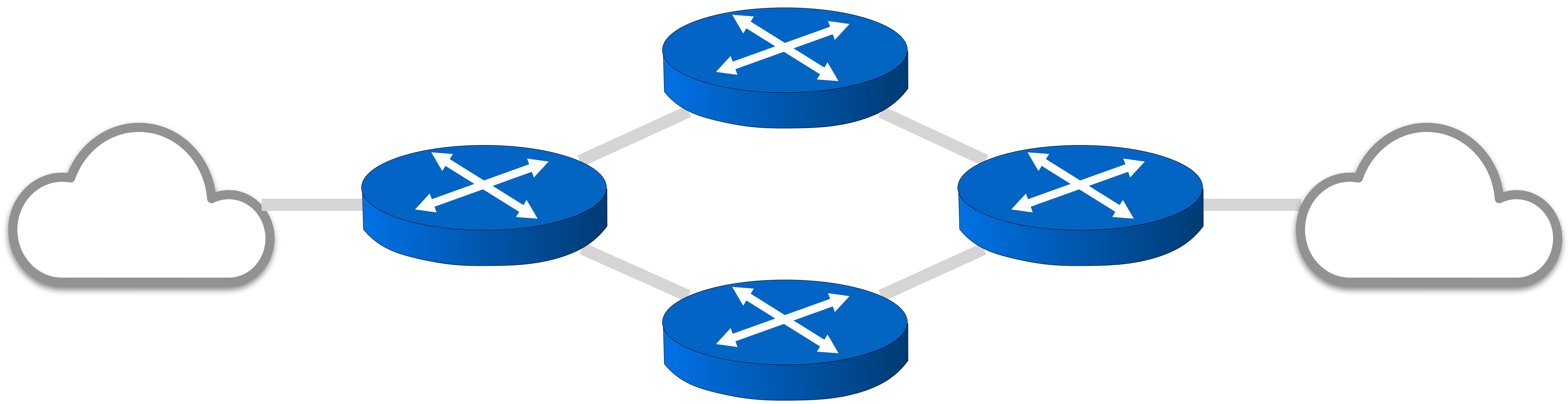
While *program verification* was always a desirable, but never an easy task, the advent of *concurrent programming* has made it significantly both more necessary and more difficult. Indeed, the conceptual complexity of concurrency increases the likelihood of the program containing errors. To quote from [OL82]: "There is rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors." The introduction of *probabilistic randomization* into algorithms (cf. [FR80, LR81]) compounds the problem, since "intuition often fails to grasp the full intricacy of the algorithm" [PZ84], and "proofs of correctness for probabilistic distributed systems are extremely slippery" [LR81].

The first step in program verification is to come up with a *formal specification* of the program. One of the more widely used specification languages for concurrent programs is *temporal logic* which was introduced by Pnueli [Pn81] (see the survey in [SM82]). Temporal logic comes in two varieties: linear time and branching time ([EH83, La80]). For simplicity we concentrate here on linear time, though our approach is also applicable to branching time. A linear temporal specification describes the computations of the program, so a program *meets* the specification (is *correct*) if all its computations satisfy the specification.

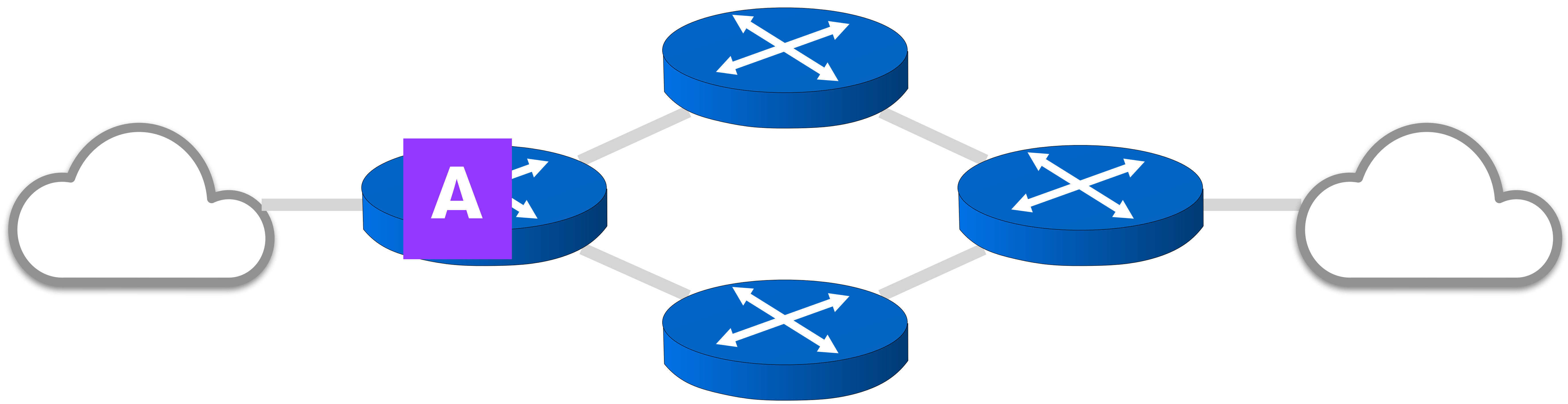
In the traditional approach to concurrent program verification (cf. [HO83, MP81, OL82, PZ84]) the correctness of the program is expressed as a formula in first-order temporal logic. To prove that the program is correct, one has to prove that the correctness formula is a theorem of a certain deductive system. Constructing this proof is done manually and is usually quite difficult. It often requires an intimate understanding of the program. Furthermore, the only extent of automation that one can hope for, is that the proof be *checked* by a machine.

A different approach was introduced in [CES83, QS82] for *finite-state* programs, i.e., programs in which the variables range over finite domains. The significance of this class follows from the fact that a significant number of the communication and synchronization protocols studied in the literature are in essence finite-

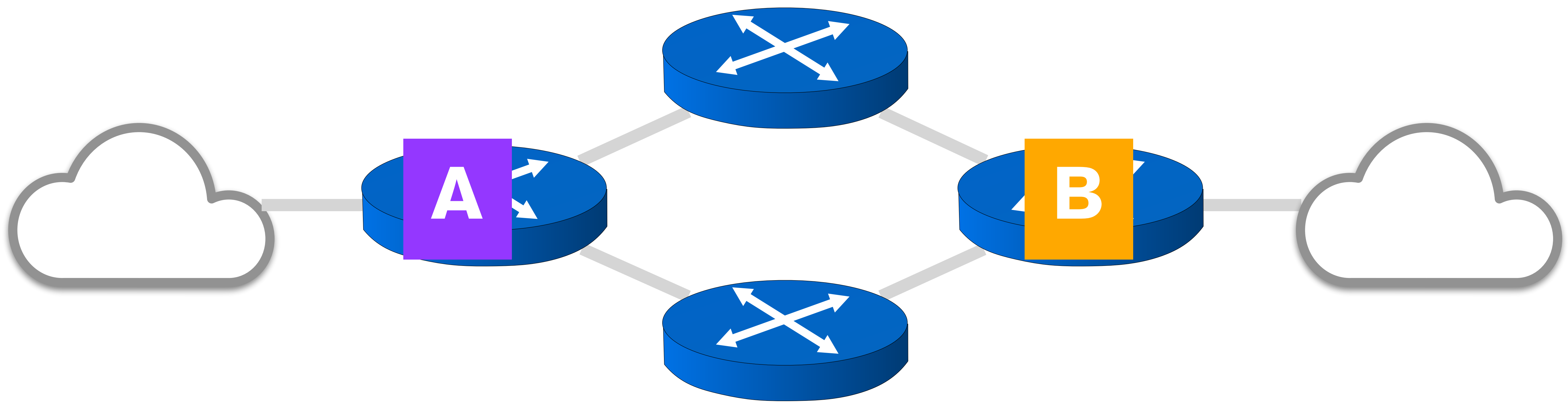
Example: reachability



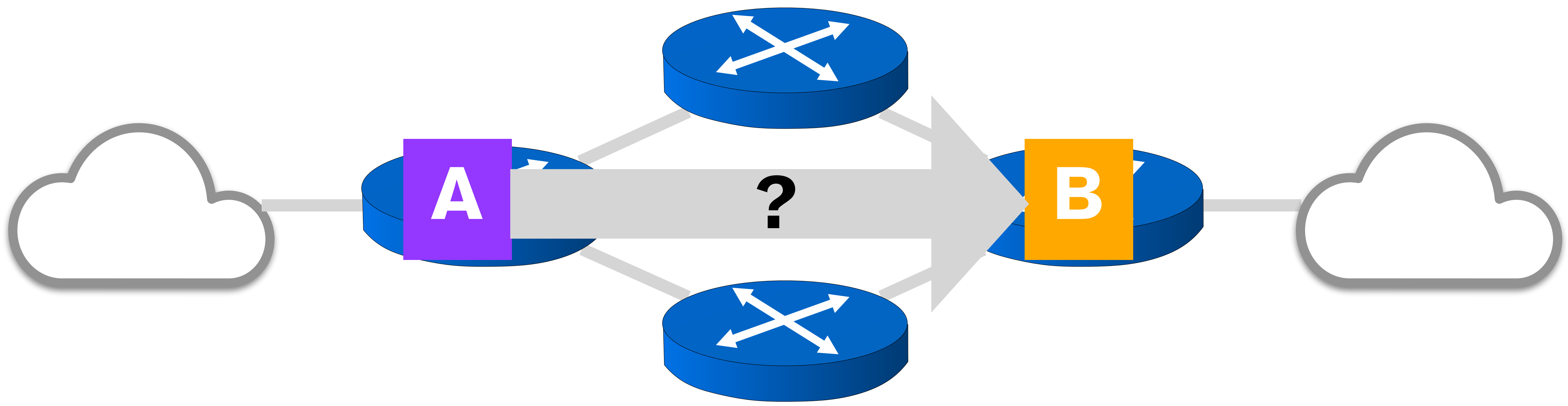
Example: reachability



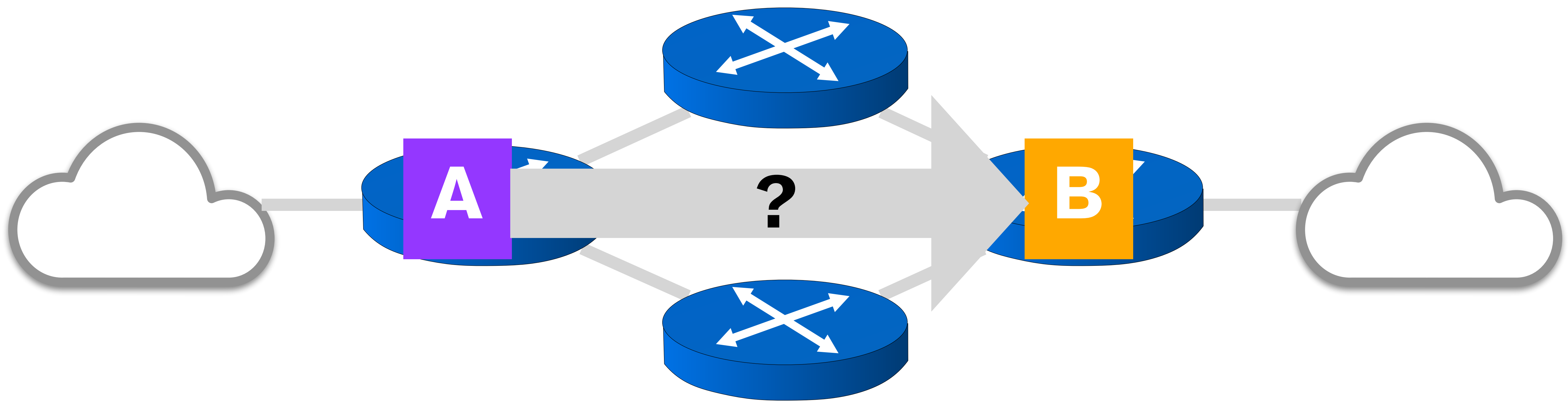
Example: reachability



Example: reachability



Example: reachability

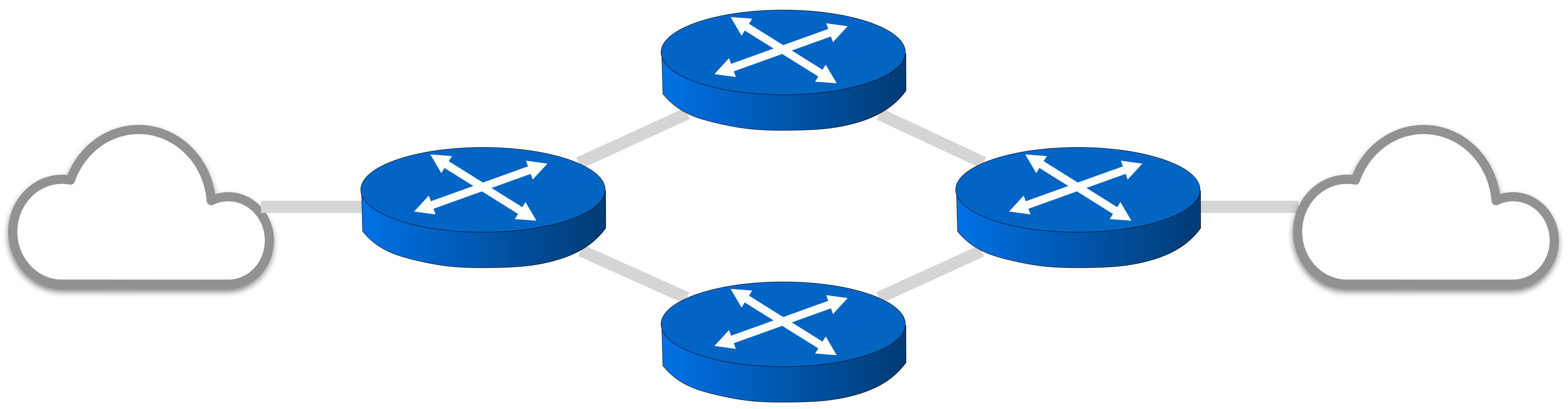


Property: B reachable from A

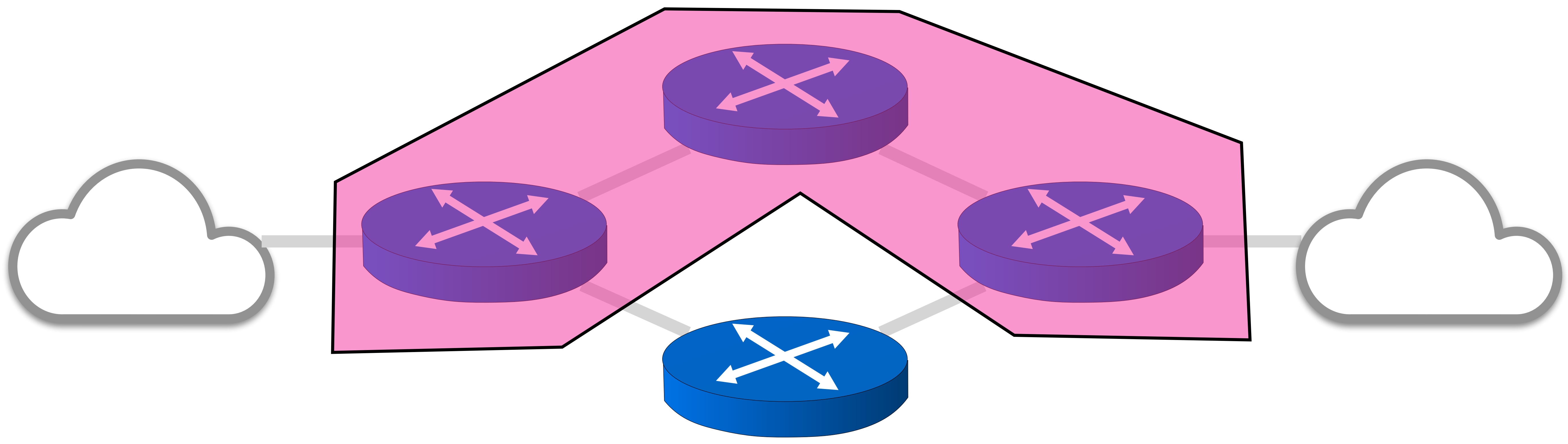
Approach:
- Build a model with A as ingress and B as egress
- Check for non-emptiness

Query: `switch=A; (switch; topo)*; switch=B ≢ drop`

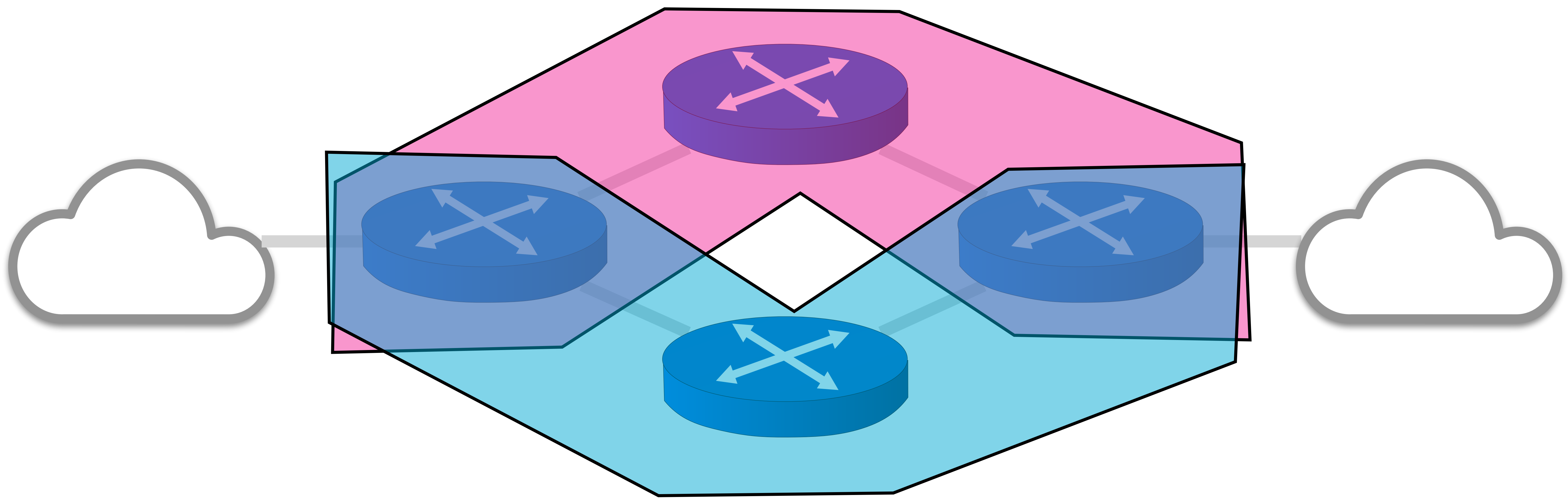
Example: isolation



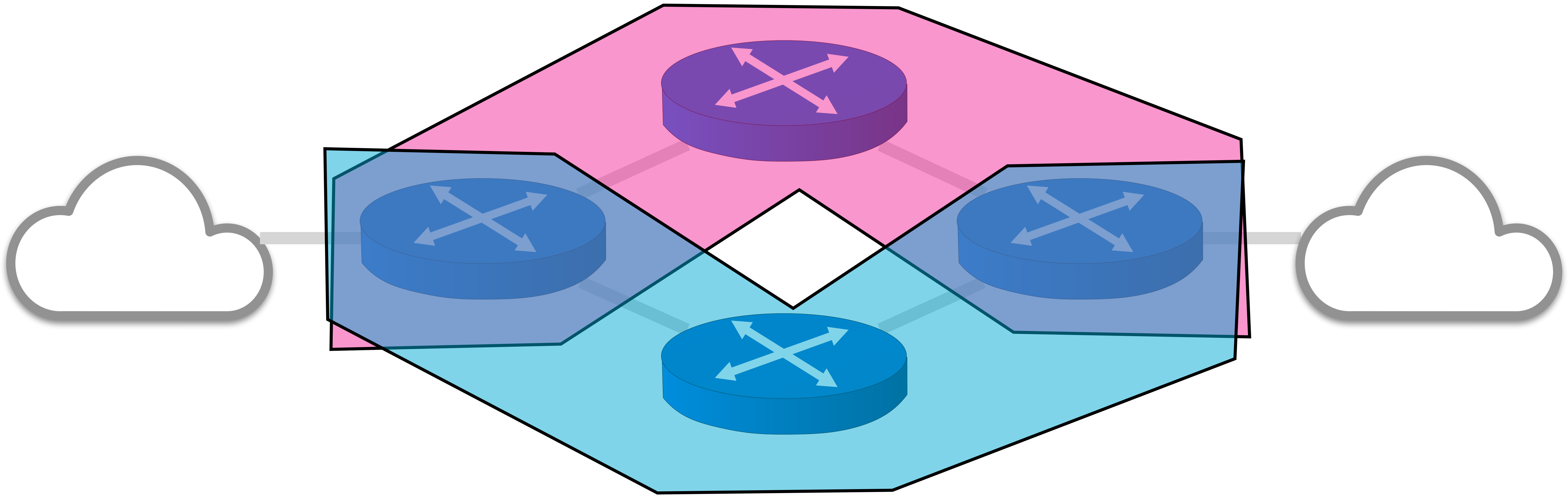
Example: isolation



Example: isolation



Example: isolation



Property: slice s logically isolated from q

Approach: Check that running s and q together is equivalent to running them independently on separate “copies” of the network

Query: `in; ([s + q] ; topo)*; eg ≡`

`[in; (s ; topo)*; eg] + [in; (q ; topo)*; eg]`

Aside: NetKAT's origin story

Machine-Verified Network Controllers

Arjun Guha
Cornell University
arjun@cs.cornell.edu

Mark Reitblatt
Cornell University
reitblatt@cs.cornell.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Abstract

In many areas of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to help programmers build reliable systems. But although networks are critical infrastructure, they have largely resisted analysis using formal techniques. Software-defined networking (SDN) is a new network architecture that has the potential to provide a foundation for network reasoning, by standardizing the interfaces used to express network programs and giving them a precise semantics.

This paper describes the design and implementation of the first machine-verified SDN controller. Starting from the foundations, we develop a detailed operational model for OpenFlow (the most popular SDN platform) and formalize it in the Coq proof assistant. We then use this model to develop a verified compiler and run-time system for a high-level network programming language. We identify bugs in existing languages and tools built without formal foundations, and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical applications.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Software-defined networking, OpenFlow, formal verification, Coq, domain-specific languages, NetCore, Frenetic.

1. Introduction

Networks are some of the most critical infrastructure in modern society and also some of the most fragile! Networks fail with alarming frequency, often due to simple misconfigurations or software bugs [8, 19, 30]. The recent news headlines contain numerous examples of network failures leading to disruptions: a configuration error during routine maintenance at Amazon triggered a sequence of cascading failures that brought down a datacenter and the customer machines hosted there; a corrupted routing table at GoDaddy disconnected their domain name servers for a day and caused a widespread outage; and a network connectivity issue at United Airlines took down their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub.

One way to make networks more reliable would be to develop tools for checking important network invariants automatically. These tools would allow administrators to answer questions such as: "does this configuration provide connectivity to every host in the network?" or "does this configuration correctly enforce the access control policy?" or "does this configuration have a forwarding loop?" or "does this configuration properly isolate trusted and untrusted traffic?" Unfortunately, until recently, building such tools has been effectively impossible due to the complexity of today's networks. A typical enterprise or datacenter network contains thousands of heterogeneous devices, from routers and switches, to web caches and load balancers, to monitoring middleboxes and firewalls. Moreover, each device executes a stack of complex protocols and is configured through a proprietary and idiosyncratic interface. To reason formally about such a network, an administrator (or tool) must reason about the proprietary programs running on each distributed device, as well as the asynchronous interactions between them. Although formal models of traditional networks exist, they have either been too complex to allow effective reasoning, or too abstract to be useful. Overall, the incidental complexity of networks has made reasoning about their behavior practically infeasible.

Fortunately, recent years have seen growing interest in a new kind of network architecture that could provide a foundation for network reasoning. In a *software-defined network* (SDN), a program on a logically-centralized *controller machine* defines the overall policy for the network, and a collection of *programmable switches* implement the policy using efficient packet-processing hardware. The controller and switches communicate via an open and standard interface. By carefully installing packet-processing rules in the hardware tables provided on switches, the controller can effectively manage the behavior of the entire network.

Compared to traditional networks, SDNs have two important simplifications that make them amenable to formal reasoning. First, they relocate control from distributed algorithms running on individual devices to a single program running on the controller. Second, they eliminate the heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, etc.—and replace them with stock programmable switches that provide a standard set of features. Together, this means that the behavior of the network is determined solely by the sequence of configuration instructions issued by the controller. To verify that the network has some property, an administrator (or tool) simply has to reason about the states of the switches as they process instructions.

In the networking community, there is burgeoning interest in tools for checking network-wide properties automatically. Systems such as FlowChecker [1], Header Space Analysis [12], Anteater [17], VeriFlow [13], and others, work by generating a logical representation of switch configurations and using an automatic solver to check properties of those configurations. The configurations are obtained by "scraping" state off of the switches or inspecting the instructions issued by an SDN controller at run-time.

These tools represent a good first step toward making networks more reliable, but they have two important limitations. First, they are based on ad hoc foundations. Although SDN platforms such as OpenFlow [21] have precise (if informal) specifications, the tools make simplifying assumptions that are routinely violated by real

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

Aside: NetKAT's origin story

Machine-Verified Network Controllers

Arjun Guha
Cornell University
arjun@cs.cornell.edu

Mark Reitblatt
Cornell University
reitblatt@cs.cornell.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Abstract

In many areas of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to help programmers build reliable systems. But although networks are critical infrastructure, they have largely resisted analysis using formal techniques. Software-defined networking (SDN) is a new network architecture that has the potential to provide a foundation for network reasoning, by standardizing the interfaces used to express network programs and giving them a precise semantics.

This paper describes the design and implementation of the first machine-verified SDN controller. Starting from the foundations, we develop a detailed operational model for OpenFlow (the most popular SDN platform) and formalize it in the Coq proof assistant. We then use this model to develop a verified compiler and run-time system for a high-level network programming language. We identify bugs in existing languages and tools built without formal foundations, and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical applications.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Software-defined networking, OpenFlow, formal verification, Coq, domain-specific languages, NetCore, Frenetic.

1. Introduction

Networks are some of the most critical infrastructure in modern society and also some of the most fragile! Networks fail with alarming frequency, often due to simple misconfigurations or software bugs [8, 19, 30]. The recent news headlines contain numerous examples of network failures leading to disruptions: a configuration error during routine maintenance at Amazon triggered a sequence of cascading failures that brought down a datacenter and the customer machines hosted there; a corrupted routing table at GoDaddy disconnected their domain name servers for a day and caused a widespread outage; and a network connectivity issue at United Airlines took down their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub.

One way to make networks more reliable would be to develop tools for checking important network invariants automatically. These tools would allow administrators to answer questions such as: "does this configuration provide connectivity to every host in the network?" or "does this configuration correctly enforce the access control policy?" or "does this configuration have a forwarding loop?" or "does this configuration properly isolate trusted and untrusted traffic?" Unfortunately, until recently, building such tools has been effectively impossible due to the complexity of today's networks. A typical enterprise or datacenter network contains thousands of heterogeneous devices, from routers and switches, to web caches and load balancers, to monitoring middleboxes and firewalls. Moreover, each device executes a stack of complex protocols and is configured through a proprietary and idiosyncratic interface. To reason formally about such a network, an administrator (or tool) must reason about the proprietary programs running on each distributed device, as well as the asynchronous interactions between them. Although formal models of traditional networks exist, they have either been too complex to allow effective reasoning, or too abstract to be useful. Overall, the incidental complexity of networks has made reasoning about their behavior practically infeasible.

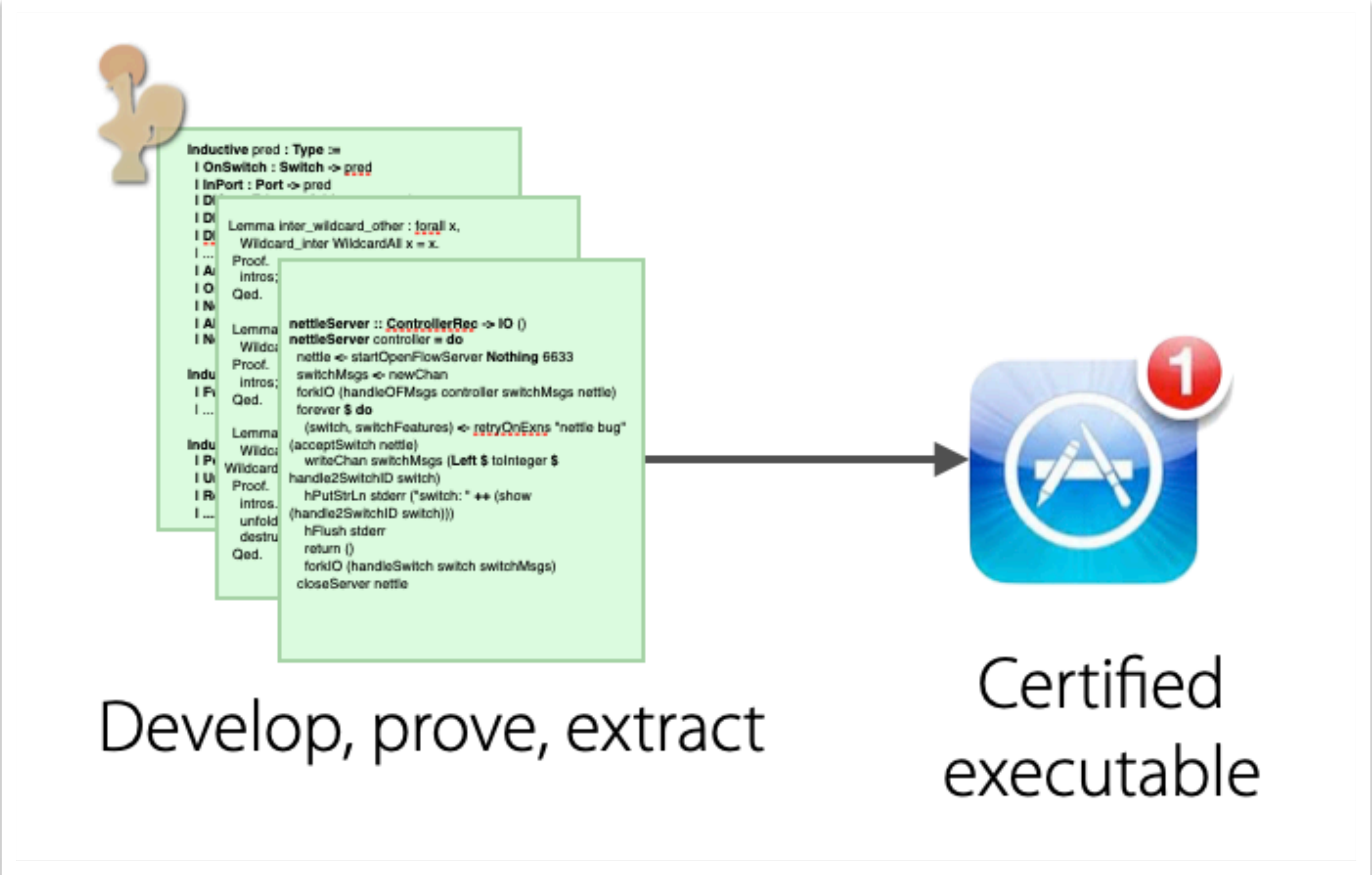
Fortunately, recent years have seen growing interest in a new kind of network architecture that could provide a foundation for network reasoning. In a *software-defined network* (SDN), a program on a logically-centralized *controller machine* defines the overall policy for the network, and a collection of *programmable switches* implement the policy using efficient packet-processing hardware. The controller and switches communicate via an open and standard interface. By carefully installing packet-processing rules in the hardware tables provided on switches, the controller can effectively manage the behavior of the entire network.

Compared to traditional networks, SDNs have two important simplifications that make them amenable to formal reasoning. First, they relocate control from distributed algorithms running on individual devices to a single program running on the controller. Second, they eliminate the heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, etc.—and replace them with stock programmable switches that provide a standard set of features. Together, this means that the behavior of the network is determined solely by the sequence of configuration instructions issued by the controller. To verify that the network has some property, an administrator (or tool) simply has to reason about the states of the switches as they process instructions.

In the networking community, there is burgeoning interest in tools for checking network-wide properties automatically. Systems such as FlowChecker [1], Header Space Analysis [12], Anteater [17], VeriFlow [13], and others, work by generating a logical representation of switch configurations and using an automatic solver to check properties of those configurations. The configurations are obtained by "scraping" state off of the switches or inspecting the instructions issued by an SDN controller at run-time.

These tools represent a good first step toward making networks more reliable, but they have two important limitations. First, they are based on ad hoc foundations. Although SDN platforms such as OpenFlow [21] have precise (if informal) specifications, the tools make simplifying assumptions that are routinely violated by real

PLDI'13, June 16–19, 2013, Seattle, WA, USA.



Aside: NetKAT's origin story

Machine-Verified Network Controllers

Arjun Guha
Cornell University
arjun@cs.cornell.edu

Mark Reitblatt
Cornell University
reitblatt@cs.cornell.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Abstract

In many areas of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to help programmers build reliable systems. But although networks are critical infrastructure, they have largely resisted analysis using formal techniques. Software-defined networking (SDN) is a new network architecture that has the potential to provide a foundation for network reasoning, by standardizing the interfaces used to express network programs and giving them a precise semantics.

This paper describes the design and implementation of the first machine-verified SDN controller. Starting from the foundations, we develop a detailed operational model for OpenFlow (the most popular SDN platform) and formalize it in the Coq proof assistant. We then use this model to develop a verified compiler and run-time system for a high-level network programming language. We identify bugs in existing languages and tools built without formal foundations, and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical applications.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Software-defined networking, OpenFlow, formal verification, Coq, domain-specific languages, NetCore, Frenetic.

1. Introduction

Networks are some of the most critical infrastructure in modern society and also some of the most fragile! Networks fail with alarming frequency, often due to simple misconfigurations or software bugs [8, 19, 30]. The recent news headlines contain numerous examples of network failures leading to disruptions: a configuration error during routine maintenance at Amazon triggered a sequence of cascading failures that brought down a datacenter and the customer machines hosted there; a corrupted routing table at GoDaddy disconnected their domain name servers for a day and caused a widespread outage; and a network connectivity issue at United Airlines took down their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub.

One way to make networks more reliable would be to develop tools for checking important network invariants automatically. These tools would allow administrators to answer questions such as: "does this configuration provide connectivity to every host in the network?" or "does this configuration correctly enforce the access control policy?" or "does this configuration have a forwarding loop?" or "does this configuration properly isolate trusted and untrusted traffic?" Unfortunately, until recently, building such tools has been effectively impossible due to the complexity of today's networks. A typical enterprise or datacenter network contains thousands of heterogeneous devices, from routers and switches, to web caches and load balancers, to monitoring middleboxes and firewalls. Moreover, each device executes a stack of complex protocols and is configured through a proprietary and idiosyncratic interface. To reason formally about such a network, an administrator (or tool) must reason about the proprietary programs running on each distributed device, as well as the asynchronous interactions between them. Although formal models of traditional networks exist, they have either been too complex to allow effective reasoning, or too abstract to be useful. Overall, the incidental complexity of networks has made reasoning about their behavior practically infeasible.

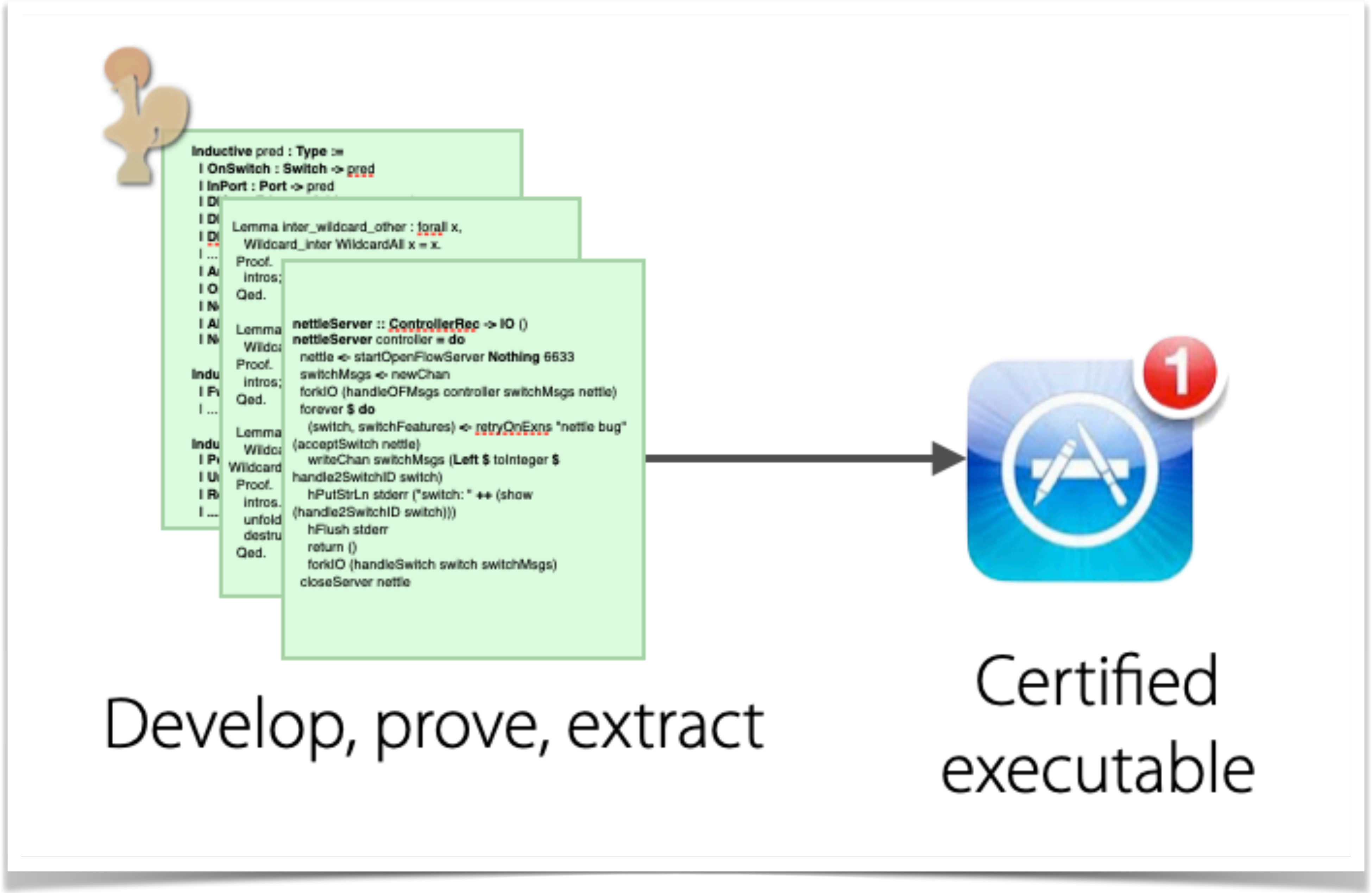
Fortunately, recent years have seen growing interest in a new kind of network architecture that could provide a foundation for network reasoning. In a *software-defined network* (SDN), a program on a logically-centralized *controller machine* defines the overall policy for the network, and a collection of *programmable switches* implement the policy using efficient packet-processing hardware. The controller and switches communicate via an open and standard interface. By carefully installing packet-processing rules in the hardware tables provided on switches, the controller can effectively manage the behavior of the entire network.

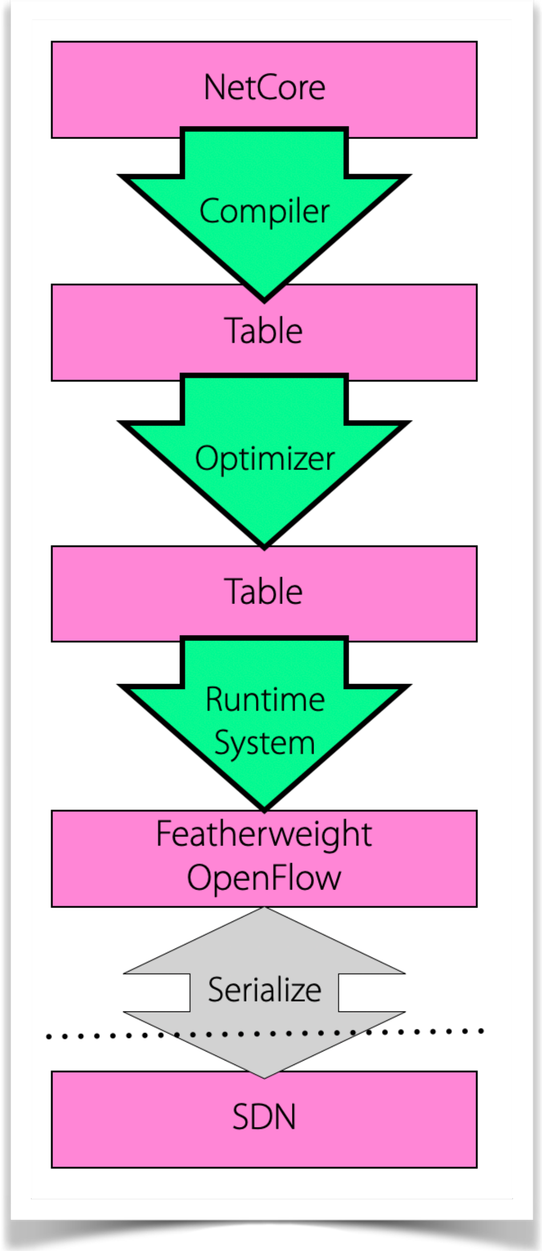
Compared to traditional networks, SDNs have two important simplifications that make them amenable to formal reasoning. First, they relocate control from distributed algorithms running on individual devices to a single program running on the controller. Second, they eliminate the heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, etc.—and replace them with stock programmable switches that provide a standard set of features. Together, this means that the behavior of the network is determined solely by the sequence of configuration instructions issued by the controller. To verify that the network has some property, an administrator (or tool) simply has to reason about the states of the switches as they process instructions.

In the networking community, there is burgeoning interest in tools for checking network-wide properties automatically. Systems such as FlowChecker [1], Header Space Analysis [12], Anteater [17], VeriFlow [13], and others, work by generating a logical representation of switch configurations and using an automatic solver to check properties of those configurations. The configurations are obtained by "scraping" state off of the switches or inspecting the instructions issued by an SDN controller at run-time.

These tools represent a good first step toward making networks more reliable, but they have two important limitations. First, they are based on ad hoc foundations. Although SDN platforms such as OpenFlow [21] have precise (if informal) specifications, the tools make simplifying assumptions that are routinely violated by real

PLDI'13, June 16–19, 2013, Seattle, WA, USA.





Aside: NetKAT's origin story

Machine-Verified Network Controllers

Arjun Guha, Cornell University, arjun@cs.cornell.edu
Mark Reitblatt, Cornell University, reitblatt@cs.cornell.edu
Nate Foster, Cornell University, jnfoster@cs.cornell.edu

Abstract

In many areas of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to help programmers build reliable systems. But although networks are critical infrastructure, they have largely resisted analysis using formal techniques. Software-defined networking (SDN) is a new network architecture that has the potential to provide a foundation for network reasoning, by standardizing the interfaces used to express network programs and giving them a precise semantics.

This paper describes the design and implementation of the first machine-verified SDN controller. Starting from the foundations, we develop a detailed operational model for OpenFlow (the most popular SDN platform) and formalize it in the Coq proof assistant. We then use this model to develop a verified compiler and run-time system for a high-level network programming language. We identify bugs in existing languages and tools built without formal foundations, and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical applications.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Software-defined networking, OpenFlow, formal verification, Coq, domain-specific languages, NetCore, Frenetic.

1. Introduction

Networks are some of the most critical infrastructure in modern society and also some of the most fragile! Networks fail with alarming frequency, often due to simple misconfigurations or software bugs [8, 19, 30]. The recent news headlines contain numerous examples of network failures leading to disruptions: a configuration error during routine maintenance at Amazon triggered a sequence of cascading failures that brought down a datacenter and the customer machines hosted there; a corrupted routing table at GoDaddy disconnected their domain name servers for a day and caused a widespread outage; and a network connectivity issue at United Airlines took down their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub.

One way to make networks more reliable would be to develop tools for checking important network invariants automatically. These tools would allow administrators to answer questions such as: "does this configuration provide connectivity to every host in the network?" or "does this configuration correctly enforce the access control policy?" or "does this configuration have a forwarding loop?" or "does this configuration properly isolate trusted and untrusted traffic?" Unfortunately, until recently, building such tools has been effectively impossible due to the complexity of today's networks. A typical enterprise or datacenter network contains thousands of heterogeneous devices, from routers and switches, to web caches and load balancers, to monitoring middleboxes and firewalls. Moreover, each device executes a stack of complex protocols and is configured through a proprietary and idiosyncratic interface. To reason formally about such a network, an administrator (or tool) must reason about the proprietary programs running on each distributed device, as well as the asynchronous interactions between them. Although formal models of traditional networks exist, they have either been too complex to allow effective reasoning, or too abstract to be useful. Overall, the incidental complexity of networks has made reasoning about their behavior practically infeasible.

Fortunately, recent years have seen growing interest in a new kind of network architecture that could provide a foundation for network reasoning. In a *software-defined network* (SDN), a program on a logically-centralized *controller machine* defines the overall policy for the network, and a collection of *programmable switches* implement the policy using efficient packet-processing hardware. The controller and switches communicate via an open and standard interface. By carefully installing packet-processing rules in the hardware tables provided on switches, the controller can effectively manage the behavior of the entire network.

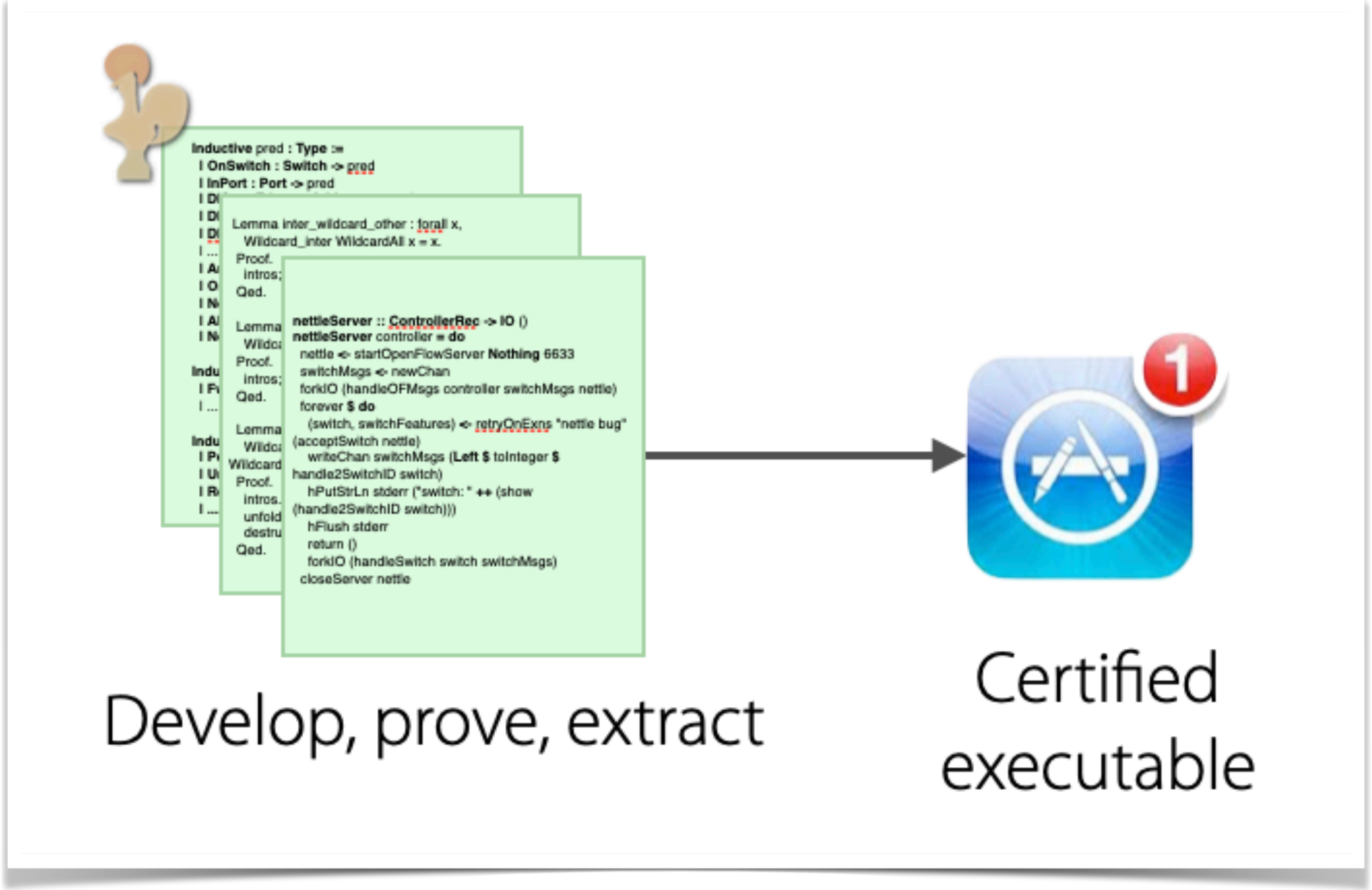
Compared to traditional networks, SDNs have two important simplifications that make them amenable to formal reasoning. First, they relocate control from distributed algorithms running on individual devices to a single program running on the controller. Second, they eliminate the heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, etc.—and replace them with stock programmable switches that provide a standard set of features. Together, this means that the behavior of the network is determined solely by the sequence of configuration instructions issued by the controller. To verify that the network has some property, an administrator (or tool) simply has to reason about the states of the switches as they process instructions.

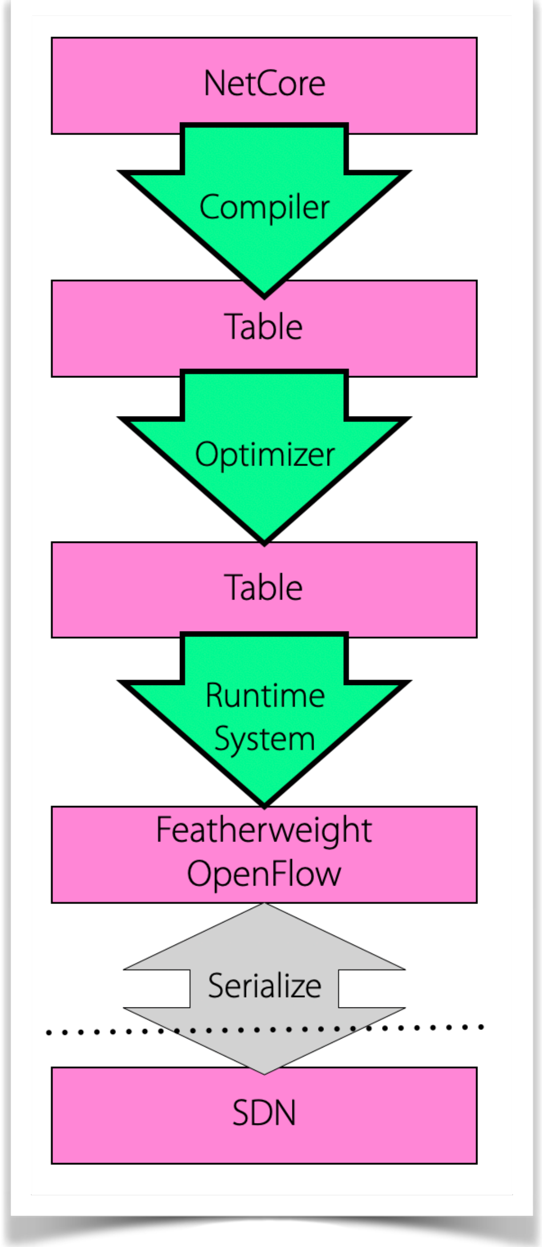
In the networking community, there is burgeoning interest in tools for checking network-wide properties automatically. Systems such as FlowChecker [1], Header Space Analysis [12], Anteater [17], VeriFlow [13], and others, work by generating a logical representation of switch configurations and using an automatic solver to check properties of those configurations. The configurations are obtained by "scraping" state off of the switches or inspecting the instructions issued by an SDN controller at run-time.

These tools represent a good first step toward making networks more reliable, but they have two important limitations. First, they are based on ad hoc foundations. Although SDN platforms such as OpenFlow [21] have precise (if informal) specifications, the tools make simplifying assumptions that are routinely violated by real

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

Ugh, I'm *sick* of re-doing
this compiler proof! But I model the
semantics as a semi-ring, I can factor out
most of it out!





Equational Axioms

Kleene Algebra Axioms [Kozen ’94]

p + (q + r) ≡ (p + q) + r
p + q ≡ q + p
p + drop ≡ p
p + p ≡ p
p; (q; r) ≡ (p; q); r
p; (q + r) ≡ p; q + p; r
(p + q); r ≡ p; r + q; r
id; p ≡ p
p ≡ p; id
drop; p ≡ drop
p; drop ≡ drop
id + p; p* ≡ p*
id + p*; p ≡ p*
p + q; r + r ≡ r ⇒ p*; q + r ≡ r
p + q; r + q ≡ q ⇒ p; r* + q ≡ q

Additional Boolean Algebra Axioms

a + (b; c) ≡ (a + b); (a + c)
a + id ≡ id
a + ! a ≡ id
a; b ≡ b; a
a; !a ≡ drop
a; a ≡ a

Packet Axioms (for f≠g, n≠m)

f := n; g := m ≡ g := m; f := n
f := n; g = m ≡ g = m; f := n
f := n; f = n ≡ f := n
f = n; f := n ≡ f = n
f := n; f := m ≡ f := m
f = n; f = m ≡ false
dup; f = n ≡ f = n; dup
Σ_i f = n_i ≡ true

Equational Axioms

Kleene Algebra Axioms [Kozen ’94]

p + (q + r) ≡ (p + q) + r
p + q ≡ q + p
p + drop ≡ p
p + p ≡ p
p; (q
p; (q
(p +
id; p
p ≡ p
drop; p ≡ drop
p; drop ≡ drop
id + p; p* ≡ p*
id + p*; p ≡ p*
p + q; r + r ≡ r ⇒ p*; q + r ≡ r
p + q; r + q ≡ q ⇒ p; r* + q ≡ q

Additional Boolean Algebra Axioms

a + (b; c) ≡ (a + b); (a + c)
a + id ≡ id
a + ! a ≡ id
a; b ≡ b; a

n
f := n; f = n ≡ f := n
f = n; f := n ≡ f = n
f := n; f := m ≡ f := m
f = n; f = m ≡ false
dup; f = n ≡ f = n; dup
Σ_i f = n_i ≡ true

Soundness: If ⊢ p ≡ q, then ⟦p⟧ = ⟦q⟧

Completeness: If ⟦p⟧ = ⟦q⟧, then ⊢ p ≡ q

NetKAT automata

We can also build automata that recognize packet traces

A *NetKAT Automaton* is a tuple $M=\langle S, s_0, \varepsilon, \delta \rangle$ where:

- S is a finite set of states,
- $s_0 \in S$ is the start state,
- $\varepsilon \in S \to \text{Packet} \to \text{Packet Set}$
- $\delta \in S \to \text{Packet} \to (S * \text{Packet}) \text{ Set}$

NetKAT automata

We can also build automata that recognize packet traces

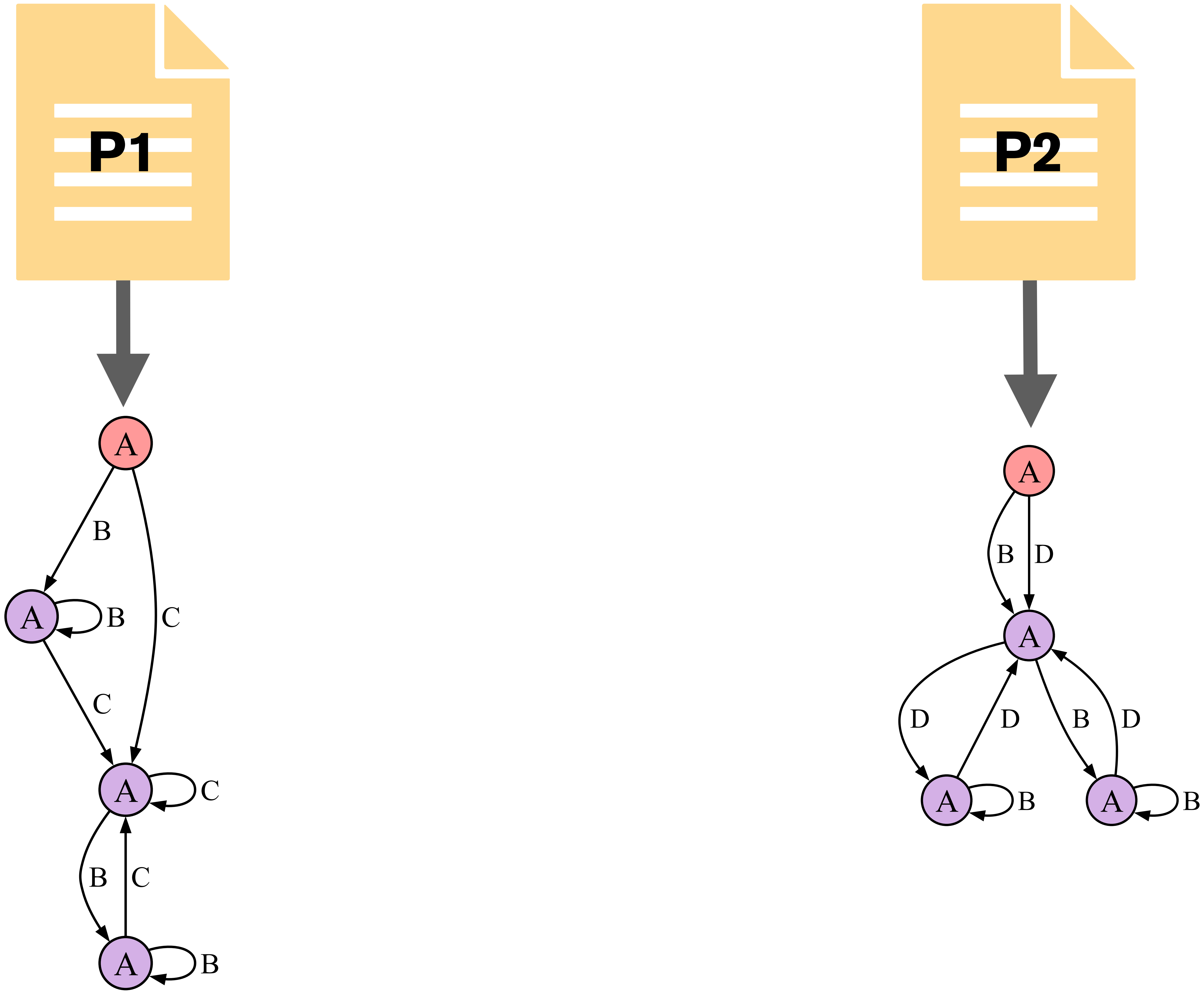
A *NetKAT Automaton* is a tuple $M = \langle S, s_0, \varepsilon, \delta \rangle$ where:

- S is a finite set of states,
- $s_0 \in S$ is the start state,
- $\varepsilon \in S \to \text{Packet} \to \text{Packet Set}$
- $\delta \in S \to \text{Packet} \to (S * \text{Packet}) \text{ Set}$

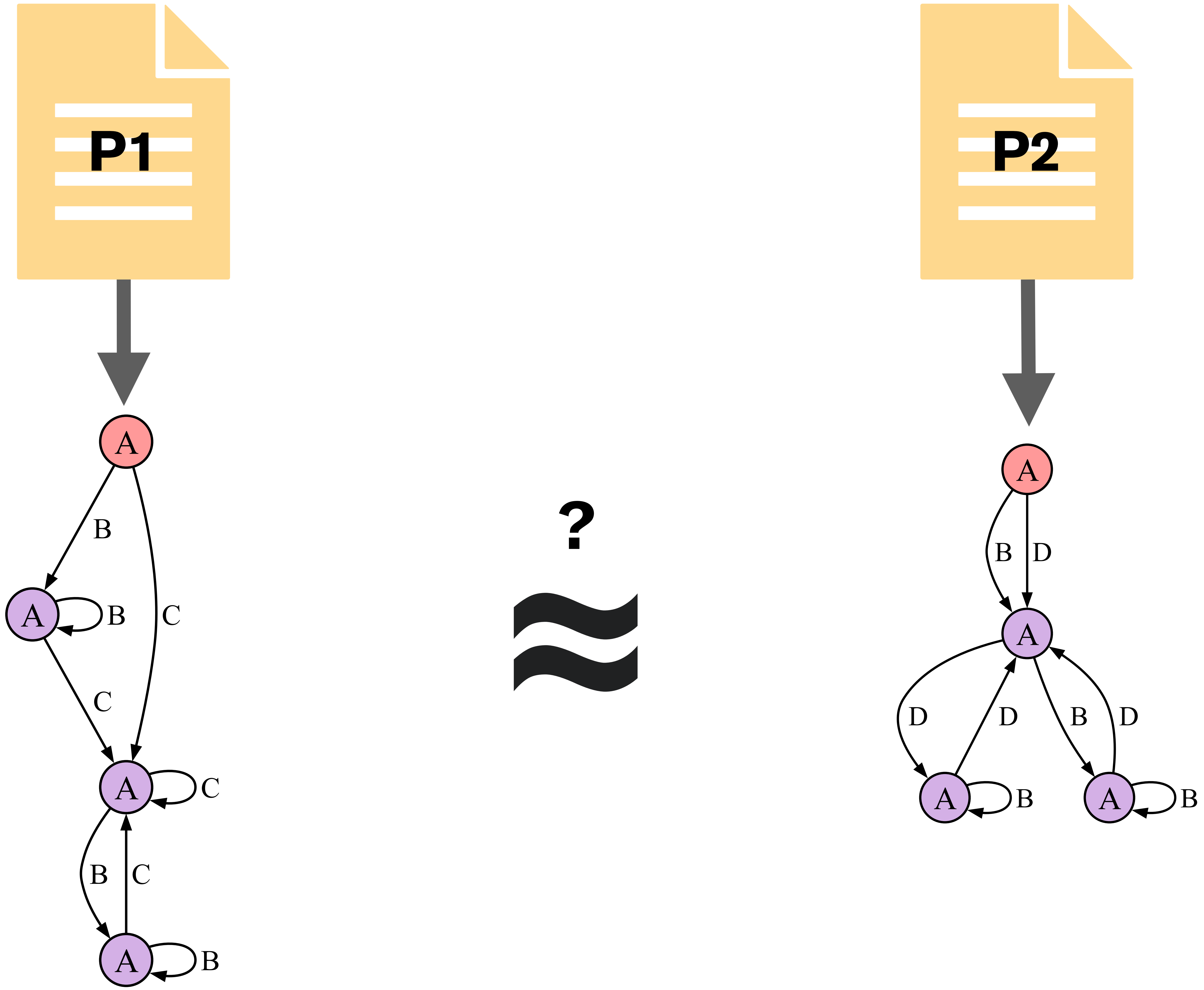
M *accepts* a trace in state s if:

- accept s [p, p'] $\Leftrightarrow$ $p' \in \varepsilon\ s\ p$
- accept s [p, p'] @ rest $\Leftrightarrow$ $\exists\, s'.\ (p', s') \in \delta\ s\ p \wedge$ accept s' (p' @ rest)

Checking equivalence



Checking equivalence



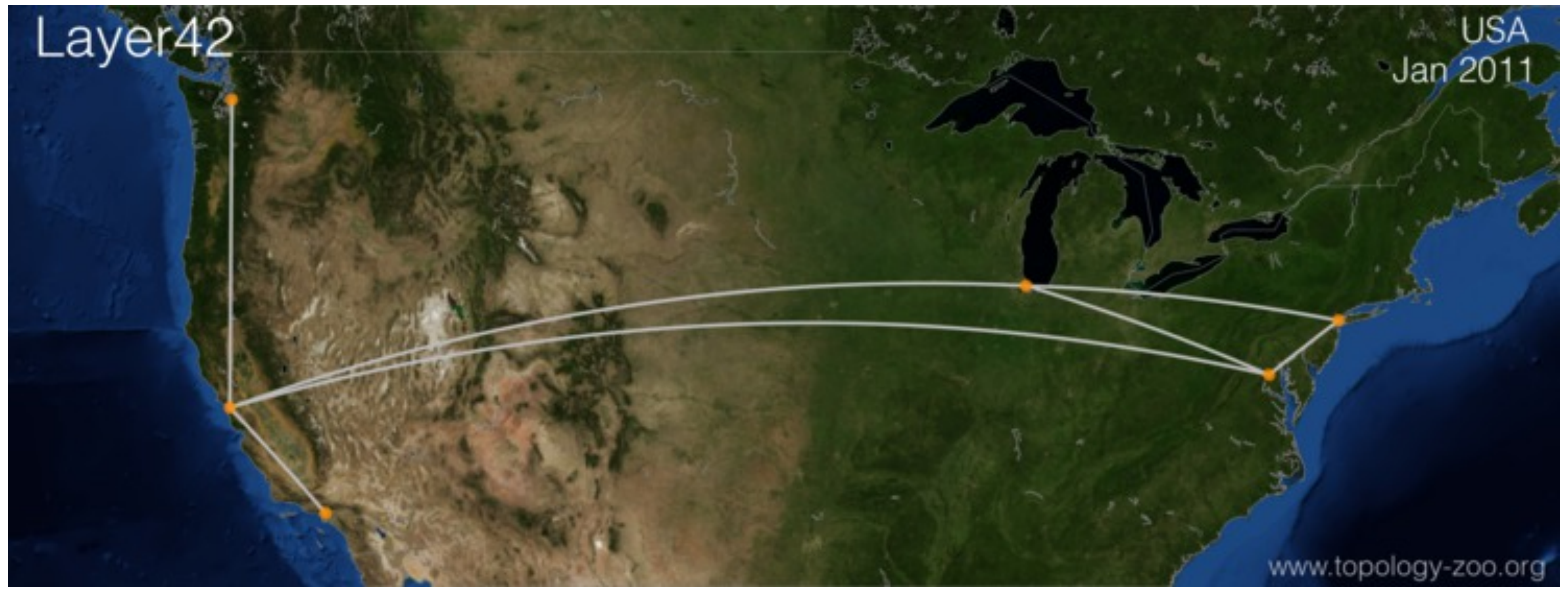
Evaluation: benchmark suite

Dataset
Internet Topology Zoo, a dataset of 140 real-world topologies, mostly large ISPs

Configurations
Synthetic programs that forward traffic along shortest paths

Property
All-pairs reachability

Key question
Performance relative to APKeep, a state-of-the-art network verification tool



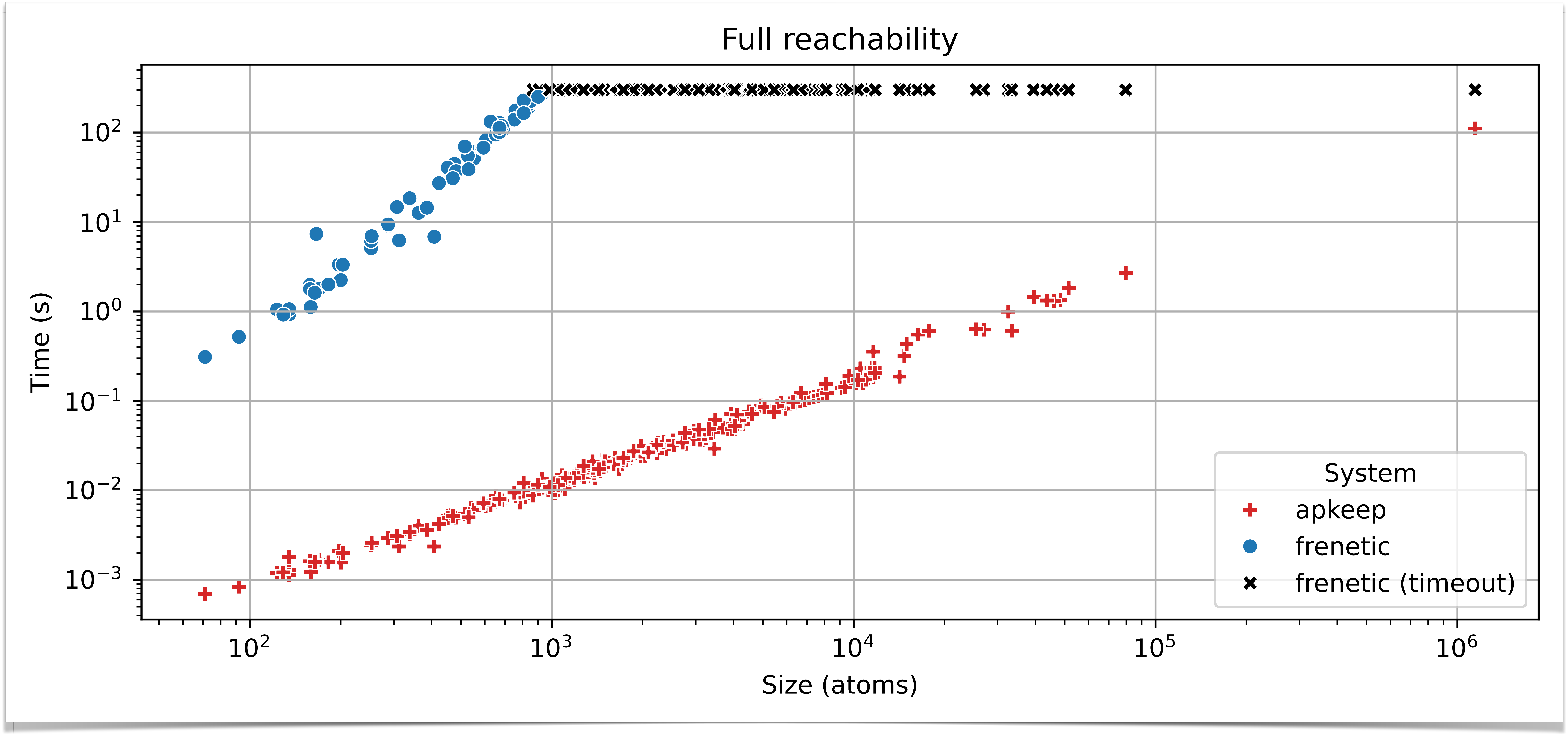
Topology

```
N0 = 0 N1 = 1 N2 = 2 N3 = 3 N4 = 4 N5 = 5
top = @pt=-1?·εU(@sw=N5?·(@pt=2?·(@sw←N4·@pt←1)U@pt=1?
·(@sw←N3·@pt←2)U@pt=0?·(@sw←N1·@pt←3))U@sw=N4?·(@pt=1?
·(@sw←N5·@pt←2)U@pt=0?·(@sw←N3·@pt←1))U@sw=N3?·(@pt=2?
·(@sw←N5·@pt←1)U@pt=1?·(@sw←N4·@pt←0)U@pt=0?
·(@sw←N1·@pt←2))U@sw=N2?·(@pt=0?·(@sw←N1·@pt←1))U@sw=N1?·(@pt=3?
·(@sw←N5·@pt←0)U@pt=2?·(@sw←N3·@pt←0)U@pt=1?·(@sw←N2·@pt←0)U@pt=0?
·(@sw←N0·@pt←0))U@sw=N0?·(@pt=0?·(@sw←N1·@pt←0)))
```

Switches

```
 @sw=N5?·(@dst=N4?·@pt←2U@dst=N3?·@pt←1U@dst=N2?·@pt←0U@dst=N1?
·@pt←0U@dst=N0?·@pt←0U@dst=N5?·@pt←-1)U@sw=N4?·(@dst=N5?
·@pt←1U@dst=N3?·@pt←0U@dst=N2?·@pt←0U@dst=N1?·@pt←0U@dst=N0?
·@pt←0U@dst=N4?·@pt←-1)U@sw=N3?·(@dst=N5?·@pt←2U@dst=N4?
·@pt←1U@dst=N2?·@pt←0U@dst=N1?·@pt←0U@dst=N0?·@pt←0U@dst=N3?
·@pt←-1)U@sw=N2?·(@dst=N5?·@pt←0U@dst=N4?·@pt←0U@dst=N3?
·@pt←0U@dst=N1?·@pt←0U@dst=N0?·@pt←0U@dst=N2?·@pt←-1)U@sw=N1?
·(@dst=N5?·@pt←3U@dst=N4?·@pt←3U@dst=N3?·@pt←2U@dst=N2?
·@pt←1U@dst=N0?·@pt←0U@dst=N1?·@pt←-1)U@sw=N0?·(@dst=N5?
·@pt←0U@dst=N4?·@pt←0U@dst=N3?·@pt←0U@dst=N2?·@pt←0U@dst=N1?
·@pt←0U@dst=N0?·@pt←-1)
```

Evaluation


Full reachability
Time (s)
Size (atoms)
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
System
apkeep
frenetic
frenetic (timeout)

Symbolic automata [PLDI '24]

A *NetKAT Automaton* is a tuple M=⟨S, s_0, ε, δ⟩ where:

- …
- ε ∈ S → **Packet** → **Packet Set**
- δ ∈ S → **Packet** → (S * **Packet) Set**

Symbolic automata I ’24]

A *NetKAT Automaton* is a tuple $M = \langle S, s_0, \varepsilon,$

- …
- $\varepsilon \in S \rightarrow$ **Packet** $\rightarrow$ **Packet Set**
- $\delta \in S \rightarrow$ **Packet** $\rightarrow$ (S * **Packet) Set**

There are a *lot* of packets (and even more relations on packets)!

Symbolic automat… '24]

There are a *lot* of packets (and even more relations on packets)!

A *NetKAT Automaton* is a tuple M=⟨S, s_0, ε,

- …
- ε ∈ S → **Packet** → **Packet Set**
- δ ∈ S → **Packet** → (S * **Packet) Set**

Idea: encode relations using a symbolic representation that is optimized for the common case–i.e., most fields are not changed

Symbolic automata ... '24]

There are a *lot* of packets (and even more relations on packets)!

A *NetKAT Automaton* is a tuple M=⟨S, s_0, ε, ...

- ...
- ε ∈ S → **Packet** → **Packet Set**
- δ ∈ S → **Packet** → (S * **Packet) Set**

Idea: encode relations using a symbolic representation that is optimized for the common case–i.e., most fields are not changed

SPPs: two-layer binary decision diagrams where first layer encodes predicates and second layer encodes modifications

Symbolic automat… '24]

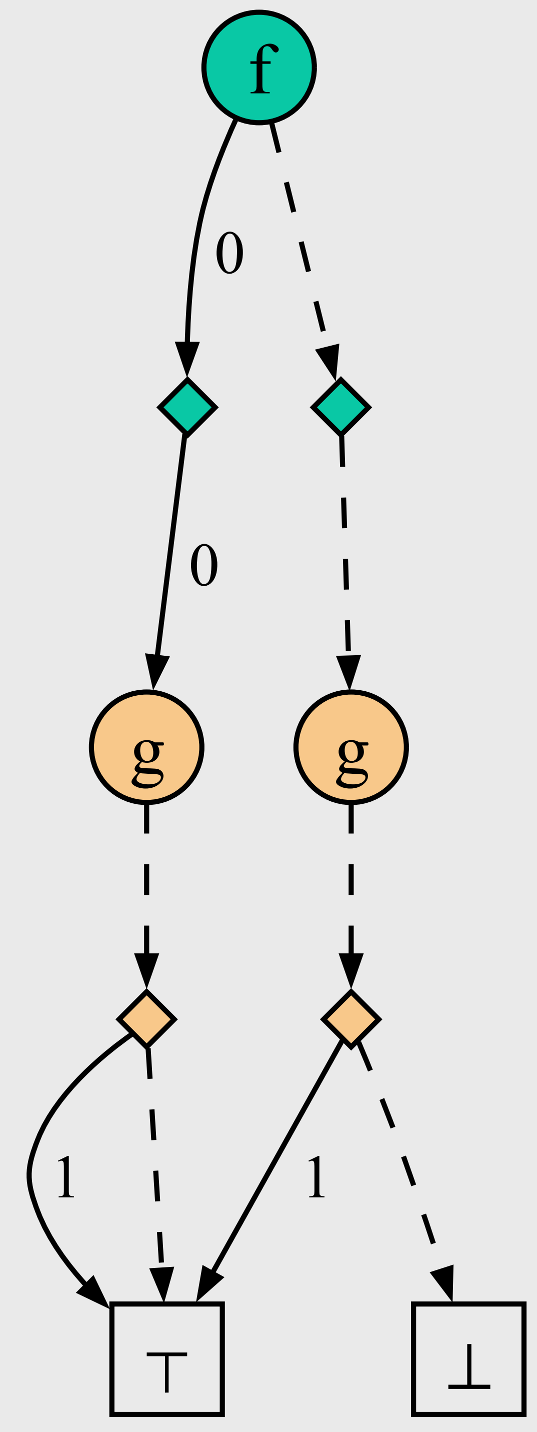
A *NetKAT Automaton* is a tuple M=⟨S, s_0, ε,

- …
- ε ∈ S → **Packet** → **Packet Set**
- δ ∈ S → **Packet** → (S * **Packet) Set**

There are a *lot* of packets (and even more relations on packets)!

Example

```
f = 0 + g := 1
```



Idea: encode relations using a symbolic representation that is optimized for the common case—i.e., most fields are not changed

SPPs: two-layer binary decision diagrams where first layer encodes predicates and second layer encodes modifications

Symbolic automata [... '24]

There are a *lot* of packets (and even more relations on packets)!

A *NetKAT Automaton* is a tuple M=⟨S, s_0, ε, ...

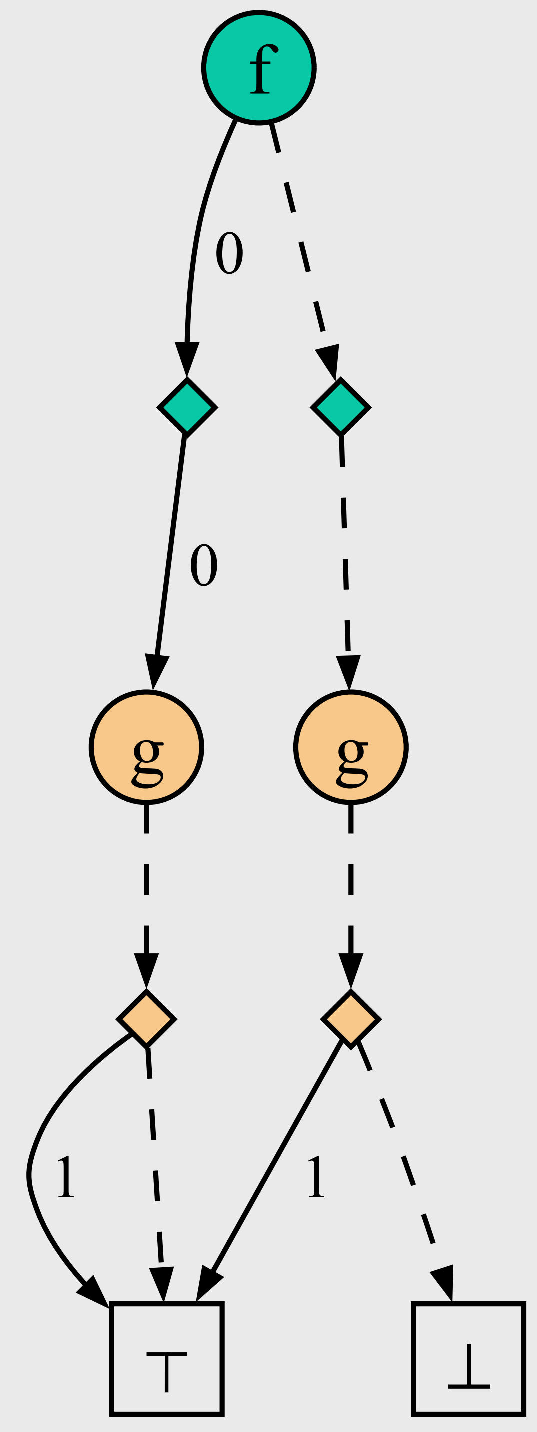
- ...
- ε ∈ S → **Packet** → **Packet Set**
- δ ∈ S → **Packet** → (S * **Packet) Set**

Idea: encode relations using a symbolic representation that is optimized for the common case–i.e., most fields are not changed

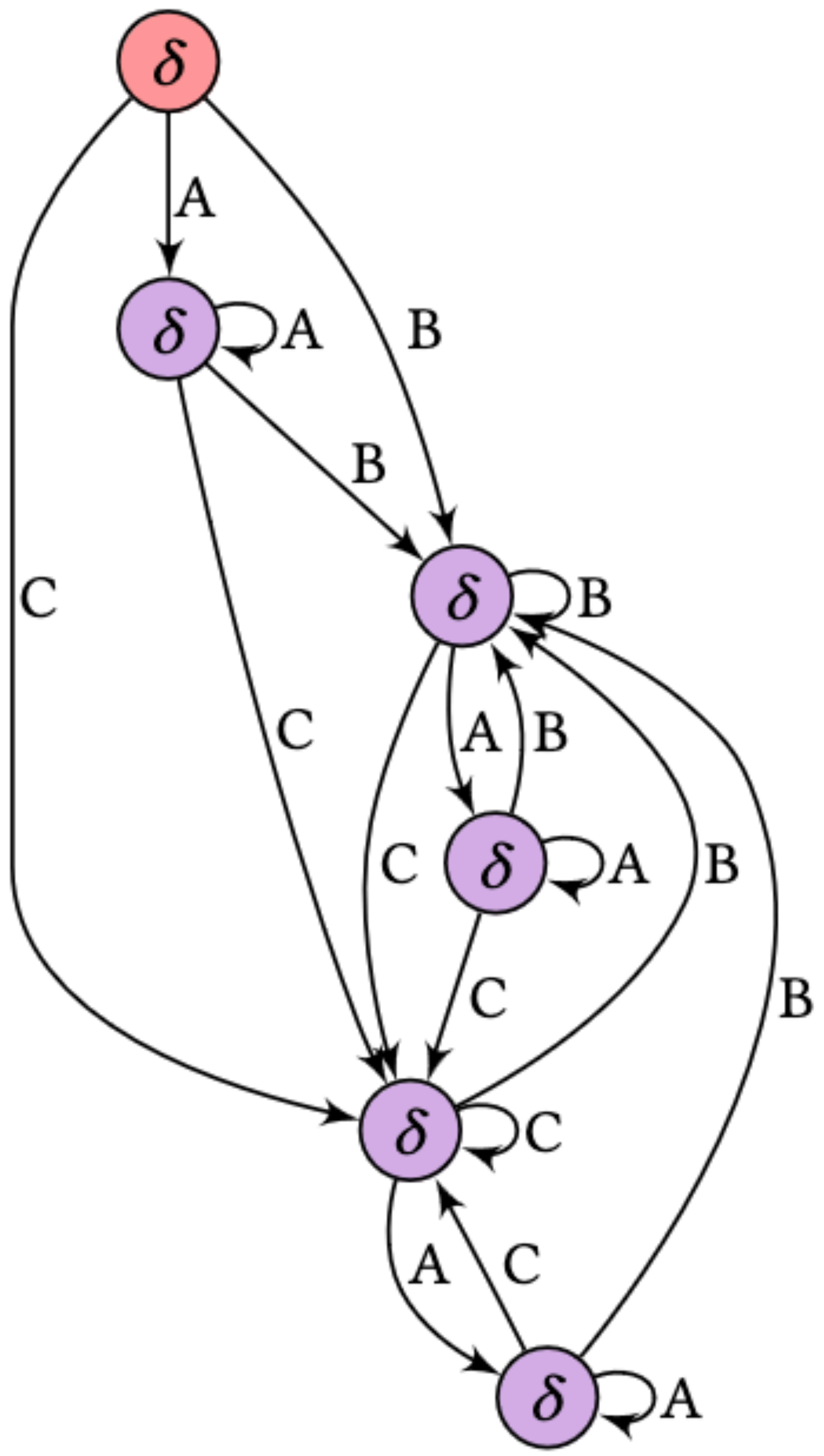
SPPs: two-layer binary decision diagrams where first layer encodes predicates and second layer encodes modifications

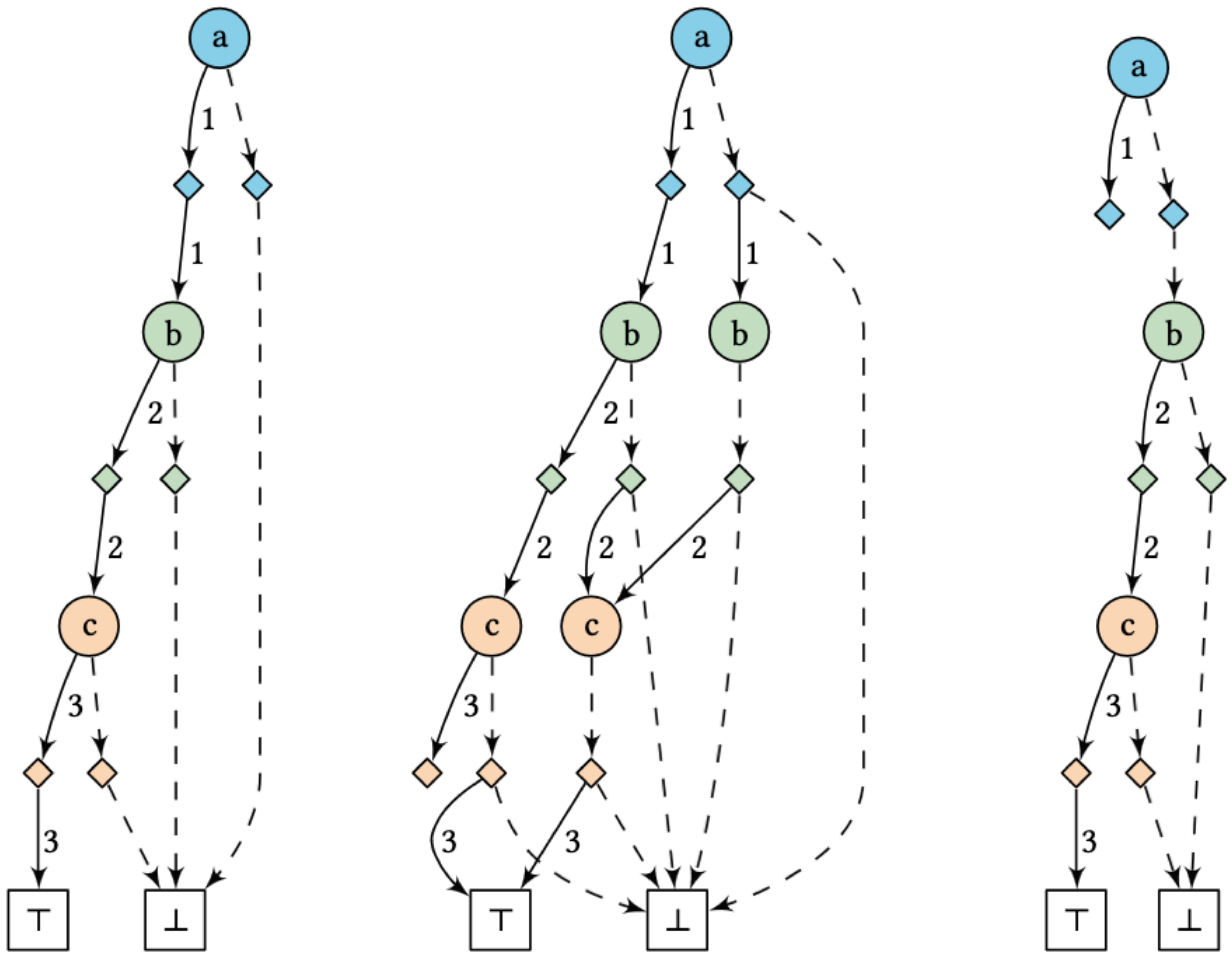
Example

```
f = 0 + g := 1
```



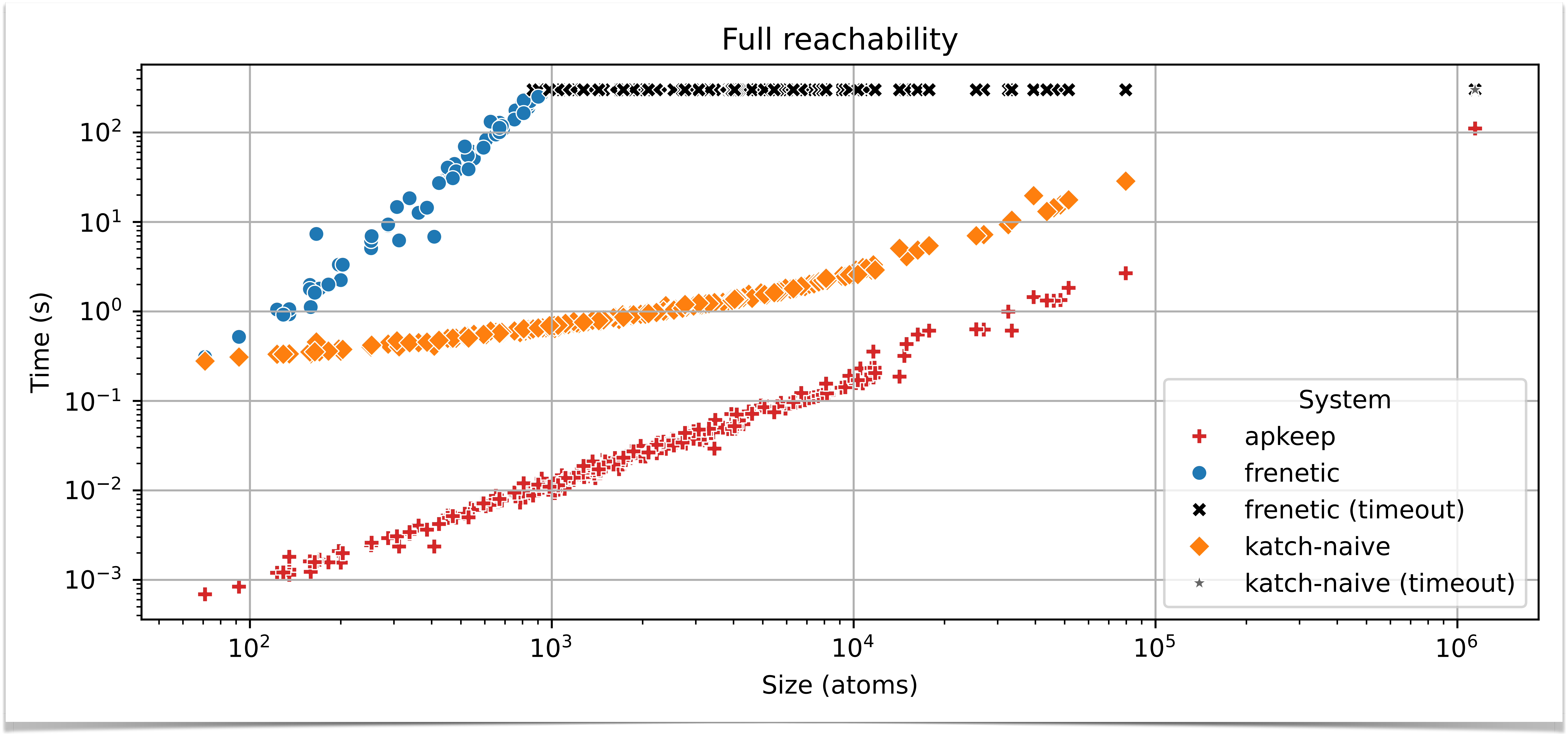
Symbolic automata



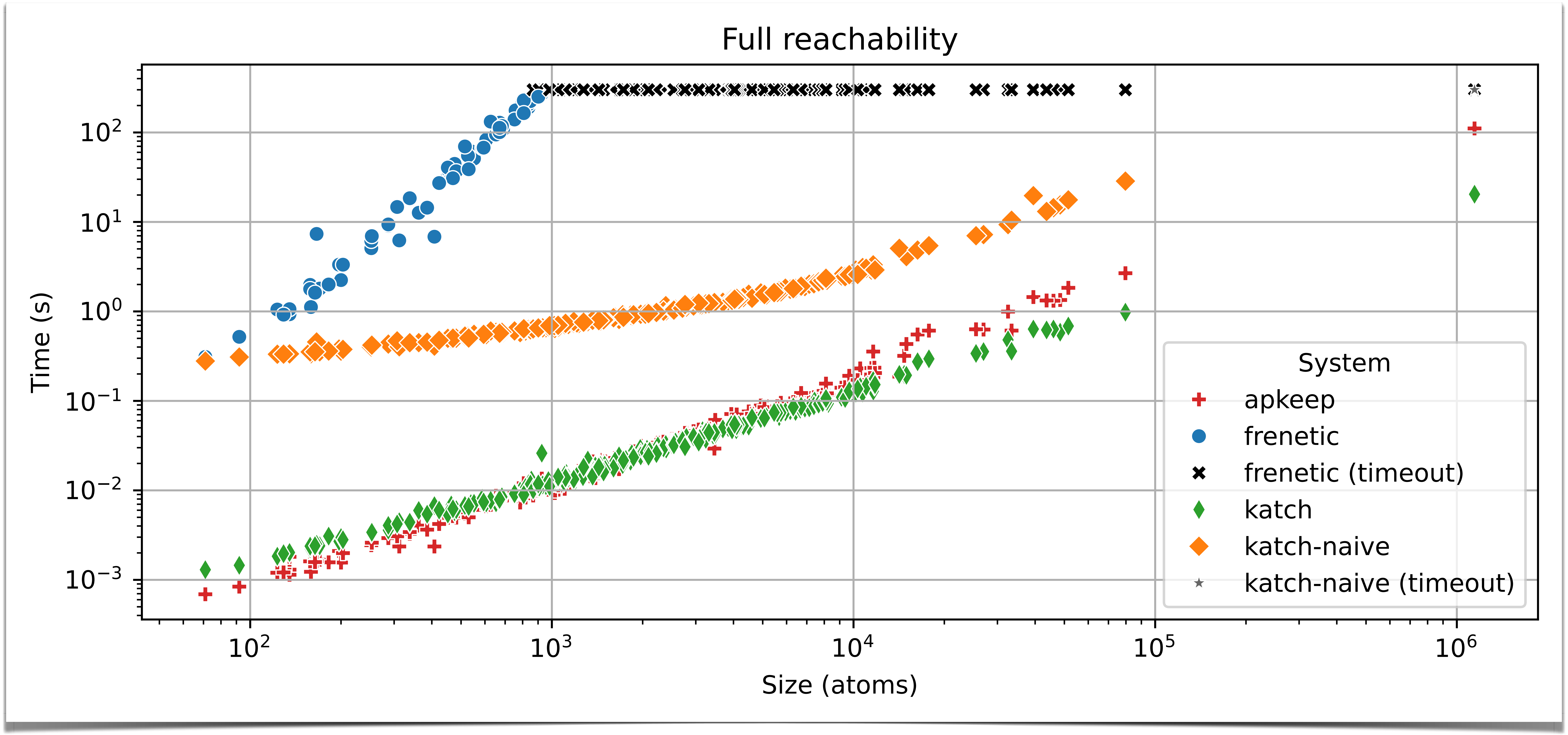


A B C

Evaluation: KATch

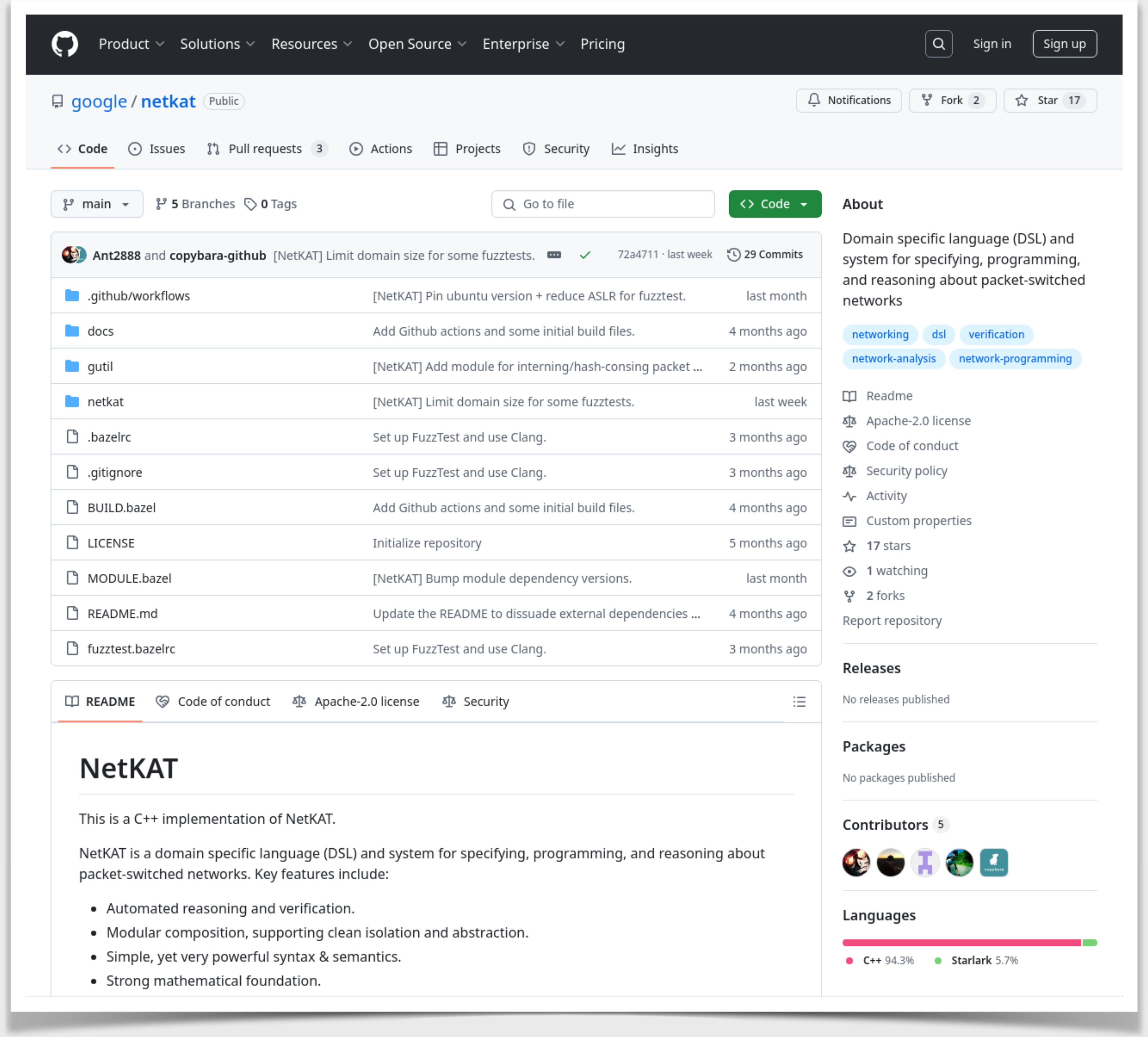


Evaluation: KATch + linear encoding



NetKAT in industry

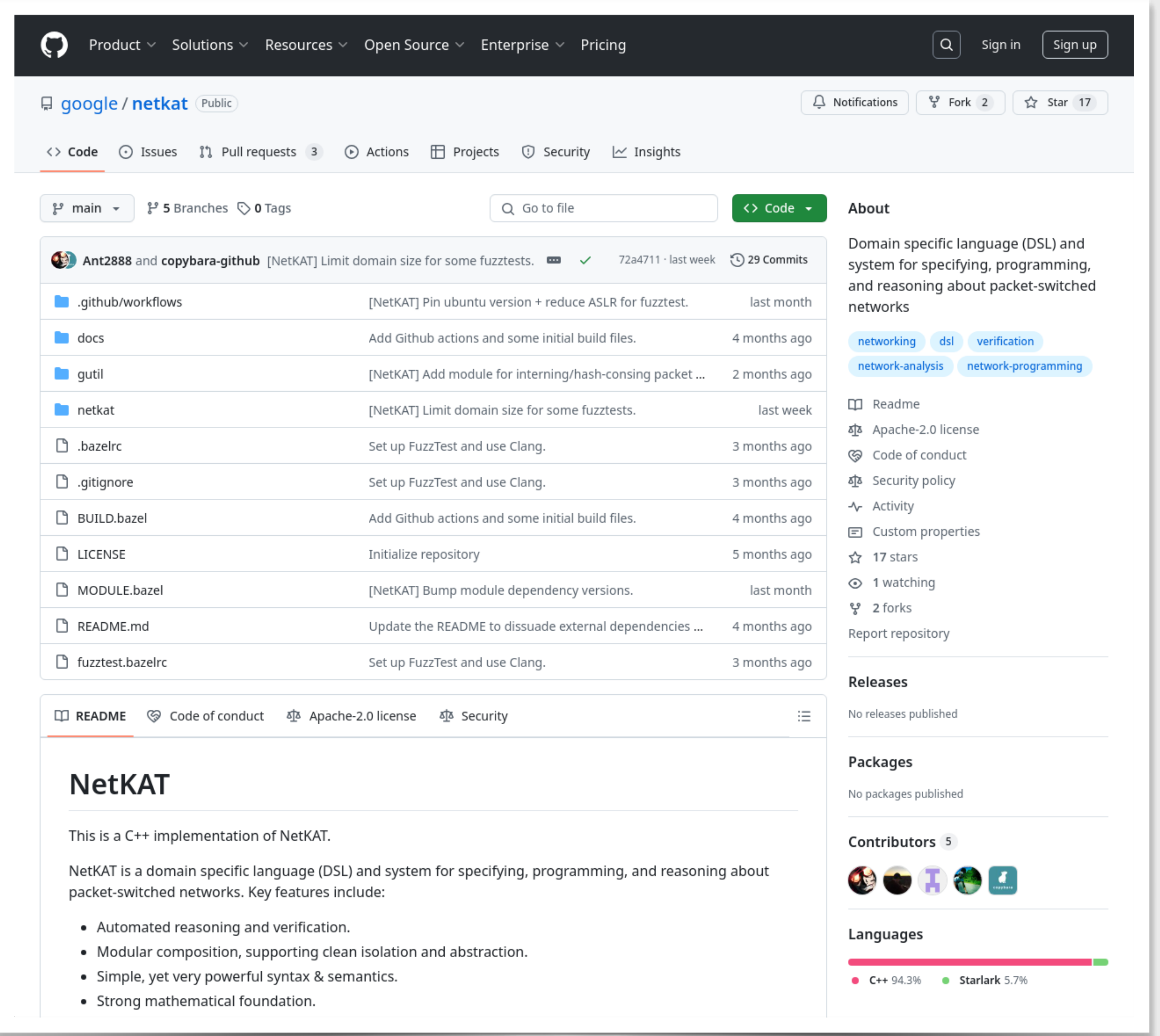
Google



C++ implementation of NetKAT for verifying cloud isolation

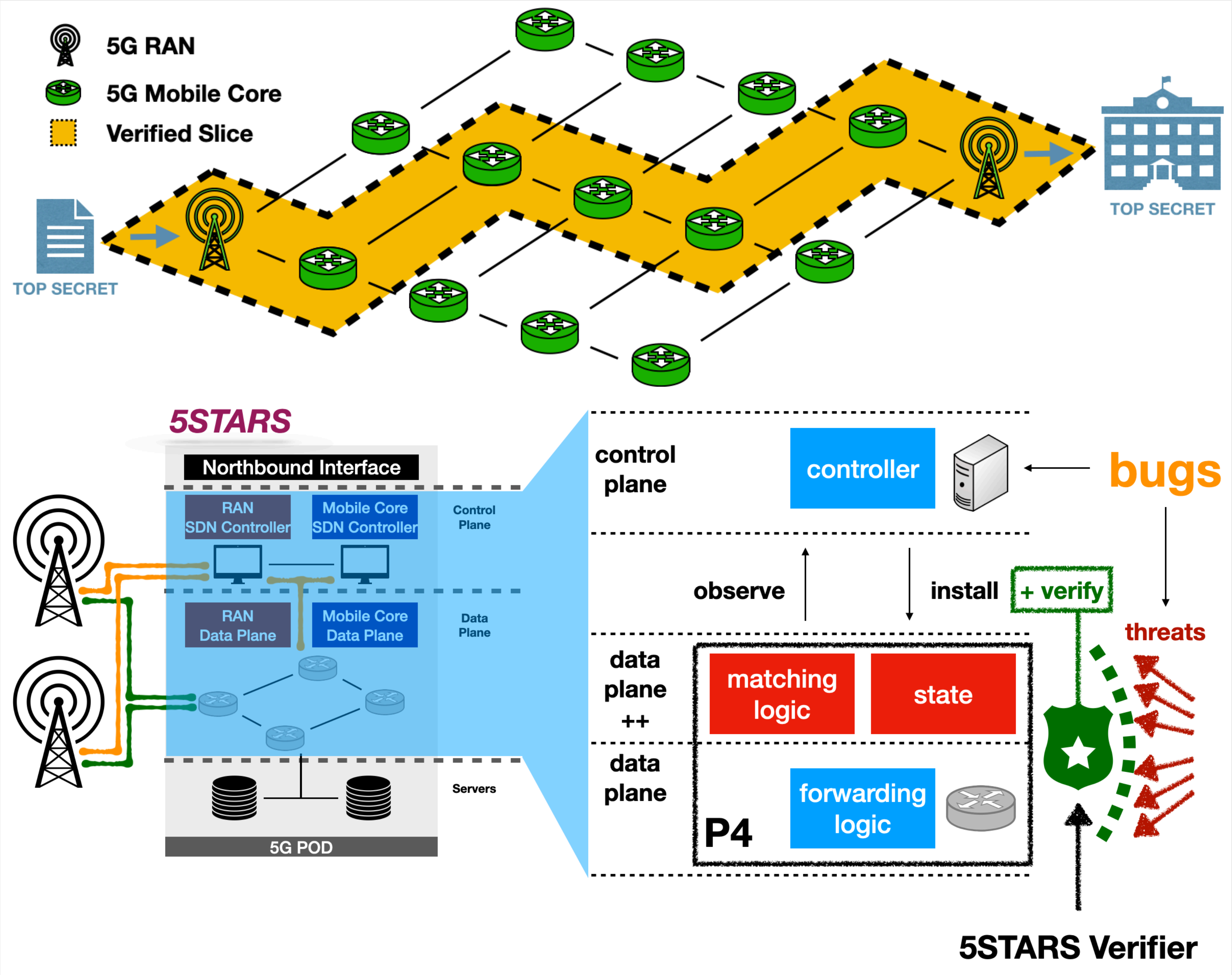
NetKAT in industry

Google



C++ implementation of NetKAT for verifying cloud isolation

galois



Haskell implementation of NetKAT for verifying secure 5G slicing

Lots of KATs

KAT with finite mutable state
LICS '14

NetKAT with temporal look around
PLDI '16

Normal Forms for Network Data Planes
CoNEXT '19

Verifying File Systems using Kleene Algebra
PLDI '19

Higher-order ProbNetKAT
POPL '20

Dynamic NetKAT
FoSSaCS '22

KA modulo theories
PLDI '22 🏆

Verifying Quantum Programs with Kleene Algebra
PLDI '22

BellNetKAT
PLDI '24

Verifying Control Flow Transformations with GKAT
POPL '25

Victory Lap

Course Goals

At the start of the semester, we set out to:

- Understand how to design languages…
- By modeling their semantics mathematically

JavaScript

```
[] + []
{} + []
[] + {}
{} + {}
```

From *Wat:*
https://www.destroyallsoftware.com/talks/wat

Looking Back

CS {4,5}110 Home Resources Schedule Syllabus Ed CMSX

Schedule

Introduction

Date	Topic	Notes	Assignments
August 25	Course Overview	slides	Introductory Survey due 8/28

Mathematical Foundations

Date	Topic	Notes	Assignments
August 27	Semantics	slides notes	
August 29	Induction	slides notes	
September 1	**Labor Day**		
September 3	Lab: Semantics	lab	
September 5	Lab: Induction	lab	A1 due 9/4

Formal Semantics

Date	Topic	Notes	Assignments
September 8	IMP *(Guest: Kozen)*	slides notes	
September 10	IMP Properties *(Guest: Kozen)*	slides notes	
September 12	Lab: IMP	lab	A2 due 9/11
September 15	Denotational Semantics	slides notes	
September 17	Program Equivalence	slides notes	
September 19	Lab: Denotational Semantics	lab	A3 due 9/18
September 22	Axiomatic Semantics	slides notes	

Program Verification

Date	Topic	Notes	Assignments
September 24	Hoare Logic	slides notes	
September 26	Lab: Axiomatic Semantics	lab handout	A4 due 9/25
September 29	Predicate Transformers	slides notes	
October 1	Separation Logic	slides notes	
October 3	Lab: Separation Logic	lab	A5 due 10/2

λ Calculus

Date	Topic	Notes	Assignments
October 6	Lambda Calculus	slides notes	
October 8	More Lambda Calculus	slides notes	
October 10	**Prelim I**		
October 13	**Fall Break**		
October 15	Definitional Translation *(Guest: Myers)*	slides notes	
October 17	Continuations *(Guest: Myers)*	slides notes	
October 20	Fixed-point Combinators	slides notes	
October 22	de Bruijn Notation and Combinators	slides notes	
October 24	Lab: Lambda-Calculus	lab	A6 due 10/23

Type Systems

Date	Topic	Notes	Assignments
October 27	Type Systems	slides notes	
October 29	Advanced Types	slides notes	
October 31	Lab: Type Systems	lab	A7 due 10/30
November 3	Polymorphism	slides notes	
November 5	Making OCaml Safe for Performance Engineering *(Guest: Minsky)*		
November 7	Lab: Polymorphism	lab	A8 due 11/8

Type Theory

Date	Topic	Notes	Assignments
November 10	**Prelim II**		
November 12	Dependent Types and Type Theory *(Guest: Barbone)*	slides	
November 14	Lab:Type Theory *(Guest: Barbone)*	lab	
November 17	Normalization and Logical Relations	slides notes	
November 19	**Foster out of town**		
November 21	Lab:Logical Relations	lab	A9 due 11/20

Advanced Topics

Date	Topic	Notes	Assignments
November 24	Logic Programming	slides notes code	
November 26	**Thanksgiving Break**		
November 28	**Thanksgiving Break**		
December 1	Lenses	slides	
December 3	Program Synthesis	slides	
December 5	Lab: Domain-Specific Languages		

Mathematical Foundations

Main Topics

- Sets
- Relations
- Functions
- Inductive Proof

Induction Principle

Every inductive set has an analogous principle.

To prove $\forall a.\ P(a)$ we must establish several cases.

- Base cases: $P(a)$ holds for each axiom

$$\frac{}{a \in A}$$

- Inductive cases: For each non-axiom inference rule

$$\frac{a_1 \in A \quad \ldots \quad a_n \in A}{a \in A}$$

if $P(a_1)$ and … and $P(a_n)$ then $P(a)$.

Formal Semantics

Main Topics

- Operational
- Denotational
- Axiomatic
- Fixed points

Denotational Semantics for IMP Commands

$$\mathcal{C}[\![\textbf{skip}]\!] = \{(\sigma,\sigma)\}$$

$$\mathcal{C}[\![x := a]\!] = \{(\sigma, \sigma[x \mapsto n]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\}$$

$$\mathcal{C}[\![c_1; c_2]\!] = \\ \{(\sigma,\sigma') \mid \exists\sigma''.\ ((\sigma,\sigma'') \in \mathcal{C}[\![c_1]\!] \wedge (\sigma'',\sigma') \in \mathcal{C}[\![c_2]\!])\}$$

$$\mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] = \\ \{(\sigma,\sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma') \in \mathcal{C}[\![c_1]\!]\} \ \cup \\ \{(\sigma,\sigma') \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!] \wedge (\sigma,\sigma') \in \mathcal{C}[\![c_2]\!]\}$$

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \mathit{fix}(f) \\ \text{where } F(f) = \{(\sigma,\sigma) \mid (\sigma, \textbf{false}) \in \mathcal{B}[\![b]\!]\} \ \cup \\ \{(\sigma,\sigma') \mid (\sigma, \textbf{true}) \in \mathcal{B}[\![b]\!] \wedge \exists\sigma''.\ ((\sigma,\sigma'') \in \mathcal{C}[\![c]\!] \wedge \\ (\sigma'',\sigma') \in f\}$$

Program Verification

Main Topics

- Partial vs. Total Correctness
- Hoare Logic
- Verification Conditions

Weakest Preconditions

$$
\begin{aligned}
wlp(\textbf{skip}, P) &= P \\
wlp(x := a, P) &= P[a/x] \\
wlp((c_1; c_2), P) &= wlp(c_1, wlp(c_2, P)) \\
wlp(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, P) &= (b \implies wlp(c_1, P)) \land \\
&\quad\; (\neg b \implies wlp(c_2, P)) \\
wlp(\textbf{while } b \textbf{ do } c, P) &= \textstyle\bigwedge_i F_i(P)
\end{aligned}
$$

where

$$
\begin{aligned}
F_0(P) &= \textbf{true} \\
F_{i+1}(P) &= (\neg b \implies P) \land (b \implies wlp(c, F_i(P)))
\end{aligned}
$$

λ-Calculus

Main Topics

- Reduction Strategies
- Encodings
- Fixed Points
- Definitional Translation

Laziness

Consider the call-by-name λ-calculus...

Syntax

$$\begin{aligned} e &::= x \\ &\mid e_1\ e_2 \\ &\mid \lambda x.\ e \\ v &::= \lambda x.\ e \end{aligned}$$

Semantics

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{}{(\lambda x.\ e_1)\ e_2 \to e_1\{e_2/x\}}\ \beta$$

Type Systems

Main Topics

- Typing Relations
- Progress
- Preservation
- Polymorphism

Simply-Typed Lambda Calculus

Static Semantics

$$\frac{}{\Gamma \vdash n : \mathbf{int}}\ \textsc{T-Int} \qquad \frac{}{\Gamma \vdash () : \mathbf{unit}}\ \textsc{T-Unit}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}\ \textsc{T-Add}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{T-Var} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'}\ \textsc{T-Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'}\ \textsc{T-App}$$

Type Theory

Main Topics

- Dependent Types
- Normalization
- Logical Relation

Logical Relation

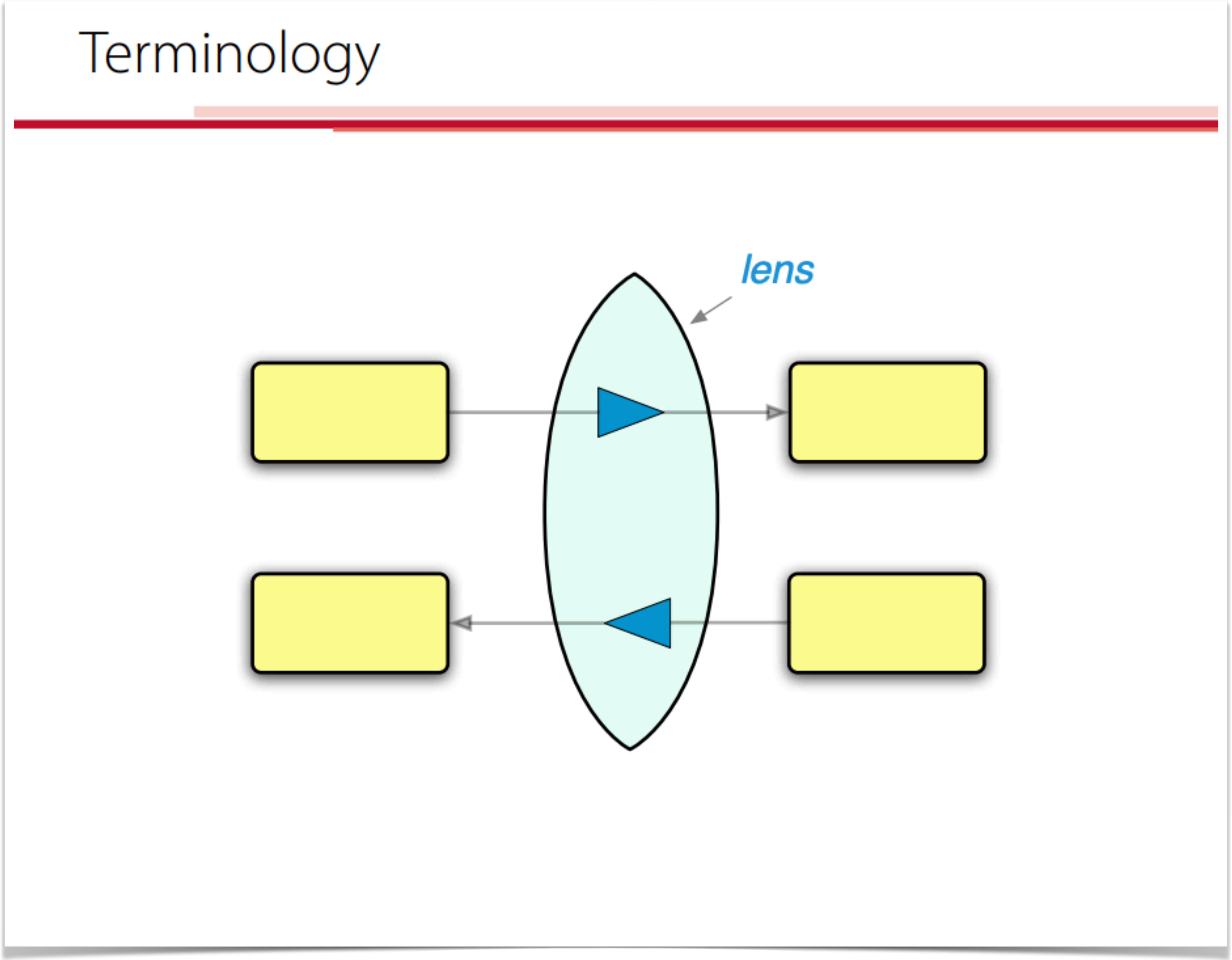
Definition (Logical Relation)

- $R_{\mathbf{unit}}(e)$ iff $\vdash e : \mathbf{unit}$ and e halts.
- $R_{\tau_1 \to \tau_2}(e)$ iff $\vdash e : \tau_1 \to \tau_2$ and e halts, and for every e' such that $R_{\tau_1}(e')$ we have $R_{\tau_2}(e\, e')$.

Advanced Topics

Main Topics

- DSLs
- Logic Programming
- Program Synthesis



Next Steps

Next Steps

Courses: CS 4120 (Compilers), CS 6110 (Advanced PL), CS 6120 (Advanced Compilers), CS 6117 (Category Theory)

Next Steps

Courses: CS 4120 (Compilers), CS 6110 (Advanced PL), CS 6120 (Advanced Compilers), CS 6117 (Category Theory)

Research: BURE, ACSU Research Night, CS 4999

Next Steps

Courses: CS 4120 (Compilers), CS 6110 (Advanced PL), CS 6120 (Advanced Compilers), CS 6117 (Category Theory)

Research: BURE, ACSU Research Night, CS 4999

After Cornell: Compilers, Formal Verification, Grad School

Thank You!


Ithaca
Lausanne
EPFL