Verifying Network Data Planes

Nate Foster
Cornell / Barefoot
Can you help me fix my WiFi?
Network operators use a variety of techniques to keep things running including:

- Generating configurations from high-level policies
- Scraping configurations using command-line tools
- Diagnosing errors with ping and traceroute
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 years¹, 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 protocols²,³,⁴.

While specific information on the DOD protocols is fairly generally available⁵,⁶,⁷, 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 gone 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  
The top level goal for the DARPA Internet Architecture was to develop an effective technique for multiplexed utilization of existing interconnected networks. Some elaboration is appropriate to make clear the meaning of that goal.

The components of the Internet were networks, which were to be interconnected to provide some larger service. The original goal was to connect together the original ARPANET with the ARPA packet radio network⁸, in order to give users on the packet radio network access to the large service machines on the ARPANET. At the time it was assumed that there would be other sorts of networks to interconnect, although the local area network had not yet emerged.

An alternative to interconnecting existing networks would have been to design a unified system which supported a variety of different transmission media, a

Internet Principles

Designed to be robust even when some nodes misbehave or even experience outright failures...

...defers many important issues such as performance, security, accountability, etc.
Modern Challenges

Networks have truly become a critical part of our infrastructure...

...they have grown dramatically in size and complexity...

...and they are becoming unwieldy for operators to manage correctly!
Desired Properties

- Connectivity
- Fault-Tolerance
- Isolation
- Loop Freedom
- Blackhole Freedom
- Service Chaining
- Load Balancing
This Talk

Two (mostly) automated approaches for verifying formal properties of network data planes
This Talk

Two (mostly) automated approaches for verifying formal properties of network data planes

Plan:

• Network-wide properties in NetKAT
• Single-device properties in P4
Network-Wide Verification in NetKAT
Conventional Networking

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

Data plane: forwards packets, enforces access control, monitors flows, etc.
Software-Defined Networking
Model

Packets → Packets
Model

Packets → Packets
Model

Packets → Packets
NetKAT Syntax

\[ \text{pol ::= } \text{false} \]
\[ \text{true} \]
\[ f = n \]
\[ f := n \]
\[ \text{pol}_1 + \text{pol}_2 \]
\[ \text{pol}_1 \cdot \text{pol}_2 \]
\[ !\text{pol} \]
\[ \text{pol}* \]
\[ \text{dup} \]
NetKAT Syntax

### Examples:

- **false**
- **true**
- \( f = n \)
- \( f := n \)
- \( \text{pol}_1 + \text{pol}_2 \)
- \( \text{pol}_1 \cdot \text{pol}_2 \)
- \( !\text{pol} \)
- \( \text{pol}^* \)
- **dup**

**Boolean Predicates**
- +

**Regular Expressions**
- +

**Packet Primitives**

Negation may only be applied to Boolean predicates: 
**true**, **false**, \( f = n \), closed under +, •, and !
Negation may only be applied to Boolean predicates: `true`, `false`, `f = n`, closed under `+`, `•`, and `!`
NetKAT Syntax

pol ::= false | true | f = n | f := n | pol₁ + pol₂ | pol₁ • pol₂ | !pol | pol* | dup

Boolean Predicates

Regular Expressions

Packet Primitives

Negation may only be applied to Boolean predicates:

true, false, f = n, closed under +, •, and !
NetKAT Syntax

**Boolean Predicates**

\[ pol ::= \text{false} \mid \text{true} \]

\[ \text{if } b \text{ then } p_1 \text{ else } p_2 \triangleq (b \cdot p_1) + (!b \cdot p_2) \]

\[ \text{while } b \text{ do } p \triangleq (b \cdot p)^* \cdot !b \]

\[ S \rightarrow S' \triangleq \text{sw}=S \cdot \text{dup} \cdot \text{sw}:=(S' \cdot \text{dup}) \]

**Primitives**

\[ pol \mid \text{dup} \]

Negation may only be applied to Boolean predicates: \[ true, false, f = n, \text{ closed under } +, \cdot, \text{ and } ! \]
Semantics

pol ::= 
| false 
| true 
| field = val 
| field ::= val 
| pol₁ + pol₂ 
| pol₁ • pol₂ 
| !pol 
| pol* 
| dup
Semantics

Local: input-output behavior of switches

\[ \Phi(\text{pol}) \in \text{Packets} \rightarrow \text{Packets} \]
\textbf{Semantics}

\begin{align*}
\text{pol} & ::= \\
& \mid \text{false} \\
& \mid \text{true} \\
& \mid \text{field} = \text{val} \\
& \mid \text{field} := \text{val} \\
& \mid \text{pol}_1 + \text{pol}_2 \\
& \mid \text{pol}_1 \cdot \text{pol}_2 \\
& \mid \neg \text{pol} \\
& \mid \text{pol}^* \\
& \mid \text{dup}
\end{align*}

\textbf{Local:} input-output behavior of switches

$\left[ \Phi(\text{pol}) \right] \in \text{Packets} \rightarrow \text{Packets}$

\textbf{Global:} network-wide paths

$\left[ \text{pol} \right] \in \text{Histories} \rightarrow \text{Histories}$
Encoding Tables and Links

Switch routing tables and network topologies can be represented in NetKAT using straightforward encodings

<table>
<thead>
<tr>
<th>Match</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstport=22</td>
<td>Drop</td>
</tr>
<tr>
<td>srcip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>*</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

if dstport=22 then false
elsif srcip=10.0.0.1 then port := 1
else port := 2
Switch routing tables and network topologies can be represented in NetKAT using straightforward encodings.

<table>
<thead>
<tr>
<th>Match</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstport=22</td>
<td>Drop</td>
</tr>
<tr>
<td>srcip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>*</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

if dstport=22 then false
elsif srcip=10.0.0.1 then port := 1
else port := 2
Encoding Networks

A network can be encoded in NetKAT by interleaving steps of processing by switches and topology.

\[(\text{pol} \cdot \text{topo})^*\]
Decision Procedure

Can check whether programs are equivalent automatically!

**Theoretical Insight:** NetKAT programs ↔ NetKAT automata
Decision Procedure

Can check whether programs are equivalent automatically!

**Theoretical Insight:** NetKAT programs ↔ NetKAT automata
Decision Procedure

Can check whether programs are equivalent automatically!

**Theoretical Insight:** NetKAT programs ↔ NetKAT automata

Algorithm checks whether automata are bisimilar
From Programs to Automata

Derivatives

\[ D_c L = \{ w \mid c \cdot w \in L \} \]

Automata

\[ \begin{align*}
&\text{p} \\
&\xrightarrow{\text{pk}_1} \text{D}_{\text{pk}_1}(p) \\
&\xrightarrow{\text{pk}_2} \text{D}_{\text{pk}_2}(p) \\
&\xrightarrow{\text{pk}_3} \text{D}_{\text{pk}_3}(p)
\end{align*} \]
From Programs to Automata

Derivatives

\[ D_c L = \{ w \mid c \cdot w \in L \} \]

Automata

Can be defined syntactically via a simple recursive definition
From Programs to Automata

**Derivatives**

\[ D_c L = \{ w \mid c \cdot w \in L \} \]

**Automata**

Can be defined syntactically via a simple recursive definition

Terminates since every program has a finite number of distinct derivatives
NetKAT Automata

NetKAT automata recognize the histories generated by packets as they traverse the network:

\[ \text{pkt}_{\text{in}} \cdot \text{pkt}_1 \cdot \text{dup} \cdot \ldots \cdot \text{dup} \cdot \text{pkt}_n \cdot \text{dup} \cdot \text{put}_{\text{out}} \]

Similar to standard automata, but generalized to packets
NetKAT Automata

NetKAT automata recognize the histories generated by packets as they traverse the network:

\[ \text{pkt}_{\text{in}} \cdot \text{pkt}_1 \cdot \text{dup} \cdot \ldots \cdot \text{dup} \cdot \text{pkt}_n \cdot \text{dup} \cdot \text{put}_{\text{out}} \]

Similar to standard automata, but generalized to packets

A *NetKAT automaton* $M = (S, s_0, \varepsilon, \delta)$ is a tuple where:

- $S$ is a finite set of states,
- $s_0 \in S$ is the start state,
- $\varepsilon \in S \rightarrow \text{Packet} \rightarrow \text{Packet Set}$ is the “acceptance” function,
- $\delta \in S \rightarrow \text{Packet} \rightarrow (\text{State} \times \text{Packet}) \text{ Set}$ is the “transition” function.
Syntactic Derivatives

\[
E(\text{false}) = \text{false} \\
E(\text{true}) = \text{true} \\
E(f = n) = f = n \\
E(f := n) = f := n \\
E(!\text{pol}) = !\text{pol} \\
E(\text{dup}^l) = \text{false} \\
E(\text{pol}_1 + \text{pol}_2) = E(\text{pol}_1) + E(\text{pol}_2) \\
E(\text{pol}_1 \cdot \text{pol}_2) = E(\text{pol}_1) \cdot E(\text{pol}_2) \\
E(\text{pol}^*) = E(\text{pol})^* \\
\]

\[
D(\text{false}) = \{} \\
D(\text{true}) = \{} \\
D(f = n) = \{} \\
D(f := n) = \{} \\
D(!\text{pol}) = \{} \\
D(\text{dup}^l) = \{(\text{true}, l, \text{true})\} \\
D(\text{pol}_1 + \text{pol}_2) = D(\text{pol}_1) + D(\text{pol}_2) \\
D(\text{pol}_1 \cdot \text{pol}_2) = D(\text{pol}_1) \cdot \text{pol}_2 + E(\text{pol}_1) \cdot D(\text{pol}_2) \\
D(\text{pol}^*) = E(\text{pol})^* \cdot D(\text{pol}) \cdot \text{pol}^* \\
\]

NetKAT Automaton

- \(S\) is the set of \text{dup}s, plus a fresh start state
- \(\varepsilon \mid \text{pkt} = \{ \text{pkt}' \mid \text{pkt}' \in [E(k)] \text{pkt} \}\)
- \(\delta \mid \text{pkt} = \{ (\text{pkt}', l') \mid (d, l', k) \in [D(k)] \land \text{pkt}' \in [d] \text{pkt} \}\)

where \(k\) is the "continuation" of \text{dup}^l
### Syntactic Derivatives

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E(\text{false}))</td>
<td>false</td>
</tr>
<tr>
<td>(E(\text{true}))</td>
<td>true</td>
</tr>
<tr>
<td>(E(f = n))</td>
<td>(f = n)</td>
</tr>
<tr>
<td>(E(f := n))</td>
<td>(f := n)</td>
</tr>
<tr>
<td>(E(\text{!pol}))</td>
<td>(\text{!pol})</td>
</tr>
<tr>
<td>(E(\text{dup}^l))</td>
<td>false</td>
</tr>
<tr>
<td>(E(\text{pol}_1 + \text{pol}_2))</td>
<td>(E(\text{pol}_1) + E(\text{pol}_2))</td>
</tr>
<tr>
<td>(E(\text{pol}_1 \cdot \text{pol}_2))</td>
<td>(E(\text{pol}_1) \cdot E(\text{pol}_2))</td>
</tr>
<tr>
<td>(E(\text{pol}^*))</td>
<td>(E(\text{pol})^*)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D(\text{false}))</td>
<td>{}</td>
</tr>
<tr>
<td>(D(\text{true}))</td>
<td>{}</td>
</tr>
<tr>
<td>(D(f = n))</td>
<td>{}</td>
</tr>
<tr>
<td>(D(f := n))</td>
<td>{}</td>
</tr>
<tr>
<td>(D(\text{!pol}))</td>
<td>{}</td>
</tr>
<tr>
<td>(D(\text{dup}^l))</td>
<td>{(\text{true}, l, \text{true})}</td>
</tr>
<tr>
<td>(D(\text{pol}_1 + \text{pol}_2))</td>
<td>(D(\text{pol}_1) + D(\text{pol}_2))</td>
</tr>
<tr>
<td>(D(\text{pol}_1 \cdot \text{pol}_2))</td>
<td>(D(\text{pol}_1) \cdot \text{pol}_2 + E(\text{pol}_1) \cdot D(\text{pol}_2))</td>
</tr>
<tr>
<td>(D(\text{pol}^*))</td>
<td>(E(\text{pol})^* \cdot D(\text{pol}) \cdot \text{pol}^*)</td>
</tr>
</tbody>
</table>

**NetKAT Automaton**

- \(S\) is the set of \textit{dup}s, plus a fresh start state
- \(\varepsilon \mid \text{pkt} = \{ \text{pkt}' \mid \text{pkt}' \in \llbracket E(k_i) \rrbracket \text{pkt} \} \)
- \(\delta \mid \text{pkt} = \{ (\text{pkt}', l') \mid (d, l', k) \in \llbracket D(k_i) \rrbracket \land \text{ pkt}' \in \llbracket d \rrbracket \text{pkt} \} \)

where \(k_i\) is the "continuation" of \textit{dup}^l
Application: Traffic Isolation

We’d like to be able to answer questions like:

“Does the network isolate A and B?"

Can reduce this question to equivalence!

\[ A \cdot (pol \cdot topo)^* \cdot B \equiv false \]
Application: Loop Freedom

Can exploit automata representations to efficiently check whether a network is free of forwarding loops...

Intuitively,

\[ \forall \alpha. (p \cdot t)^+ \cdot \alpha \cdot (p \cdot t)^+ \cdot \alpha \equiv \text{false} \]

Formally:

- \( \forall \text{pkt, pkt'}. \text{pkt'} \in \llbracket E(\Phi(\text{in} \cdot (p \cdot t)^+)) \rrbracket \text{pkt} \)
- Check whether \( \text{pkt'} \in \llbracket E(\Phi(p \cdot t)^+) \rrbracket \text{pkt'} \)

Can be made fast using sparse matrix representation
Single-Device Verification in P4
P4/RMT [SIGCOMM '13]

Diagram showing the traffic management components:
- Parser
- Ingress
- Traffic Manager
- Egress
- Deparser
P4/RMT [SIGCOMM '13]

Parser  Ingress  Traffic Manager  Egress  Deparser
P4/RMT [SIGCOMM '13]
P4/RMT [SIGCOMM '13]

Parser  Ingress  Traffic Manager  Egress  Deparser
P4/RMT [SIGCOMM '13]

- Parser
- Ingress
- Traffic Manager
- Egress
- Deparser
P4/RMT [SIGCOMM '13]

Parser  Ingress  Traffic Manager  Egress  Deparser
P4/RMT [SIGCOMM '13]

Parser  Ingress  Traffic Manager  Egress  Deparser
P4/RMT [SIGCOMM '13]

Traffic Manager

Parser  Ingress  Egress  Deparser
• Slogan: "constant work in constant time"
  - No pointers or complex data types
  - Bounded state
  - No loops

• Key construct is a match-action table

```p4
action learn() {
    generate_digest(RECV, learn_digest);
}

table smac {
    reads {
        ethernet.srcAddr : exact;
    }
    actions {
        learn; nop;
    }
    default_action: nop;
}
```

<table>
<thead>
<tr>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00:00:00:00:01</td>
<td>learn</td>
</tr>
<tr>
<td>00:00:00:00:00:02</td>
<td>learn</td>
</tr>
<tr>
<td>*</td>
<td>nop</td>
</tr>
</tbody>
</table>
**Example: Ethernet Switch**

```plaintext
header_type ethernet_t {
  fields {
    dstAddr : 48;
    srcAddr : 48;
    etherType : 16;
  }
}

header_type intrinsic_metadata_t {
  fields {
    mcast_grp : 4;
    egress_rid : 4;
    mcast_hash : 16;
    lf_field_list : 32;
  }
}

header ethernet_t ethernet;
metadata intrinsic_metadata_t intrinsic_metadata;
parser start {
  return parse_ethernet;
}
parser parse_ethernet {
  extract(ethernet);
  return ingress;
}

field_list mac_learn_digest {
  ethernet.srcAddr;
  standard_metadata.ingress_port;
}

action mac_learn() {
  generate_digest(MAC_LEARN_RECEIVER, mac_learn_digest);
}

action forward(port) {
  modify_field(standard_metadata.egress_spec, port);
}

action broadcast() {
  modify_field(intrinsic_metadata.mcast_grp, 1);
}

table smac {
  reads {
    ethernet.srcAddr : exact;
  }
  actions {
    mac_learn;
    nop;
  }
  size : 512;
}

table dmac {
  reads {
    ethernet.dstAddr : exact;
  }
  actions {
    forward;
    broadcast;
  }
  size : 512;
}

table mcast_src_pruning {
  reads {
    standard_metadata.instance_type : exact;
  }
  actions {
    nop;
    drop;
  }
  size : 1;
}

collection ingress {
  apply(smac);
  apply(dmac);
}

collection egress {
  (if(standard_metadata.ingress_port ==
       standard_metadata.egress_port) {
    apply(mcast_src_pruning);
  })
}
```
Example: Ethernet Switch

Types

header_type ethernet_t {
  fields {
    dstAddr : 48;
    srcAddr : 48;
    etherType : 16;
  }
}

header_type intrinsic_metadata_t {
  fields {
    mcast_grp : 4;
    egress_rid : 4;
    mcast_hash : 16;
    lf_field_list : 32;
  }
}

header ethernet_t ethernet;
metadata intrinsic_metadata_t intrinsic_metadata;

Parsers

parser start {
  return parse_ethernet;
}

parser parse_ethernet {
  extract(ethernet);
  return ingress;
}

field_list mac_learn_digest {
  ethernet.srcAddr;
  standard_metadata.ingress_port;
}

Actions

action mac_learn() {
  generate_digest(MAC_LEARN_RECEIVER, mac_learn_digest);
}

action forward(port) {
  modify_field(standard_metadata.egress_spec, port);
}

action broadcast() {
  modify_field(intrinsic_metadata.mcast_grp, 1);
}

Tables

table smac {
  reads {
    ethernet.srcAddr : exact;
  }
  actions {
    mac_learn;
    nop;
  }
  size : 512;
}

table dmac {
  reads {
    ethernet.dstAddr : exact;
  }
  actions {
    forward;
    broadcast;
  }
  size : 512;
}

table mcast_src_pruning {
  reads {
    standard_metadata.instance_type : exact;
  }
  actions {
    nop;
    drop;
  }
  size : 1;
}

Controls

cancel ingress {
  apply(smac);
  apply(dmac);
}

cancel egress {
  (if(standard_metadata.ingress_port ==
    standard_metadata.egress_port)
    apply(mcast_src_pruning));
}
Data Plane Errors

Making switches more programmable increases flexibility...
...but also opens up possibilities for new kinds of errors:

- Reading/writing invalid headers
- Unhandled exceptions
- Incorrect use of packet metadata
- Malformed parsers/deparsers
- Unintended control flows
An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming; proofs of programs, formal language definitions, programming language design, module-independent programming, program documentation

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

\[ x = x + 0 \]
\[ 0 < r \Rightarrow r + p \times q = (r - y) + y \times (1 + q) \]

The proof of the second of these is:

\[ A5 \quad (r - y) + y \times (1 + q) = (r - y) + (y \times 1 + y \times q) \]
\[ A9 \quad (r - y) + (y \times q) \]
\[ A3 \quad (r - y) + y \times q \]
\[ A6 \quad r = r + y \times q \quad \text{provided} \quad y \neq r \]

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to nonnegative numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which \( 0, 1, 2, 3 \) and \( 4 \) are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

\[ A10 \quad \exists x \forall y \quad (y < x) \]
where all finite arithmetic satisfies:

\[ A10 \quad \forall x \quad (x < \text{max}) \]

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of \( \text{max} + 1 \):

\[ A11a \quad \neg \exists x \quad (x = \text{max} + 1) \quad \text{(strict interpretation)} \]
\[ A11b \quad \text{max} + 1 = \text{max} + 1 \quad \text{(firm boundary)} \]
\[ A11c \quad \text{max} + 1 = 0 \quad \text{(modulo arithmetic)} \]

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,
Example: Header Validity

Reading or writing an invalid header yields an undefined value (!) but packet headers have complex dependencies
Example: Header Validity

Reading or writing an invalid header yields an undefined value (!) but packet headers have complex dependencies.

Worse, P4's type system does not offer good constructs for precisely documenting which headers are valid!
Demo
Example: Correct Decapsulation

```plaintext
action remove_vlan_single_tagged() {
    modify_field(ethernet.etherType, vlan_tag_[0].etherType);
    remove_header(vlan_tag_[0]);
}

action remove_vlan_double_tagged() {
    modify_field(ethernet.etherType, vlan_tag_[1].etherType);
    remove_header(vlan_tag_[0]);
    remove_header(vlan_tag_[1]);
}

table vlan_decap {
    reads {
        vlan_tag_[0]: valid;
        vlan_tag_[1]: valid;
    }
    actions {
        nop;
        remove_vlan_single_tagged;
        remove_vlan_double_tagged;
    }
}

@pragma assert not(valid(vlan[0]) or valid(vlan[1]))
```
A P4 program is really only half of a program

The match-action tables are populated by the control plane which is unknown!

Need a way to document assumptions about the control plane

<table>
<thead>
<tr>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00:00:00:00:01</td>
<td>learn</td>
</tr>
<tr>
<td>00:00:00:00:00:02</td>
<td>learn</td>
</tr>
<tr>
<td>*</td>
<td>nop</td>
</tr>
</tbody>
</table>
Ghost State

Idea: instrument program with ghost state

```c
header_type _p4v_zombie_t {
    fields {
        ...
        decap_order : 2;
        decap_hit : 1;
        decap_action : 2;
        decap_reads_0 : 1;
    }
}
action decap_nop() {
    modify_field(_p4v_zombie.decap_hit, 1);
    modify_field(_p4v_zombie.decap_reads_0, valid(vlan[0]));
    modify_field(_p4v_zombie.decap_action, 1);
}
```

Can then formulate control-plane assumptions

```c
@pragma assume hit(decap)
```
Wrapping up...
Conclusions

The intersection between formal methods and networking has gotten very interesting in recent years.

The emergence of SDN/P4 offers a unique opportunity to shape how networks are built and operated for decades to come.

Many challenging problems remain:
- Stateful verification
- Quantitative properties
- Usability by non-experts
Thank you!

http://frenetic-lang.org/

- Carolyn Anderson (UMass)
- Shrutarshi Basu (Cornell)
- Spiros Eliopoulos (Inhabited Type)
- Arjun Guha (UMass)
- Bill Callahan (Yale)
- Jean-Baptiste Jeannin (Samsung Labs)
- Dexter Kozen (Cornell)
- Praveen Kumar (Cornell)
- JK Lee (Barefoot Networks)
- Matthew Milano (Cornell)
- Mark Reitblatt (Facebook)
- Cole Schlesinger (Barefoot Networks)
- Robert Soulé (USI / Barefoot Networks)
- Alexandra Silva (UCL)
- Steffen Smolka (Cornell)
- Laure Thompson (Cornell)
- David Walker (Princeton)
- Han Wang (Barefoot Networks)
Join Us!
Backup Slides
Network Updates

How can we transition between global states?

Initial State

Target State
How can we transition between global states?
Network Updates

How can we transition between global states?

Initial State

Target State

Problem: naive updates can break important invariants!
Consistent Updates [SIGCOMM '12]

**Consistency Guarantee:**
every packet (or flow) will be processed by a single version of the network-wide configuration

**Implementations:**
- Two-Phase Update
- One-Touch Update
- Order Update