Specifying and Verifying Network Behavior with NetKAT

Nate Foster
Cornell University

ExCAPE Webinar
Collaborators

- Spiros Eliopoulos (Cornell → Inhabited Type)
- Arjun Guha (UMass Amherst)
- Dexter Kozen (Cornell)
- Jean-Baptiste Jeannin (Cornell → Samsung)
- Konstantinos Mamouras (Cornell → Penn)
- Matthew Milano (Cornell)
- Mark Reitblatt (Cornell → Facebook)
- Cole Schlesinger (Princeton → Samsung)
- Alexandra Silva (University College London)
- Steffen Smolka (Cornell)
- Laure Thompson (Cornell)
- David Walker (Princeton)
This Talk

Design and implementation of a high-level language for programming networks
This Talk

Design and implementation of a high-level language for programming networks

Outline:

• Software-Defined Networking
• NetKAT Design
• Formal Reasoning
• Experience
Software-Defined Networking
Software-Defined Networking

- Controller
- Ox Controller Platform or POX, Beacon, Floodlight, etc.
- OpenFlow API
- OpenFlow Switch
- OpenFlow-compatible switches: Pica8, Dell, NEC, HP, and many others.
Software-Defined Networking

Your Program goes here!

Ox Controller Platform
or POX, Beacon, Floodlight, etc.

OpenFlow API

OpenFlow-compatible switches
Pica8, Dell, NEC, HP, and many others
SDN Switch

General-purpose packet-processing device that can be used to implement switches, routers, firewalls, etc.
SDN Switch

General-purpose packet-processing device that can be used to implement switches, routers, firewalls, etc.

<table>
<thead>
<tr>
<th>Match</th>
<th>Actions</th>
<th>Counters</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.1</td>
<td>Drop</td>
<td>(73,2458)</td>
</tr>
<tr>
<td>10.0.0.2</td>
<td>Forward 2</td>
<td>(16,846)</td>
</tr>
<tr>
<td>10.0.0.3</td>
<td>Forward 3</td>
<td>(23,5729)</td>
</tr>
<tr>
<td>*</td>
<td>Controller</td>
<td>(5,472)</td>
</tr>
</tbody>
</table>

Key data structure is a forwarding table containing a prioritized list of match-action rules and counters.
SDN Controller

Switch to controller:
- `switch_connected`
- `switch_disconnected`
- `port_status`
- `packet_in`
- `stats_reply`

Controller to switch:
- `flow_mod`
- `packet_out`
- `stats_request`
Repeater in Ox

open OxPlatform
open OpenFlow0x01_Core

module MyApplication = struct

  include OxStart.DefaultTutorialHandlers

  let switch_connected (sw : switchId) : unit =
    send_flow_mod sw 0l (del_flow 0 any [])
    send_flow_mod sw 0l (add_flow 0 any [Flood])

  let packet_in (sw : switchId) (xid : xid) (pk : packetIn) : unit =
    send_packet_out sw 0l
    { output_payload = pk.input_payload;
      port_id = None;
      apply_actions = [Flood] }
  end

module Controller = OxStart.Make (MyApplication)
Repeater in Ox

open OxPlatform
open OpenFlow0x01_Core

module MyApplication = struct

  include OxStart.DefaultTutorialHandlers

  let switch_connected (sw : switchId) : unit =
    send_flow_mod sw 0l (del_flow 0 any [])
    send_flow_mod sw 0l (add_flow 0 any [Flood])

  let packet_in (sw : switchId) (xid : xid) (pk : packetIn) : unit =
    send_packet_out sw 0l
    { output_payload = pk.input_payload;
      port_id = None;
      apply_actions = [Flood] }

end

module Controller = OxStart.Make (MyApplication)
open OxPlatform
open OpenFlow0x01_Core

module MyApplication = struct

include OxStart.DefaultTutorialHandlers

let switch_connected (sw : switchId) : unit =
  send_flow_mod sw 0l (del_flow θ any [])
  send_flow_mod sw 0l (add_flow θ any [Flood])

let packet_in (sw : switchId) (xid : xid) (pk : packetIn) : unit =
  send_packet_out sw 0l
  { output_payload = pk.input_payload;
    port_id = None;
    apply_actions = [Flood] } end

module Controller = OxStart.Make (MyApplication)
### Route

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstip=10.0.0.1</td>
<td>Fwd 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Fwd 2</td>
</tr>
</tbody>
</table>

### Monitor

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
<tr>
<td>Pattern</td>
<td>Actions</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.1</td>
<td>Fwd 1, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.2</td>
<td>Fwd 2, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
<tr>
<td>dstip=10.0.0.1</td>
<td>Fwd 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Fwd 2</td>
</tr>
<tr>
<td>Pattern</td>
<td>Actions</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.1</td>
<td>Fwd 1, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.2</td>
<td>Fwd 2, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
<tr>
<td>dstip=10.0.0.1</td>
<td>Fwd 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Fwd 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpdst = 22</td>
<td>Drop</td>
</tr>
<tr>
<td>*</td>
<td>Fwd ?</td>
</tr>
<tr>
<td>Pattern</td>
<td>Actions</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td><code>srcip=1.2.3.4, dstip=10.0.0.1</code></td>
<td>Fwd 1, Count</td>
</tr>
<tr>
<td><code>srcip=1.2.3.4, dstip=10.0.0.2</code></td>
<td>Fwd 2, Count</td>
</tr>
<tr>
<td><code>srcip=1.2.3.4</code></td>
<td>Count</td>
</tr>
<tr>
<td><code>dstip=10.0.0.1</code></td>
<td>Fwd 1</td>
</tr>
<tr>
<td><code>dstip=10.0.0.2</code></td>
<td>Fwd 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tcpdst = 22</code></td>
<td>Drop</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Fwd ?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>srcip=1.2.3.4, tcpdst = 22</code></td>
<td>Count, Drop</td>
</tr>
<tr>
<td><code>srcip=1.2.3.4, dstip=10.0.0.1</code></td>
<td>Fwd 1, Count</td>
</tr>
<tr>
<td><code>srcip=1.2.3.4, dstip=10.0.0.2</code></td>
<td>Fwd 2, Count</td>
</tr>
<tr>
<td><code>srcip=1.2.3.4</code></td>
<td>Count</td>
</tr>
<tr>
<td><code>tcpdst = 22</code></td>
<td>Drop</td>
</tr>
<tr>
<td><code>dstip=10.0.0.1</code></td>
<td>Fwd 1</td>
</tr>
<tr>
<td><code>dstip=10.0.0.2</code></td>
<td>Fwd 2</td>
</tr>
</tbody>
</table>
A machine model describes behavior in terms of concepts like pipelines of hardware lookup tables.
A machine model describes behavior in terms of concepts like pipelines of hardware lookup tables.

A programming model describes behavior in terms of concepts like mathematical functions on packets.
Language Design
Any network programming language should provide these essential features:

- Modular composition
- Packet classification
- Packet forwarding
Modular Composition

One module for each task

- Monitor
- Route
- Firewall
- Load Balance

NetKAT Language

Run-Time System

Benefits:
- Easier to write, test, and debug
- Enables code reuse
- Provides portability
NetKAT Language

\[
\text{pol ::= } \begin{array}{l}
\text{false} \\
\text{true} \\
\text{field = val} \\
\text{field := val} \\
\text{pol}_1 + \text{pol}_2 \\
\text{pol}_1 ; \text{pol}_2 \\
\neg \text{pol} \\
\text{pol}^* \\
\text{S} \rightarrow \text{S}'
\end{array}
\]
NetKAT Language

pol ::= false
    | true
    | field = val
    | field ::= val
    | pol₁ + pol₂
    | pol₁ ; pol₂
    | !pol
    | pol*
    | S → S'

Boolean Algebra
NetKAT Language

| \text{pol} ::= & \text{false} & \text{Boolean Algebra} \\
| & \text{true} & \text{Kleene Algebra} \\
| & \text{field} = \text{val} \\
| & \text{field} ::= \text{val} \\
| & \text{pol}_1 + \text{pol}_2 \\
| & \text{pol}_1 ; \text{pol}_2 \\
| & \text{!pol} \\
| & \text{pol}^{*} \\
| & \text{S} \rightarrow \text{S}' |
NetKAT Language

| pol ::= | **false** |
|         | **true** |
|         | field = val |
|         | field := val |
|         | pol₁ + pol₂ |
|         | pol₁ ; pol₂ |
|         | !pol |
|         | pol* |
|         | S → S' |

Boolean Algebra

Kleene Algebra

Packet Primitives
NetKAT Language

\[ \text{pol ::= } \text{false} \mid \text{true} \mid \text{field = val} \mid \text{field ::= val} \mid \text{pol}_1 + \text{pol}_2 \mid \text{pol}_1 ; \text{pol}_2 \mid \neg \text{pol} \mid \text{pol}^* \mid S \rightarrow S' \]
NetKAT Language

\[
pol ::= \begin{array}{c}
\text{false} \\
\text{true} \\
\text{field = val} \\
\text{field ::= val} \\
\text{pol} \_1 + \text{pol} \_2 \\
\text{pol} \_1 ; \text{pol} \_2 \\
!\text{pol} \\
\text{pol}^* \\
S \rightarrow S'
\end{array}
\]

Boolean Algebra + Kleene Algebra + Packet Primitives

KAT

NetKAT
NetKAT Language

\[ \text{pol ::= } \text{false} | \text{true} | \text{field = val} | \text{field := val} \]

\[ \text{if } p_1 \text{ then } p_2 \text{ else } p_3 \equiv (p_1 ; p_2) + (!p_1 ; p_3) \]
Semantics

pol ::=  
  | false  
  | true  
  | field = val  
  | field := val  
  | pol₁ + pol₂  
  | pol₁ ; pol₂  
  | !pol  
  | pol*  
  | S ↦ S'
Semantics

Local NetKAT: input-output behavior of switches

\[
\begin{align*}
pol & := \\
& \mid \text{false} \\
& \mid \text{true} \\
& \mid \text{field} = \text{val} \\
& \mid \text{field} := \text{val} \\
& \mid pol_1 + pol_2 \\
& \mid pol_1 ; pol_2 \\
& \mid !pol \\
& \mid pol^* \\
& \mid S \rightarrow S'
\end{align*}
\]

\[[pol] \in \text{Packet} \rightarrow \text{Packet Set}\]
**Semantics**

<table>
<thead>
<tr>
<th>pol ::=</th>
<th>Local NetKAT: input-output behavior of switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td><img src="https://example.com/diagram1.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>true</td>
<td><img src="https://example.com/diagram2.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>field = val</td>
<td><img src="https://example.com/diagram3.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>field ::= val</td>
<td><img src="https://example.com/diagram4.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>pol₁ + pol₂</td>
<td><img src="https://example.com/diagram5.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>pol₁ ; pol₂</td>
<td><img src="https://example.com/diagram6.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>!pol</td>
<td><img src="https://example.com/diagram7.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>pol*</td>
<td><img src="https://example.com/diagram8.png" alt="Diagram of packet behavior" /></td>
</tr>
<tr>
<td>S → S'</td>
<td><img src="https://example.com/diagram9.png" alt="Diagram of packet behavior" /></td>
</tr>
</tbody>
</table>

Global NetKAT: network-wide behavior

\[
[pol] \in \text{Packet} \rightarrow \text{Packet Set}
\]

\[
[pol] \in \text{Trace} \rightarrow \text{Trace Set}
\]
Example

1 A
2
3
4
5 B
6
Local NetKAT Program

A

pol_A

B

pol_B
Local NetKAT Program

```
port := 3
```

```
???
```
Local NetKAT Program

```
port = 1; tag := 1; port := 3
+ port = 2; tag := 2; port := 3
```

???
Local NetKAT Program

Port assignments:
- \( \text{port} = 1; \text{tag} = 1; \text{port} = 3 \)
- \( \text{port} = 2; \text{tag} = 2; \text{port} = 3 \)
- \( \text{tag} = 1; \text{port} = 5 \)
- \( \text{tag} = 2; \text{port} = 6 \)
Local NetKAT Program

Tedious for programmers... difficult to get right!
Global NetKAT Program

port = 1; A ⇔ B;

port := 5 +

port = 2; A ⇔ B; port := 6
Global NetKAT Program

Simple and elegant!

port = 1; A ⇸ B; port := 5

port = 2; A ⇸ B; port := 6
Virtual NetKAT Program

1  A  3  4  B  5

2  6

OpenFlow Switch  OpenFlow Switch
Virtual NetKAT Program

virtual "big switch"
Virtual NetKAT Program

virtual "big switch"

Even simpler!

```
port = 1; port := 5
  +
port = 2; port := 6
```
NetKAT Compiler

NetKAT Compiler Pipeline

3  2  1
NetKAT Compiler Pipeline

[Diagram showing a pipeline with three stages: 3, 2, and Local Compiler. Each stage is connected by arrows.]

- Stage 3: local policy
- Stage 2: forward pattern actions
- Stage 1: Local Compiler

Pattern Actions:
- dstpt=2: drop
- srcpt=7: fwd 1
- *: fwd 2

NetKAT Compiler is ~100x faster than competitors.
NetKAT Compiler

NetKAT Compiler Pipeline

3 global policy

Global Compiler

local policy

Local Compiler

network-wide behavior

~ 100x faster than competitors
NetKAT Compiler

NetKAT Compiler Pipeline

- **Virtual Compiler**: abstract topologies
- **Global Compiler**: network-wide behavior
- **Local Compiler**: ~ 100x faster than competitors

Pattern Actions Table:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstpt=2</td>
<td>drop</td>
</tr>
<tr>
<td>srcpt=7</td>
<td>fwd 1</td>
</tr>
<tr>
<td>*</td>
<td>fwd 2</td>
</tr>
</tbody>
</table>
NetKAT Compiler

NetKAT Compiler Pipeline

- **Virtual Compiler**: abstract topologies
- **Global Compiler**: network-wide behavior
- **Local Compiler**: ~100x faster than competitors

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstpt=2</td>
<td>drop</td>
</tr>
<tr>
<td>srcpt=7</td>
<td>fwd 1</td>
</tr>
<tr>
<td>*</td>
<td>fwd 2</td>
</tr>
</tbody>
</table>
NetKAT Compiler

NetKAT Compiler Pipeline

Virtual Compiler
- abstract topologies
- based on

Global Compiler
- network-wide behavior

Local Compiler
- ~100x faster than competitors

Pattern | Actions
---|---
dstpt=2 | drop
srcpt=7 | fwd 1
* | fwd 2

virtual policy
global policy
local policy
NetKAT Compiler

NetKAT Compiler Pipeline

- **Virtual Compiler**: abstract topologies
- **Global Compiler**: network-wide behavior
- **Local Compiler**: ~ 100x faster than competitors

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstpt=2</td>
<td>drop</td>
</tr>
<tr>
<td>srcpt=7</td>
<td>fwd 1</td>
</tr>
<tr>
<td>*</td>
<td>fwd 2</td>
</tr>
</tbody>
</table>

Based on...
Local Compilation

Input: local program

Output: collection of flow tables, one per switch

Challenges: efficiency and size of generated tables
Local Compilation

Virtual Compiler  Global Compiler  Local Compiler

Input: local program
Output: collection of flow tables, one per switch
Challenges: efficiency and size of generated tables
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
    port := learn

let monitor =
  if (tcpSrc = 22 + tcpDst = 22) then
    port:=console
  else
    false
**Traditional Approach**

```plaintext
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
    port := learn

let monitor =
  if (tcpSrc = 22 + tcpDst = 22) then
    port := console
  else
    false
```

**Pattern** | **Actions**
---|---
src=10.0.0.1 | Fwd 1
src=10.0.0.2 | Fwd 2
* | Controller

**Pattern** | **Actions**
---|---
tcpSrc=22 | Controller
tcpDst=22 | Controller
* | Drop
Traditional Approach

let **route** =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
    port := learn

let **monitor** =
  if (tcpSrc = 22 + tcpDst = 22) then
    port:=console
  else
    false

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>src=10.0.0.1</td>
<td>Fwd 1</td>
</tr>
<tr>
<td>src=10.0.0.2</td>
<td>Fwd 2</td>
</tr>
<tr>
<td>*</td>
<td>Controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpSrc=22</td>
<td>Controller</td>
</tr>
<tr>
<td>tcpDst=22</td>
<td>Controller</td>
</tr>
<tr>
<td>*</td>
<td>Drop</td>
</tr>
</tbody>
</table>
Traditional Approach

let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else
    if ipDst = 10.0.0.2 then
      port := 2
    else
      port := learn

let monitor =
  if (tcpSrc = 22 + tcpDst = 22) then
    port:=console
  else
    false

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>src=10.0.0.1</td>
<td>Fwd 1</td>
</tr>
<tr>
<td>src=10.0.0.2</td>
<td>Fwd 2</td>
</tr>
<tr>
<td>*</td>
<td>Controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpSrc=22</td>
<td>Controller</td>
</tr>
<tr>
<td>tcpDst=22</td>
<td>Controller</td>
</tr>
<tr>
<td>*</td>
<td>Drop</td>
</tr>
</tbody>
</table>

Inefficient!

Tables are a hardware abstraction, not an efficient data structure!!
Our Approach

```plaintext
let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
    port := learn

let monitor =
  if (tcpSrc = 22 + tcpDst = 22) then
    port := console
  else
    false
```
Our Approach

let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
    port := learn

let monitor =
  if (tcpSrc = 22 + tcpDst = 22) then
    port:=console
  else
    false
Our Approach

**let route =**

if ipDst = 10.0.0.1 then
  port := 1
else if ipDst = 10.0.0.2 then
  port := 2
else
  port := learn

**let monitor =**

if (tcpSrc = 22 + tcpDst = 22) then
  port := console
else
  false

Efficient!
Our Approach

let route =
  if ipDst = 10.0.0.1 then
    port := 1
  else if ipDst = 10.0.0.2 then
    port := 2
  else
    port := learn

let monitor =
  if (tcpSrc = 22 + tcpDst = 22) then
    port := console
  else
    false

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipDst=10.0.0.1, tcpSrc=22</td>
<td>Forward 1, Controller</td>
</tr>
<tr>
<td>ipDst=10.0.0.1, tcpDst=22</td>
<td>Forward 1, Controller</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
IR: Forwarding Decision Diagrams

if (tcpSrc = 22 + tcpDst = 22)
  then
    port := console
  else
    drop

Inspired by Binary Decision Diagrams
IR: Forwarding Decision Diagrams

if (tcpSrc = 22 + tcpDst = 22) then
  port := console
else
  drop

Inspired by Binary Decision Diagrams

NetKAT operators (+, ;, *, !) can be implemented efficiently on FDDs using standard BDD techniques
**Global Compilation**

**Virtual Compiler** → **Global Compiler** → **Local Compiler**

**Input:** NetKAT program *(with links)*

**Output:** equivalent local program *(without links)*
Global Compilation

**Input:** NetKAT program *(with links)*

**Output:** equivalent local program *(without links)*
Main Challenges
Main Challenges

1. Adding Extra State "Tagging"
Main Challenges

1. Adding Extra State "Tagging"

2. Avoiding Duplication (naive tagging is unsound!)
Our Solution

Global Program
Our Solution

Adding Extra State
= Translation to Automaton

NetKAT NFA
Our Solution

Adding Extra State
= Translation to Automaton

NetKAT NFA

Avoiding Duplication
= Determinization

NetKAT DFA
Our Solution

Global Program

Adding Extra State = Translation to Automaton

NetKAT NFA

Avoiding Duplication = Determinization

NetKAT DFA

Local Program
Our Solution

Global Program

Adding Extra State
= Translation to Automaton

NetKAT NFA

Automaton Minimization
= Tag Elimination

NetKAT DFA

Avoiding Duplication
= Determinization

Local Program
NetKAT Automata

Transition relation \( \delta : Q \rightarrow \text{Packet} \rightarrow P(Q \times \text{Packet}) \)
NetKAT Automata

Transition relation \( \delta : Q \rightarrow \text{Packet} \rightarrow P(Q \times \text{Packet}) \)

"Alphabet size": \(|\text{Packet} \times \text{Packet}|\)
NetKAT Automata

Transition relation $\delta : Q \rightarrow \text{Packet} \rightarrow P(Q \times \text{Packet})$

"Alphabet size": $|\text{Packet} \times \text{Packet}|$

Can represent $\delta$ symbolically using FDDs!
NetKAT Automata

Transition relation \( \delta : Q \rightarrow \text{Packet} \rightarrow P(Q \times \text{Packet}) \)

"Alphabet size" : \(|\text{Packet} \times \text{Packet}|\)

Can represent \( \delta \) symbolically using FDDs!

Automata construction:
Antimirov partial derivatives & Position Automata
Virtual Compilation

Input: program written against virtual topology

Output: global program that simulates virtual behavior
Virtual Compilation

**Input:** program written against virtual topology

**Output:** global program that simulates virtual behavior
Virtualization

Virtual

Physical
Virtualization
Can formulate execution as a two-player game...

The compiler synthesizes a physical program that encodes a winning strategy to all instances of the game.
Formal Reasoning
Motivation

Networks are now a critical part of our computing infrastructure...

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

...and are quickly becoming unwieldy for operators to manage!
Network Management

Current approach:

- Inspect configurations through command-line interfaces
- Diagnose errors using tools like ping and traceroute
Network Management

Current approach:

- Inspect configurations through command-line interfaces
- Diagnose errors using tools like ping and traceroute

Better alternative:

- Encode configurations into a high-level language
- Verify invariants (connectivity, loop freedom, etc.) automatically
Focus on \textit{reachability properties} that capture the essential function of a network: moving data from one location to another.
Encoding Networks

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

<table>
<thead>
<tr>
<th>Pattern</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>

**Code:**

```plaintext
if dstport=22 then false
elsif srcip=10.0.0.1 then port := 1
else port := 2
```

A → B + B → A + B → C + C → B
Encoding Networks

An entire network can be represented in NetKAT by interleaving steps of processing by switches and topology.

\[
\begin{align*}
\text{policy} & \rightarrow \text{topo} \\
\text{policy} & + (\text{policy}; \text{topo}); \text{policy} \\
& + (\text{policy}; \text{topo}; \text{policy}; \text{topo}); \text{policy} \\
& : (\text{policy}; \text{topo})^*; \text{policy}
\end{align*}
\]
Given a network encoded this way, we’d like to be able to automatically answer questions like:

“Does the network forward from ingress to egress?”

Can reduce this question (and others) to equivalence

\[
\text{in;} (\text{policy;} \text{topo})^*; \text{policy;} \text{out} \equiv \text{in;} \text{out}
\]
Given a network encoded this way, we’d like to be able to automatically answer questions like:

“Does the network forward from ingress to egress?”

Can reduce this question (and others) to equivalence:

\[ \text{in;} \ (\text{policy;} \ \text{topo})^*; \ \text{policy;} \ \text{out} \equiv \text{in;} \ \text{out} \]

Other properties:
- Access control
- Traffic Isolation
- Loop freedom
- Blackhole freedom
### Boolean Algebra Axioms

\[
\begin{align*}
    a + (b ; c) & \equiv (a + b) ; (a + c) \\
    a + \text{true} & \equiv \text{true} \\
    a + !a & \equiv \text{true} \\
    a ; b & \equiv b ; a \\
    a ; !a & \equiv \text{false} \\
    a ; a & \equiv a
\end{align*}
\]

### Kleene Algebra Axioms

\[
\begin{align*}
    p + (q + r) & \equiv (p + q) + r \\
    p + q & \equiv q + p \\
    p + \text{false} & \equiv p \\
    p + p & \equiv p \\
    p ; (q ; r) & \equiv (p ; q) ; r \\
    p ; (q + r) & \equiv p ; q + p ; r \\
    (p + q) ; r & \equiv p ; r + q ; r \\
    \text{true} ; p & \equiv p \\
    p & \equiv p ; \text{true} \\
    \text{false} ; p & \equiv \text{false} \\
    p ; \text{false} & \equiv \text{false} \\
    \text{true} + p ; p^* & \equiv p^* \\
    \text{true} + p^* ; p & \equiv p^* \\
    p + q ; r + r & \equiv r \Rightarrow p^* ; q + r & \equiv r \\
    p + q ; r + q & \equiv q \Rightarrow p ; r^* + q & \equiv q \\
\end{align*}
\]

### Packet Axioms

\[
\begin{align*}
    f := n ; f' := n' & \equiv f' := n ; f := n & \text{if } f \neq f' \\
    f := n ; f' = n' & \equiv f' = n ; f := n & \text{if } f \neq f' \\
    f := n ; f = n & \equiv f := n \\
    f = n ; f := n & \equiv f = n \\
    f := n ; f' = f := n' & \equiv f := n' & \text{if } n \neq n' \\
    f = n ; f' = \text{false} & \equiv \text{false} & \text{if } n \neq n' \\
    A \rightarrow B ; f = n & \equiv f = n ; A \rightarrow B & \text{if } f \neq \text{switch}
\end{align*}
\]

### NetKAT Equational Axioms

\[
\begin{align*}
    f := n ; f' := n' & \equiv f' := n ; f := n & \text{if } f \neq f' \\
    f := n ; f' = n' & \equiv f' = n ; f := n & \text{if } f \neq f' \\
    f := n ; f = n & \equiv f := n \\
    f = n ; f := n & \equiv f = n \\
    f := n ; f' = f := n' & \equiv f := n' & \text{if } n \neq n' \\
    f = n ; f' = \text{false} & \equiv \text{false} & \text{if } n \neq n' \\
    A \rightarrow B ; f = n & \equiv f = n ; A \rightarrow B & \text{if } f \neq \text{switch}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Kleene Algebra Axioms</th>
<th>Boolean Algebra Axioms</th>
<th>Packet Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p + (q + r) = (p + q) + r)</td>
<td>(a + (b ; c) = (a + b) ; (a + c))</td>
<td>(f := n; f' := n' = f' := n'; f := n) if (f \neq f')</td>
</tr>
<tr>
<td>(p + q = q + p)</td>
<td>(a + \text{true} = \text{true})</td>
<td>(f := n; f' = n' = f' = n'; f := n) if (f \neq f')</td>
</tr>
<tr>
<td>(p + \text{false} = p)</td>
<td>(a + \text{true} = \text{true})</td>
<td>(f := n; f = n = f := n)</td>
</tr>
<tr>
<td>(p + p = p)</td>
<td>(a + \text{true} = \text{true})</td>
<td>(f := n; f := n = f := n)</td>
</tr>
<tr>
<td>(p; (q; r) = (p; q); r)</td>
<td>(a + \text{false} = \text{false})</td>
<td>(f := n; f := n' = f := n')</td>
</tr>
<tr>
<td>(p; (q + r) = p; q + p; r)</td>
<td>(a ; a = a)</td>
<td>(f = n; f = n' = \text{false}) if (n \neq n')</td>
</tr>
<tr>
<td>((p + q); r = p; r + q; r)</td>
<td>(a ; a = a)</td>
<td>(A)</td>
</tr>
</tbody>
</table>
NetKAT Equational Axioms

Kleene Algebra Axioms
\[ p + (q + r) = (p + q) + r \]
\[ p + q = q + p \]
\[ p + \text{false} = p \]
\[ p + p = p \]
\[ p; (q; r) = (p; q); r \]
\[ p; (q + r) = p; q + p; r \]
\[ (p + q); r = p; r + q; r \]
\[ \text{true}; p = p \]
\[ p = p; \text{true} \]
\[ \text{false}; p = \text{false} \]
\[ p; \text{false} = \text{false} \]
\[ \text{true} \]
\[ \text{true} \]
\[ p \]

Boolean Algebra Axioms
\[ a + (b ; c) = (a + b) ; (a + c) \]
\[ a + \text{true} = \text{true} \]
\[ a + ! a = \text{true} \]
\[ a ; b = b ; a \]
\[ a ; ! a = \text{false} \]
\[ a ; a = a \]

Packet Axioms
\[ f := n; f' := n' = f' := n'; f := n \quad \text{if } f \neq f' \]
\[ f := n; f' = n' = f' = n'; f := n \quad \text{if } f \neq f' \]
\[ f := n; f = n = f := n \]
\[ f = n; f := n = f = n \]
\[ f := n; f := n' = f := n' \]
\[ f = n; f = n' = \text{false} \quad \text{if } n \neq n' \]
A
Boolean Algebra Axioms

\[ a + (b \land c) \equiv (a + b) \land (a + c) \]
\[ a + \text{true} \equiv \text{true} \]
\[ a + \neg a \equiv \text{true} \]
\[ a \land b \equiv b \land a \]
\[ a \land \neg a \equiv \text{false} \]
\[ a \land a \equiv a \]

Kleene Algebra Axioms

\[ p + (q + r) \equiv (p + q) + r \]
\[ p + q \equiv q + p \]
\[ p + \text{false} \equiv p \]
\[ p + p \equiv p \]
\[ p; (q; r) \equiv (p; q); r \]
\[ p; (q + r) \equiv p; q + p; r \]
\[ (p + q); r \equiv p; r + q; \]
\[ \text{true}; p \equiv p \]
\[ p \equiv p; \text{true} \]
\[ \text{false}; p \equiv \text{false} \]
\[ p; \text{false} \equiv \text{false} \]
\[ \text{true} \]
\[ \text{true} \]
\[ p \]
\[ p \]

Packet Axioms

\[ f := n; f' := n' \equiv f' := n'; f := n \quad \text{if } f \neq f' \]
\[ f := n; f' = n' \equiv f' = n'; f := n \quad \text{if } f \neq f' \]
\[ f := n; f = n \equiv f := n \]
\[ f = n; f := n \equiv f = n \]
\[ f := n; f := n' \equiv f := n' \]
\[ f = n; f = n' \equiv \text{false} \quad \text{if } n \neq n' \]

NetKAT Equational Axioms
Kleene Algebra Axioms
\[ p + (q + r) \equiv (p + q) + r \]
\[ p + q \equiv q + p \]
\[ p + \text{false} \equiv p \]
\[ p + p \equiv p \]
\[ p; (q; r) \equiv (p; q); r \]
\[ p; (q + r) \equiv p; q + p; r \]
\[ (p + q); r \equiv p; r + q; r \]
\[ \text{true}; p \equiv p \]
\[ p \equiv p; \text{true} \]
\[ \text{false}; p \equiv \text{false} \]
\[ p; \text{false} \equiv \text{false} \]
\[ \text{true} \]
\[ \text{true} \]
\[ p \]
\[ p \]

Boolean Algebra Axioms
\[ a + (b ; c) \equiv (a + b) ; (a + c) \]
\[ a + \text{true} \equiv \text{true} \]
\[ a + ! a \equiv \text{true} \]
\[ a ; b \equiv b ; a \]
\[ a ; ! a \equiv \text{false} \]
\[ a ; a \equiv a \]

Packet Axioms
\[ f := n; f' := n' \equiv f' := n'; f := n \quad \text{if } f \neq f' \]
\[ f := n; f' = n' \equiv f' = n'; f := n \quad \text{if } f \neq f' \]
\[ f := n; f = n \equiv f := n \]
\[ f := n; f = n \equiv \text{false} \quad \text{if } n \neq n' \]
\[ A \]
Kleene Algebra Axioms

\( p + (q + r) \equiv (p + q) + r \)
\( p + q \equiv q + p \)
\( p + \text{false} \equiv p \)
\( p + p \equiv p \)
\( p; (q; r) \equiv (p; q); r \)
\( p; (q + r) \equiv p; q + p; r \)
\( (p + q); r \equiv p; r + q; r \)
\( \text{true}; p \equiv p \)
\( p = p; \text{true} \)
\( \text{false}; p \equiv \text{false} \)
\( p; \text{false} \equiv \text{false} \)
\( \text{true} \equiv \text{true} \)
\( \text{true} \equiv \text{true} \)
\( p \)
\( p \)

Boolean Algebra Axioms

\( a + (b ; c) \equiv (a + b) ; (a + c) \)
\( a + \text{true} \equiv \text{true} \)
\( a + ! a \equiv \text{true} \)
\( a ; b \equiv b ; a \)
\( a ; !a \equiv \text{false} \)
\( a ; a \equiv a \)

Packet Axioms

\( f := n; f' := n' \equiv f' := n'; f := n \quad \text{if } f \neq f' \)
\( f := n; f' = n' \equiv f' = n'; f := n \quad \text{if } f \neq f' \)
\( f := n; f = n \equiv f := n \)
\( f = n; f := n \equiv f = n \)
\( f := n; f := n' \equiv f := n' \)
\( f = n; f = n' \equiv \text{false} \quad \text{if } n \neq n' \)

NetKAT Equational Axioms
NetKAT Equational Axioms

**Kleene Algebra Axioms**
- \( p + (q + r) = (p + q) + r \)
- \( p + q = q + p \)
- \( p + \text{false} = p \)
- \( p + p = p \)
- \( p; (q; r) = (p; q); r \)
- \( (p + q); r \equiv p; r + q; r \)
- \( p + q \equiv (p; q) + r \)

**Boolean Algebra Axioms**
- \( a + (b ; c) \equiv (a + b) ; (a + c) \)
- \( a + \text{true} = \text{true} \)
- \( a + \neg a \equiv \text{true} \)
- \( a ; b \equiv b ; a \)
- \( a ; \neg a \equiv \text{false} \)
- \( a ; a \equiv a \)

**NetKAT Equational Axioms**

**Soundness:** If \( \vdash p \equiv q \), then \( \llbracket p \rrbracket = \llbracket q \rrbracket \)

**Completeness:** If \( \llbracket p \rrbracket = \llbracket q \rrbracket \), then \( \vdash p \equiv q \)
Reduced NetKAT

Complete tests
\[ \alpha ::= \text{switch} = n \cdot \text{port} = n \]

Complete assignments
\[ \beta ::= \text{switch} := n \cdot \text{port} := n \]

Reduced terms
\[ p, q ::= \alpha \quad (* \text{complete test} *) \\
| \beta \quad (* \text{complete assignment} *) \\
| p + q \quad (* \text{union} *) \\
| p; q \quad (* \text{sequence} *) \\
| p^* \quad (* \text{Kleene star} *) \\
| \text{dup} \quad (* \text{Duplication} *) \]
Reduced NetKAT

Complete tests
\[ \alpha ::= \text{switch} = n \cdot \text{port} = n \]

Complete assignments
\[ \beta ::= \text{switch} := n \cdot \text{port} := n \]

Reduced terms
\[ p, q ::= \alpha \quad (\text{* complete test *}) \]
\[ \beta \quad (\text{* complete assignment *}) \]
\[ p + q \quad (\text{* union *}) \]
\[ p; q \quad (\text{* sequence *}) \]
\[ p^* \quad (\text{* Kleene star *}) \]
\[ \text{dup} \quad (\text{* Duplication *}) \]

For simplicity, only consider two fields
Reduced NetKAT

Complete tests
\[ \alpha ::= \text{switch} = n \cdot \text{port} = n \]

Complete assignments
\[ \beta ::= \text{switch} := n \cdot \text{port} := n \]

Reduced terms
\[ p,q ::= \alpha \quad (\ast \text{complete test} \ast) \]
\[ \beta \quad (\ast \text{complete assignment} \ast) \]
\[ p + q \quad (\ast \text{union} \ast) \]
\[ p; q \quad (\ast \text{sequence} \ast) \]
\[ p^* \quad (\ast \text{Kleene star} \ast) \]
\[ \text{dup} \quad (\ast \text{Duplication} \ast) \]

**Lemma:** For every NetKAT term \( p \), there is a reduced NetKAT term \( p' \) such that \( \vdash p \equiv p' \)
Regular Interpretation

Can interpret terms as regular languages over an alphabet of complete tests/assignments and a new operator \texttt{dup}:
Can interpret terms as regular languages over an alphabet of complete tests/assignments and a new operator \texttt{dup}:

\[ R(p) \subseteq (A \cup \Pi \cup \{\texttt{dup}\})^* \]

- \( R(\alpha) = \{\alpha\} \)
- \( R(\pi) = \{\pi\} \)
- \( R(p + q) = R(p) \cup R(q) \)
- \( R(p ; q) = R(p) ; R(q) \)
- \( R(p^*) = R(p)^* \)
- \( R(\texttt{dup}) = \{\texttt{dup}\} \)

Unfortunately \( \llbracket p \rrbracket = \llbracket q \rrbracket \) does not imply \( R(p) = R(q) \).
Can interpret terms as regular languages over an alphabet of complete tests/assignments and a new operator \texttt{dup}:

\textbf{Regular Interpretation:} $R(p) \subseteq (A \cup \Pi \cup \{\texttt{dup}\})^*$

- $R(\alpha) = \{\alpha\}$
- $R(\pi) = \{\pi\}$
- $R(p + q) = R(p) \cup R(q)$
- $R(p; q) = R(p) ; R(q)$
- $R(p^*) = R(p)^*$
- $R(\texttt{dup}) = \{\texttt{dup}\}$

Unfortunately $\llbracket p \rrbracket = \llbracket q \rrbracket$ does not imply $R(p) = R(q)$

\textbf{Example:}

$sw=1; \ pt=1; \ sw=1; \ pt=2 \sim sw=1; \ pt=1; \ sw=2; \ pt=1$
Language Model
Language Interpretation: $G(p) \subseteq A \cdot (B \cdot \{\text{dup}\})* \cdot B$

$G(\alpha) = \{\alpha \cdot \pi_\alpha\}$

$G(\beta) = \{\alpha \cdot \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \cdot q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\text{dup}) = \{\alpha \cdot \beta_\alpha \cdot \text{dup} \cdot \beta_\alpha \mid \alpha \in A\}$
Language Interpretation: $G(p) \subseteq A \cdot (B \cdot \{\text{dup}\})^* \cdot B$

$G(\alpha) = \{\alpha \cdot \pi_\alpha\}$

$G(\beta) = \{\alpha \cdot \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \cdot q) = G(p) \odot G(q)$

$G(p^*) = G(p)^*$

$G(\text{dup}) = \{\alpha \cdot \beta_\alpha \cdot \text{dup} \cdot \beta_\alpha \mid \alpha \in A\}$
Language Model

Language Interpretation: $G(p) \subseteq A \cdot (B \cdot \{\text{dup}\})^* \cdot B$

$G(\alpha) = \{ \alpha \cdot \pi_\alpha \}$

$G(\beta) = \{ \alpha \cdot \beta \mid \alpha \in A \}$

$G(p + q) = G(p) \cup G(q)$

$G(p \cdot q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\text{dup}) = \{ \alpha \cdot \beta_\alpha \cdot \text{dup} \cdot \beta_\alpha \mid \alpha \in A \}$

Guarded strings

Guarded concatenation
Language Model

Language Interpretation: $G(p) \subseteq A \cdot (B \cdot \{\text{dup}\})^* \cdot B$

- $G(\alpha) = \{\alpha \cdot \pi_\alpha\}$
- $G(\beta) = \{\alpha \cdot \beta \mid \alpha \in A\}$
- $G(p + q) = G(p) \cup G(q)$
- $G(p \cdot q) = G(p) \cdot G(q)$
- $G(p^*) = G(p)^*$
- $G(\text{dup}) = \{\alpha \cdot \beta_\alpha \cdot \text{dup} \cdot \beta_\alpha \mid \alpha \in A\}$

**Example:** $\alpha_1 \cdot \beta_2 \cdot \text{dup} \cdot \beta_3 \cdot \text{dup} \cdot ... \cdot \text{dup} \cdot \beta_n$
Language Interpretation: $G(p) \subseteq A \cdot (B \cdot \{\text{dup}\})^* \cdot B$

$G(\alpha) = \{\alpha \cdot \pi_\alpha\}$

$G(\beta) = \{\alpha \cdot \beta \mid \alpha \in A\}$

$G(p + q) = G(p) \cup G(q)$

$G(p \cdot q) = G(p) \diamond G(q)$

$G(p^*) = G(p)^*$

$G(\text{dup}) = \{\alpha \cdot \beta_\alpha \cdot \text{dup} \cdot \beta_\alpha \mid \alpha \in A\}$

**Example:** $\alpha_1 \cdot \beta_2 \cdot \text{dup} \cdot \beta_3 \cdot \text{dup} \cdot \ldots \cdot \text{dup} \cdot \beta_n$

**Intuition:** models packet trajectories through the network
Language Model

Language Interpretation: \( G(p) \subseteq A \cdot (B \cdot \{\text{dup}\})^* \cdot B \)

- \( G(\alpha) = \{\alpha \cdot \pi_\alpha\} \)
- \( G(\beta) = \{\alpha \cdot \beta \mid \alpha \in A\} \)
- \( G(p + q) = G(p) \cup G(q) \)
- \( G(p \cdot q) = G(p) \bowtie G(q) \)
- \( G(p^*) = G(p)^* \)
- \( G(\text{dup}) = \{\alpha \cdot \beta_\alpha \cdot \text{dup} \cdot \beta_\alpha \mid \alpha \in A\} \)

Guarded strings

Guarded concatenation

Example: \( \alpha_1 \cdot \beta_2 \cdot \text{dup} \cdot \beta_3 \cdot \text{dup} \cdot \ldots \cdot \text{dup} \cdot \beta_n \)

Intuition: models packet trajectories through the network

Theorem: \([p] = [q]\) if and only if \(G(p) = G(q)\)
Recall that, by definition, $R(p) \subseteq (A \cup \Pi \cup \{\text{dup}\})^*$
Recall that, by definition, $R(p) \subseteq (A \cup \Pi \cup \{\text{dup}\})*$

**Normal Form**

A term $p$ is said to be in *normal form* if $R$ generates a set of strings in the language model

$R(p) \subseteq A; (\Pi; \{\text{dup}\})*; \Pi$
Normal Forms

Recall that, by definition, $R(p) \subseteq (A \cup \Pi \cup \{\text{dup}\})^*$

**Normal Form**

A term $p$ is said to be in *normal form* if $R$ generates a set of strings in the language model

$$R(p) \subseteq A; (\Pi; \{\text{dup}\})^*; \Pi$$

**Lemma:** For all terms $p$, there exists a normal form $\hat{p}$ such that $\vdash p \equiv \hat{p}$
Recall that, by definition, $R(p) \subseteq (A \cup \Pi \cup \{\text{dup}\})^*$

**Normal Form**

A term $p$ is said to be in *normal form* if $R$ generates a set of strings in the language model

$$R(p) \subseteq A; (\Pi; \{\text{dup}\})^*; \Pi$$

**Lemma:** For all terms $p$, there exists a normal form $\hat{p}$ such that $\vdash p \equiv \hat{p}$

**Lemma:** If $p$ is in normal form, then $R(p) = G(p)$
Completeness Proof

\( p \) and \( q \) such that \( \llbracket p \rrbracket = \llbracket q \rrbracket \)
Completeness Proof

\[ p \ \text{and} \ q \ \text{such that} \ \llbracket p \rrbracket = \llbracket q \rrbracket \]

\[ \vdash p \equiv \hat{p} \ \text{and} \ \vdash q \equiv \hat{q} \]

Reduce and Normalize
Completeness Proof

\( p \) and \( q \) such that \( \llbracket p \rrbracket = \llbracket q \rrbracket \)

\[ \vdash p = \hat{p} \quad \text{and} \quad \vdash q = \hat{q} \]

\[ \llbracket \hat{p} \rrbracket = \llbracket \hat{q} \rrbracket \]

Reduce and Normalize

Soundness
Completeness Proof

\[ p \text{ and } q \text{ such that } \llbracket p \rrbracket = \llbracket q \rrbracket \]

\[ \vdash p = \hat{p} \quad \text{and} \quad \vdash q = \hat{q} \]

\[ \llbracket \hat{p} \rrbracket = \llbracket \hat{q} \rrbracket \]

\[ G(\hat{p}) = G(\hat{q}) \]

Reduce and Normalize

Soundness

Language Model
Completeness Proof

\( p \) and \( q \) such that \([p] = [q]\)

\( \vdash p = \hat{p} \) and \( \vdash q = \hat{q} \)

\([\hat{p}] = [\hat{q}]\)

\( G(\hat{p}) = G(\hat{q}) \)

\( R(\hat{p}) = R(\hat{q}) \)

Reduce and Normalize

Soundness

Language Model

Normal Forms
Completeness Proof

\( p \) and \( q \) such that \( \llbracket p \rrbracket = \llbracket q \rrbracket \)

\( \vdash p = \hat{p} \) and \( \vdash q = \hat{q} \)

\( \llbracket \hat{p} \rrbracket = \llbracket \hat{q} \rrbracket \)

\( G(\hat{p}) = G(\hat{q}) \)

\( R(\hat{p}) = R(\hat{q}) \)

\( \vdash \hat{p} \equiv \hat{q} \)

Reduce and Normalize

Soundness

Language Model

Normal Forms

Kleene Algebra Completeness

[ Kozen '94 ]
Completeness Proof

\[ p \text{ and } q \text{ such that } \llbracket p \rrbracket = \llbracket q \rrbracket \]

\[ \vdash p = \hat{p} \text{ and } \vdash q = \hat{q} \]

\[ \llbracket \hat{p} \rrbracket = \llbracket \hat{q} \rrbracket \]

\[ G(\hat{p}) = G(\hat{q}) \]

\[ R(\hat{p}) = R(\hat{q}) \]

\[ \vdash \hat{p} = \hat{q} \]

\[ \vdash p \equiv q \]

Reduce and Normalize

Soundness

Language Model

Normal Forms

Kleene Algebra Completeness [Kozen '94]

Transitivity

Reduce and Normalize
Can exploit NetKAT’s regular structure to build finite automata

This provides a practical way to decide program equivalence...

... but need symbolic representations to get good performance

\[(x=1; x:=2; A \Rightarrow B + x=2; x:=1; B \Rightarrow A)^*\]
**Verified Controllers**

**Question:** How can we trust the compiler and run-time?

**Answer:** implement it in a proof assistant!

- Formalize source and target languages in Coq
- Prove that transformations preserve semantics
- Extract code to OCaml and execute on switches
Experience
Compiler vs State of the Art

Two orders of magnitude speedup!
Verification Benchmarks

Networks:
• Topology Zoo
• FatTree
• Stanford Backbone

Policies:
• Shortest-path forwarding
• Stanford production policy

Questions:
• Point-to-point reachability
• All-Pairs connectivity
• Loop freedom
• Translation validation
Verification Benchmarks

Topology Zoo
- Connectivity
- Loop Freedom
- Translation Validation

FatTree
- Scalability
- Relative Performance

Stanford Backbone
Point-to-point reachability in 0.67s (vs 13s for HSA)
Ongoing Work
Now that we have the necessary models of sender, receiver, and network, we can start to look at how a system responds to the different components of the traffic flow.

**Probabilistic Behavior**

- **Congestion**: given a model of traffic received at the ingress, predict the amount of congestion on each link

- **Failures**: given a model of device/link failure, predict the likelihood that some packets will be dropped

- **Randomization**: use advanced routing schemes such as Valiant load balancing, oblivious routing, gossip etc.
Probabilistic NetKAT

pol ::= \texttt{false} \mid \texttt{true} \mid \texttt{field = val} \mid \texttt{pol}_1 \& \texttt{pol}_2 \mid \texttt{pol}_1 ; \texttt{pol}_2 \mid \neg \texttt{pol} \mid \texttt{pol}^* \mid \texttt{pol}_1 \oplus_a \texttt{pol}_2 \mid \texttt{field := val} \mid S \rightarrow T
The semantics of the language turns out to be subtle...

- Define a measurable space over sets of packet traces
- Give semantics in terms of Markov kernels
- New & operator combines distributions on sets of histories
- Kleene star defined in terms of an infinite stochastic process
Other Work

Declarative Queries [ICFP 2011]
- Declarative language for reading network state
- Decouples monitoring from forwarding

Reactive Compilation [POPL 2012]
- Expressive intermediate language
- Efficient proactive compiler

Consistent Updates [SIGCOMM 2012]
- Policy updates with strong consistency guarantees
- Runtime system automatically applies optimizations

Modular Composition [NSDI 2013]
- Virtual networks via topology views
- Implementation via sequential composition

Protocol Synthesis [PLDI 2015]
- Generate update protocols from formal specifications
- Incremental model checker improves performance
Conclusion
Conclusion

Fast, Flexible, and Fully implemented in OCaml:
http://github.com/frenetic-lang/frenetic/

Go ahead and use it!
(others are using it already)