Software Synthesis for Networks

Nate Foster
Cornell / Barefoot
Software-Defined Networking

Global Visibility and Control

Open APIs

Programmable Data Planes
Software-Defined Networking

Controller

Global Visibility and Control

Open APIs

Switch

Your Program goes here!

Programmable Data Planes
But how do we write all of this software?
Software Synthesis

What if programmers could...

• **Sketch** the structure of their program...

• Give **examples** and **scenarios**...

• Specify functional **behavior**...

• Write down high-level **requirements**...

• Express resource **constraints**...

...and a tool **automatically synthesized** a correct and efficient implementation?
Software Synthesis

Specification
Software Synthesis

Specification

Synthesizer
Software Synthesis

Specification

Synthesizer

Program
Software Synthesis

Specification → Synthesizer → Program
Software Synthesis

Synthesizer

Program
Software Synthesis

Partial Program

Synthesizer

Program
Software Synthesis

Partial Program

Logical Formula

Synthesizer

Program
Software Synthesis

Partial Program

Logical Formula

Input-Output Examples

Synthesizer

Program
Software Synthesis

Programmers can express their insights in a wide variety of ways, not just in standard code!
What?!
Synthesis for Networks
Synthesis for Networks

- Programs are large, but simple and highly structured—e.g., loop free!
- The desired behavior of the network is often clear (at least at an intuitive level)
- Most difficult aspects of network programming stem from limited resources and inherent concurrency
This Talk

Synthesis is an effective means for automating some of the trickiest aspects of network programming
This Talk

Synthesis is an effective means for automating some of the trickiest aspects of network programming

Outline:

• Network Update Synthesis
• Event-Driven Network Programs
• Future Directions
Efficient Synthesis of Network Updates
Dynamic SDN Applications

Application

Controller Platform

Controller Run-Time

Network-wide Configuration

- Topology change
- Host change
- Traffic statistics
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?

Problem: naive updates can break important invariants!
Example: Data Center
Update: upd T1; upd C2; upd A3; upd A1
Naive Update
Naive Update
Naive Update
Naive Update
Naive Update
Problem: *naive update creates a blackhole!*
Naive Update
Naive Update
Naive Update
Naive Update
Problem: naive update leads to access control violation!
At 12:47 AM PDT on April 21st, a network change was performed as part of our normal scaling activities...

During the change, one of the steps is to shift traffic off of one of the redundant routers...

The traffic shift was executed incorrectly and the traffic was routed onto the lower capacity redundant network.

This led to a “re-mirroring storm”...

During this re-mirroring storm, the volume of connection attempts was extremely high and nodes began to fail, resulting in more volumes left needing to re-mirror. This added more requests to the re-mirroring storm...

The trigger for this event was a network configuration change.
Per-Packet Consistency

Consistency Guarantee: every packet (or flow) in the network “sees” a single policy version

Two-Phase Update:
• Tag configurations with versions
• Install new configuration in core
• Install new configuration at edge
• Wait for in-flight packets to exit
• Delete old configurations
Per-Packet Consistency

Consistency Guarantee: every packet (or flow) in the network “sees” a single policy version

Two-Phase Update:
- Tag configurations with versions
- Install new configurations in core
- Install new configurations at edge
- Wait for in-flight packets to exit
- Delete old configurations

Limitations:
- Doubles peak memory usage
- Updates are slow to implement
Theorem (Universal Property Preservation): a network update is per-packet consistent if and only if it preserves all safety properties.
Per-Packet Consistent Updates

Theorem (Universal Property Preservation): a network update is per-packet consistent if and only if it preserves all safety properties.

Questions:
• Can we implement a per-packet consistent update by simply updating switches in the right order?
• If not, can we relax the requirements in a reasonable way to obtain an efficient mechanism?
Example: Data Center
Example: Data Center

Update: upd T1; upd C2; upd A3; upd A1

✔
Example: Data Center

Update: upd T1; upd C2; upd A3; upd A1
Example: Data Center

Update: upd T1; upd C2; upd A3; upd A1
Example: Data Center

Update: upd T1; upd C2; upd A3; upd A1
Example: Data Center

Update: upd T1; upd C2; upd A3; upd A1 ✔
Naive Update

Diagram showing network connections and nodes labeled with letters and numbers.
Naive Update

- **Update:** upd A2; upd A4; upd T1; upd C1 ✗
- **Update:** upd A2; upd A4; upd C1; upd T1 ✗
- There is **no update** that ensures per-packet consistency
Relaxing Per-Packet Consistency
Relaxing Per-Packet Consistency

Idea: all packets eventually delivered via $A_1$ or $A_4$
Relaxing Per-Packet Consistency

Idea: all packets eventually delivered via $A_1$ or $A_4$

- **Update:** upd $A_2$; upd $A_4$; upd $T_1$; upd $C_1$  ✗

- **Update:** upd $A_2$; upd $A_4$; upd $C_1$; upd $T_1$  ✔
How to Specify Properties?

Reachability: every packet that starts at $s_i$ reaches $d_i$
How to Specify Properties?

Reachability: every packet that starts at $s_i$ reaches $d_i$
How to Specify Properties?

**Reachability:** every packet that starts at $s_i$ reaches $d_i$

$LTL: \land_i (s_i \rightarrow F d_i)$
How to Specify Properties?

Reachability: every packet that starts at $s_i$ reaches $d_i$

LTL: $\land_i (s_i \rightarrow F d_i)$

Waypointing: all packets traverse $w$ before exiting
How to Specify Properties?

**Reachability:** every packet that starts at $s_i$ reaches $d_i$

$LTL$: $\bigwedge_i (s_i \rightarrow F d_i)$

**Waypointing:** all packets traverse $w$ before exiting
How to Specify Properties?

Reachability: every packet that starts at \( s_i \) reaches \( d_i \)

\[
\bigwedge_i (s_i \rightarrow F d_i)
\]

Waypointing: all packets traverse \( w \) before exiting

\[
\neg g \mathbf{U} w_2 \land F g
\]
How to Specify Properties?

**Reachability:** every packet that starts at $s_i$ reaches $d_i$

LTL: $\land_i (s_i \rightarrow F d_i)$

**Waypointing:** all packets traverse $w$ before exiting

LTL: $\neg g U w_2 \land F g$

**Chaining:** all packets traverse $w_1$ and $w_2$ before exiting
How to Specify Properties?

Reachability: every packet that starts at $s_i$ reaches $d_i$

LTL: $\land_i (s_i \rightarrow F d_i)$

Waypointing: all packets traverse $w$ before exiting

LTL: $\neg g \ U w_2 \land F g$

Chaining: all packets traverse $w_1$ and $w_2$ before exiting
How to Specify Properties?

**Reachability:** every packet that starts at $s_i$ reaches $d_i$

$$\LTL: \bigwedge_i (s_i \rightarrow F d_i)$$

**Waypointing:** all packets traverse $w$ before exiting

$$\LTL: \neg g U w_2 \land F g$$

**Chaining:** all packets traverse $w_1$ and $w_2$ before exiting

$$\LTL: \neg g U w_2 \land \neg w_2 U w_1 \land F g$$
Network Update Synthesis

- Initial and Final Configurations
- LTL Specification
- Update at most once

Update Synthesizer

Update Program
Synthesis Algorithm

\[ \phi \]

LTL Specification

Old and New Configurations
Synthesis Algorithm

$\phi$

LTL Specification
Synthesis Algorithm

$LTL$ Specification

$\phi$
Synthesis Algorithm

Depth-First Search:

- Attempt to update the switches one-by-one
- Backtrack whenever a bad configuration is reached
Synthesis Algorithm

Depth-First Search:
• Attempt to update the switches one-by-one
• Backtrack whenever a bad configuration is reached

Challenges:
• Search space is huge
• Checking a configuration means solving an LTL model checking problem (PSPACE-complete)!
Synthesis Algorithm

Depth-First Search:
- Attempt to update the switches one-by-one
- Backtrack whenever a bad configuration
- Searching a configuration
- Checking a configuration means solving an LTL model checking problem (PSPACE-complete)!

Two main ideas:
- **Learn from counter-examples** to aggressively prune the search space
- Use an **incremental model checker**
Main Limitation

For some topologies, configurations, and specifications, there is no correct ordering we can use.
Main Limitation

For some topologies, configurations, and specifications, there is no correct ordering we can use.

Example: "double diamond"

[DISC '16]
Main Limitation

For some topologies, configurations, and specifications, there is no correct ordering we can use.

Example: "double diamond"

Our implementation reverts to a two-phase update...
Evaluation

Questions:
- Impact of optimizations:
  - Pruning search space
  - Incremental model checking
- Scalability of approach:
  - Topology
  - Complexity of specifications
  - Total space explored
Evaluation

Questions:
• Impact of optimizations:
  ‣ Pruning search space
  ‣ Incremental model checking
• Scalability of approach:
  ‣ Topology
  ‣ Complexity of specifications
  ‣ Total space explored

Methodology:
• Real-world topologies (TopoZoo, FatTrees, Small World)
• Synthetic configurations (e.g., shortest-path forwarding)
• Standard properties (reachability, waypointing, etc.)
Impact of Optimizations

- **Configurations:** shortest-path forwarding
- **LTL Specification:** all-pairs reachability
Scalability

- **Configurations:** "diamond" / "double diamond"
- **Specifications:** reachability, waypointing, chaining
Event-Driven Network Programming
Dynamic SDN Applications

Want to push stateful computation into the dataplane!
Example: Stateful Firewall
Example: Stateful Firewall

Program:
- $H_1$ may send packets to $H_4$
- Upon receiving a packet, $H_4$ may send packets to $H_1$
Example: Firewall with Handshake
Example: Firewall with Handshake

Program:
- H₁ may send a REQ to H₄
- Upon receiving REQ, H₄ may respond with an ACK
- Afterwards, H₁ and H₄ may communicate
Example: Distributed Firewall
Example: Distributed Firewall

Program:

- $H_1$ may multicast $\text{REQ}$ to $H_4$ and $H_5$
- After receiving a $\text{REQ}$, at most one of $H_4$ or $H_5$ (e.g., first to $\text{ACK}$) may send packets to $H_1$
Model
Model

- Programs describe how the network reacts to events, modeled as event-transition system or ETS
  - States correspond to network-wide configurations
  - Transitions correspond to network-level events (e.g., receipt of a packet at a particular switch.)
Model

• **Programs** describe how the network **reacts to events**, modeled as **event-transition system** or ETS
  - States correspond to network-wide configurations
  - Transitions correspond to network-level events (e.g., receipt of a packet at a particular switch.)

• **Example:**

  ![Diagram](image_url)
Model

- **Programs** describe how the network *reacts to events*, modeled as *event-transition system* or ETS
  - States correspond to network-wide configurations
  - Transitions correspond to network-level events (e.g., receipt of a packet at a particular switch.)

- **Example:**

- **Challenges:** ensuring that ETSs can be *efficiently implemented* in a way that provides *reasonable consistency guarantees*
Example: Stateful Firewall
Example: Stateful Firewall

ETS: C₁ → e → C₂

- C₁: H₁ may send traffic to H₄
- C₂: H₁ and H₄ may send traffic to each other
- e: packet from H₁ to H₄ at S₄
Example: Firewall with Handshake
Example: Firewall with Handshake

ETS:

- $C_1$: $H_1$ may send $REQ$
- $C_2$: $H_1$ may send $REQ$, $H_4$ may send $ACK$
- $C_3$: $H_1$ and $H_4$ may send traffic to each other
- $e_1$: $REQ$ packet from $H_1$ at $S_4$
- $e_2$: $ACK$ packet from $H_4$ at $S_1$
Example: Distributed Firewall
Example: Distributed Firewall

ETS:

- $C_1$: $H_1$ may send $\text{REQ}$
- $C_2$: $H_1$ may send $\text{REQ}$, $H_4$ and $H_5$ may send $\text{ACK}$
- $C_{3/4}$: $H_1$ and $H_4$ (or $H_5$) may send traffic to each other
- $e_2$: $\text{REQ}$ packet from $H_1$ at $S_4$
- $e_{3/4}$: $\text{ACK}$ packet from $H_{4/5}$ at $S_1$
Event-Driven Consistent Update

C1 \( e \) C2
How should we implement network-wide updates?
Event-Driven Consistent Update

How should we implement network-wide updates?

**Intuition:** per-packet + causal consistency!
Event-Driven Consistent Update

How should we implement network-wide updates?

**Intuition:** per-packet + causal consistency!

- All packets processed using either $C_1$ or $C_2$
How should we implement network-wide updates?

**Intuition:** per-packet + causal consistency!

- All packets processed using either $C_1$ or $C_2$
- If **all** of the switches involved in processing a packet have "heard about" $e$, then process using $C_2$
Event-Driven Consistent Update

How should we implement network-wide updates?

**Intuition:** per-packet + causal consistency!

- All packets processed using either $C_1$ or $C_2$
- If **all** of the switches involved in processing a packet have "heard about" $e$, then process using $C_2$
- If **none** of the switches involved in processing a packet have "heard about" $e$, then process using $C_1$
Conflicting Events
Conflicting Events

ETS:

C1 -> C2 -> C3

C1 -> C2 -> C4
Conflicting Events

ETS:

C1 → C2
  |   e2
  |   ↓
  |   ↓
e3  |   C3
  |   ↓
e4  |   C4
Conflicting Events
Conflicting Events

ETS:

Problem: switches $S_4$ and $S_5$ must somehow determine which of the ACKs was received first...

...but it is unclear how to do this efficiently!
Locality Condition: Events that are simultaneously enabled and "incompatible" with each other must occur at the same switch!

Intuition: rules out "action at a distance"

Inspired by Event Structures [Winskel]

Problem: switches $S_4$ and $S_5$ must somehow determine which of the ACKs was received first...

...but it is unclear how to do this efficiently!
Theorem: an ETS can be implemented efficiently if and only if it satisfies the locality condition.
Theorem: an ETS can be implemented efficiently if and only if it satisfies the locality condition.

Definition: "efficiently" means that switches are guaranteed to process each packet within a fixed time bound—i.e., no excessive packet buffering!
Main Result

**Theorem:** an ETS can be implemented efficiently if and only if it satisfies the locality condition.

**Definition:** "efficiently" means that switches are guaranteed to process each packet within a fixed time bound—i.e., no excessive packet buffering!

**Proof Sketch:**
- Inspired by the proof of the CAP Theorem
- In the absence of locality condition, there exist programs for which the switches would have to buffer arbitrarily many packets
Digest-Carrying Implementation

Overview:

- States of the ETS encoded as sets of events
- Switches maintain a configuration for each state
- Packets stamped with a set of states on ingress
- Packets propagate "digests" of events they've seen as they are forwarded through the network
Case Study: Stateful Firewall

ETS

Naive

ETS

Time (s)
Case Study: Learning Switch

Naive

ETS
Case Study: Learning Switch

ETS

Naive

ETS

ETS
Ongoing Work
Idea: given a set of example packet traces, synthesize and ETS—i.e., the concurrency control for the network.
Learning From Examples

Idea: given a set of example packet traces, synthesize and ETS—i.e., the concurrency control for the network

Positive examples:

H₁ S₁ S₄ H₄
H₁ S₁ S₄ H₄ S₄ S₁ H₁ H₄
H₁ S₁ S₄ H₄ S₄ S₁ H₄ H₁
Learning From Examples

Idea: given a set of example packet traces, synthesize and ETS—i.e., the concurrency control for the network

Positive examples:
H₁ S₁ S₄ H₄
H₁ S₁ S₄ H₄ S₄ S₁ H₁ H₄
H₁ S₁ S₄ H₄ S₄ S₁ H₄ H₁

Negative examples:
H₄ S₄ S₁ H₁
H₁ S₁ H₄ S₄ S₁ S₄ H₄ H₁
Learning From Examples

Idea: given a set of example packet traces, synthesize and ETS—i.e., the concurrency control for the network

Positive examples:
- $H_1 \ s_1 \ s_4 \ H_4$
- $H_1 \ s_1 \ s_4 \ H_4 \ s_4 \ s_1 \ H_1 \ H_4$
- $H_1 \ s_1 \ s_4 \ H_4 \ s_4 \ s_1 \ H_4 \ H_1$

Negative examples:
- $H_4 \ s_4 \ s_1 \ H_1$
- $H_1 \ s_1 \ H_4 \ s_4 \ s_1 \ s_4 \ H_4 \ H_1$
Idea: implement a P4 interpreter for symbolic
configurations encoded as FDDs [ICFP '15]
Idea: implement a P4 interpreter for symbolic configurations encoded as FDDs [ICFP '15]
Idea: implement a P4 interpreter for symbolic configurations encoded as FDDs [ICFP '15]
Reading


Related Work


Cantor Meets Scott:
Semantic Foundations for Probabilistic Networks

Steffen Smolka

Software Research Lunch
12-1pm
Friday, December 9th
Gates 463a