Reasoning about Software Defined Networks

Shachar Itzhaky & Mooly Sagiv
msagiv@acm.org
03-640-7606
Tel Aviv University
Thursday 16-18 (Physics 105)
Monday 14-16 Schrieber 317
Adviser: Michael Shapira
Hebrew University

http://www.cs.tau.ac.il/~msagiv/courses/rsdn.html

Some slides from J. Rexford POPL’12 invited talk
Some Lessons from Michael’s talk

• Some tasks are more suitable for distribution
  – Frequently Executed
  – Simple
  – Regular
  – Can be implemented in hardware
  – Data is distributed

• Some tasks can be executed sequentially without effecting scalability
  – Complicated
  – Rarely executed
  – Increase the effectiveness of the distributed process

• The SDN provides an interesting compromise
• OpenFlow is a reasonable realization
Content

• Challenges in SDNs
• Programming Language Abstractions
• Programming Language Principles
• Program Language Tools
• Other useful tools
Challenges in SDN

• Programming complexity
• Modularity
• Composing operations
• Handling Updates
• Reliability
• Performance
• Testing
• Productivity
• Non-expert programmers
How hard is it to program networks

• Routers with 20+ million lines of code
• Cascading failures, vulnerabilities, etc
• Low level programming
• No abstractions
  – Virtual memory
  – Operating Systems
  – Resource Allocations
  – Libraries
  – Types
Modularity: Simple Repeater

When a switch joins the network, install two forwarding rules.

def repeater(switch):
    # Repeat Port 1 to Port 2
    pat1 = {in_port:1}
    act1 = [forward(2)]
    install(switch, pat1, DEFAULT, act1)

    # Repeat Port 2 to Port 1
    pat2 = {in_port:2}
    act2 = [forward(1)]
    install(switch, pat2, DEFAULT, act2)
Monitor Web ("port 80") traffic

```python
def web_monitor(switch):
    # Web traffic from Internet
    pat = {inport:2,tp_src:80}
    install(switch, pat, DEFAULT, [])
    query_stats(switch, pat)

def stats_in(switch, pat, bytes, …):
    print bytes
    sleep(30)
    query_stats(switch, pat)
```

When a switch joins the network, install one monitoring rule.
def switch_join(switch):
    pat1 = {inport:1}
    pat2 = {inport:2}
    pat2web = {in_port:2, tp_src:80}
    install(switch, pat1, DEFAULT, None, [forward(2)])
    install(switch, pat2web, HIGH, None, [forward(1)])
    install(switch, pat2, DEFAULT, None, [forward(1)])
    query_stats(switch, pat2web)

def stats_in(switch, xid, pattern, packets, bytes):
    print bytes
    sleep(30)
    query_stats(switch, pattern)

Must think about both tasks at the same time.
Concurrency: Switch-Controller Delays

• Common programming idiom
  – First packet goes to the controller
  – Controller installs rules
Concurrency: Switch-Controller Delays

- More packets arrive before rules installed?
  - Multiple packets reach the controller
Concurrency: Switch-Controller Delays

• Rules along a path installed out of order?
  – Packets reach a switch before the rules do

Must think about all possible packet and event orderings.
Debugging is hard [Kazemian NSDI’13]

- Distributed switch tables
- Varied delays
- Cascading effects on switch tables
- Multiple protocols
- Human interaction
- ...

SDN Performance

• Given an SDN controller code

• Estimate the cost of routing per packet
  – Length of the path
  – Weight of the path
  – Number of controller interventions
Programming Language Abstractions

• Hide the implementation
• Interpreter/Compiler guarantees performance
• Benefits
  – Portability
  – Ease of use
  – Reliability
  – Compositionality
  – Productivity
Language Abstractions

- Procedures
- Data Types
  - Enumerated types
  - Arrays
- Abstract data types
- Classes and Objects
- Resources
  - Memory management
  - Garbage collection
- Control abstractions
  - Exceptions
  - Iterators
  - Tree Pattern Matching
  - Higher order functions
  - Continuation
  - Lazy evaluation
  - Monads
- Libraries
- Domain specific languages
- Declarative programming
Programming Language Principles

• Rigorously define the meaning of programs
• Supports abstraction
• Prove compiler correctness
• Prove that two SDN programs are observationally equivalent
  – The same packet transmissions
A Simple Example

• Numeric Expressions
  – `<exp> ::= <exp> + <exp> | < exp> * <exp> | number`
• The semantic of expressions is a number
  \( E[\cdot] : <exp> \rightarrow int \)
• Inductively defined
  – \( E[n] = n \)
  – \( E[e_1 + e_2] = E[e_1] + E[e_2] \)
  – \( E[e_1 \times e_2] = E[e_1] \times E[e_2] \)
• Compositional
• Fully abstract
• \( e_1 \approx e_2 \) iff \( E[e_1] = E[e_2] \)
• \( 5 + 6 \approx 7 + 3 + 1 \)
A Simple Example + Var

• Numeric Expressions
  – \(<exp> ::= <exp> + <exp> | <exp> * <exp> | \text{number} | \text{id}\)
  – \(<com> ::= \text{skip} | \text{id} := \text{exp} | <com> ; <com>\)

• Need to introduce states
  \(\text{Var} \rightarrow \text{Int}\)

• Examples
  – \([x \mapsto 2, y \mapsto 3] x =\)
  – \([x \mapsto 2, y \mapsto 3] z =\)
  – \([x \mapsto 2, y \mapsto 3][z \mapsto 5] = [x \mapsto 2, y \mapsto 3,z \mapsto 5]\)
  – \([x \mapsto 2, y \mapsto 3][x \mapsto 5] = [x \mapsto 5, y \mapsto 3]\)
  – \([x \mapsto 2, y \mapsto 3][x \mapsto 5]x = 5\)
  – \([x \mapsto 2, y \mapsto 3][x \mapsto 5]y = 2\)
A Simple Example + Var

• Numeric Expressions
  – \( <\text{exp}> ::= <\text{exp}> + <\text{exp}> | <\text{exp}> * <\text{exp}> | \text{number} | \text{id} \)

• The semantic of expressions is a function from states to numbers
  \( E[\cdot] : <\text{exp}> \rightarrow (\text{Var} \rightarrow \text{Int}) \rightarrow \text{Int} \)

• Inductively defined
  – \( E[n] \sigma = n \)
  – \( E[e_1 + e_2] \sigma = E[e_1] \sigma + E[e_2] \sigma \)
  – \( E[e_1 * e_2] \sigma = E[e_1] \sigma * E[e_2] \sigma \)
  – \( E[\text{id}] \sigma = \sigma \text{id} \)
  – \( E[x+2][x \mapsto 5] = E[x][x \mapsto 5] + E[2][x \mapsto 5] = [x \mapsto 5]x + 2 = 5 + 2 = 7 \)
  – \( x + x \approx 2 * x \)
  – \( e_1 + e_2 \approx e_2 + e_1 \)
The semantics of commands

• Commands
  – `<com> ::= skip | id := exp | <com> ; <com> | id`

• The semantic of command is a function from states to states
  \[ C[]: \text{<com>} \rightarrow (\text{Var} \rightarrow \text{Int}) \rightarrow (\text{Var} \rightarrow \text{Int}) \]

• Inductively defined
  – \[ C[\text{skip}]\sigma = \sigma \]
  – \[ C[\text{id} := e]\sigma = \sigma[\text{id} \mapsto E[e]\sigma] \]
  – \[ C[c_1 ; c_2]\sigma = C[c_2] (C[c_1]\sigma) \]
  – \[ c ; \text{skip} \approx \text{skip} ; c \approx c \]
  – \[ c_1 ; (c_2 ; c_3) \approx (c_1 ; c_2) ; c_3 \]
Benefits of formal semantics

- Can be done for arbitrary languages
- Automatically generate interpreters
- Rational language design
- Formal proofs of language properties
  - Observational equivalence
  - Type safety
- Correctness of tools
  - Interpreter
  - Compiler
  - Static Analyzer
  - Model Checker
A Simple Controller Language

- No packet changes
- No priorities
- No statistics
- Simple updates
  - Per-flow installation
A Simple Controller Language

- `<controller> ::= <reld>* <evnt>`
- `<reld> ::= rel <rid> (<tid>*) // auxiliary relations`
- `<evnt> ::= packetIn (sid, pid, <pr>) ⇒ <com>`
- `<pr> ::= port(int)`
- `<com> ::= skip
  | send(<pr>)
  | install (sid, pid, <pr>, <opr>) // update flow table
  | rid.insert(<exp>*) // update auxiliary relations
  | if rid(<exp>*) then <com>* else <com>*
  | if rid(<exp>*) then <com>*
  | <com> ; <com>`
- `<opr> ::= <pr> | none // drop the package`
- `<exp> ::= id | exp. id | <pr>`
Learning Switch with 3 ports

rel connected (Sw, Pr, Ho)
PacketIn(s, p, port(1)) ->
  connected.insert (s, port(1), p.src)
  if connected(s, port(2), p.dst) then {
    send (port(2))
    install(s, p, port(1), port(2))
  }
else if  connected(s, port(3), p.dst) then {
  send (port(3))
  install(s, p, port(1), port(3))
}
else  { // flood
  send (port(2))
  send (port(3))
}
PacketIn(s, p, port(2)) -> ...
PacketIn(s, p, port(3)) -> ...
The semantics of the controller

• What is the meaning of package sent by a given host?
  – A set of paths
• Example: Learning Switch
The semantics of the controller

• What is the meaning of package sent by a given host?
  – A set of paths
• Dynamically changed
  – Even for fixed topology
• Unbounded length
• But can it be defined?
The State of a switch

• The switch table is a relation between packets input and output ports

\[ FT = \text{Sw} \rightarrow P(\text{Pk} \times \text{Pr} \times \text{Pr} \cup \{\text{none}\}) \]

• Updated by the controller

• Actual nodes depend on the topology graph
The Semantics of Expressions

- \(<\text{exp}> ::= \text{id} \mid \text{exp}.\text{id} \mid \text{<pr>}\)
- \(\text{Val = Int} \cup \text{Pr} \cup \{P(T[t_1] \times \ldots T[t_k]) \mid t_1, \ldots, t_k \text{ are valid types}\}\)
- Controller State \(\sigma \in \Sigma\) such that \(\sigma: \text{Var} \rightarrow \text{Val}\)
- \(E[\text{id}]\ \sigma = \sigma \text{id}\)
- \(E[\text{exp}.\text{id}]\ \sigma = \text{F[<id>} (E[e] \ \sigma)\)
- \(E[\text{port}(n)]\ \sigma = n\)
The Meaning of Commands

- `<com> ::= skip`  
  `| send(<pr>)`  
  `| install (sid, pid, <pr>, <opr>)`  
  `| rid.insert(<exp>*)`  
  `| if RID(<exp>*) then <com>* else <com>*`  
  `| if RID(<exp>*) then <com>*`  
  `| <com> ; <com>`

- `<opr> ::= <pr> | none`

- `C[]: <com> → (Σ × FT) → (Σ × FT × P(Pr ∪{none}))`
  
  input-state output
Atomic Commands

- $C[\text{skip}] \langle \sigma, ft \rangle = \langle \sigma, ft, \emptyset \rangle$
- $C[\text{send}(pr)] \langle \sigma, ft \rangle = \langle \sigma, ft, \{E[pr]\} \rangle$
- $C[\text{install}(sw, pk, pr_{in}, pr_{out})] \langle \sigma, ft \rangle =$
  \[
  \langle \sigma, ft[sw \mapsto ft(sw) \cup \{\langle pk, E[pr_{in}], E[pr_{out}] \rangle\}], \emptyset \rangle
  \]
- $C[r.\text{insert}(e)] \langle \sigma, ft \rangle =$
  \[
  \langle \sigma[r \mapsto \sigma r \cup \{\langle E[e] \rangle\}], ft, \emptyset \rangle
  \]
Conditional Commands

\[ C[\text{if } r(e) \text{ then } c_1 \text{ else } c_2] \langle \sigma, ft \rangle = \]

\[
\begin{cases}
  C[c_1] \langle \sigma, ft \rangle & \text{if } \langle E[e] \rangle \in \sigma r \\
  C[c_2] \langle \sigma, ft \rangle & \text{if } \langle E[e] \rangle \notin \sigma r
\end{cases}
\]
Sequential Composition

- $C[c_1 ; c_2 ] \langle \sigma, ft \rangle =$
  
  let $\langle \sigma', ft', \text{prs}_1 \rangle = C[c_1 ] \langle \sigma, ft \rangle$
  
  $\langle \sigma'', ft'', \text{prs}_2 \rangle = C[c_2 ] \langle \sigma', ft' \rangle$
  
  in
  
  $\langle \sigma'', ft'', \text{prs}_1 \cup \text{prs}_2 \rangle$
Semantics of Events

- $G[\llbracket evnt \rrbracket] : \langle Sw \times Pk \times Pr \rangle \rightarrow P(\langle \Sigma \times FT \rangle \times (\Sigma \times FT \times P(Pr \cup \{\text{none}\})))$

- $G[\llbracket \text{packetIn}(s, p, i) \Rightarrow c \rrbracket \langle sw, pk, ip \rangle = R$
  where
  $\langle \sigma, ft \rangle R \langle \sigma', ft', \text{ports} \rangle$
  Iff
  $\neg \exists \text{op} : \text{Pr}. \langle pk, ip, op \rangle \in ft)$ $\land$
  $C[\llbracket c \rrbracket] \langle \sigma[s\rightarrow sw, p\rightarrow pk, i\rightarrow ip], ft \rangle = \langle \sigma', ft', sp \rangle$
Packet Histories

• Packets can influence the routing of other packets
• Histories can be seen as sequences of packet transmissions
• $\text{HST}= (Sw \times Pr \times Pk \times (Pr \cup \{\text{none}\}))^*$
Small-step semantics (Controller)

- $\Rightarrow_{ctrl} : (\Sigma \times FT \times HST) \times (\Sigma \times FT \times HST)$

- $\langle \sigma, ft, h \rangle \Rightarrow_{ctrl} \langle \sigma', ft', h' \rangle$ iff

  $\exists sw : Sw, pk : Pk, ip : Pr, sp : P(Pr)$. 

  $\langle \sigma, ft, sw \rangle (G[evnt] \langle sw, pk, ip \rangle) \langle \sigma', ft', sp \rangle \wedge h' = h \cdot \langle sw, ip, pk, j \rangle_j \in sp \lor (sp = \emptyset \wedge j=\text{none})$
Small-step Semantics (Switch)

- \( \Rightarrow_{sw} : (\Sigma \times FT \times HST) \times (\Sigma \times FT \times HST) \)

- \( \langle \sigma, ft, h \rangle \Rightarrow_{\text{switch}} \langle \sigma', ft', h' \rangle \) iff
  \[
  \sigma = \sigma' \land ft = ft' \land \\
  \exists sw : Sw, pk : Pk, ip : Pr, op : Pr \cup \{\text{none}\}. \\
  \langle pk, ip, op \rangle \in ft \sw \\
  h' = h \cdot \langle sw, ip, pk, op \rangle
  \]
Executions

• Combined step semantics:
  \[ \implies = \implies \text{ctrl} \cup \implies \text{switch} \]

• A sequence of event processing steps has the from
  \[ a_1 \implies a_2 \implies \ldots \implies a_n \]
Feasible Histories

• Given a topology graph $G$ over $V = H_0 \cup (Sw \times Pr)$

• A history $h$ is feasible w.r.t. a given topology graph $G$ iff for any packet $pk \in P_k$:
  – any two consecutive entries for $p$ in $h$:
    • $\langle sw_1, ip_1, pk, op_1 \rangle$ and $\langle sw_2, ip_2, pk, op_2 \rangle$
    • there exists an edge between $\langle sw_1, op_1 \rangle$ and $\langle sw_2, ip_2 \rangle$ in $G$
In
Packet
Out

Port in | Packet | Port out
-------|--------|--------

connected =

<table>
<thead>
<tr>
<th>Port</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Port in</th>
<th>Packet</th>
<th>Port out</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>β→α</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>β→α</td>
<td>3</td>
</tr>
</tbody>
</table>

connected =

<table>
<thead>
<tr>
<th>Port</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>β</td>
</tr>
</tbody>
</table>
Example

Port in | Packet | Port out
--- | --- | ---
2 | $\beta \rightarrow \alpha$ | 1
2 | $\beta \rightarrow \alpha$ | 3
3 | $\gamma \rightarrow \beta$ | 2

connected =

<table>
<thead>
<tr>
<th>Port</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$\beta$</td>
</tr>
<tr>
<td>3</td>
<td>$\gamma$</td>
</tr>
</tbody>
</table>
Example

Port in | Packet | Port out
--- | --- | ---
2 | $\beta \rightarrow \alpha$ | 1
2 | $\beta \rightarrow \alpha$ | 3
3 | $\gamma \rightarrow \beta$ | 2
3 | $\gamma \rightarrow \beta$ | 2

connected =

<table>
<thead>
<tr>
<th>Port</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$\beta$</td>
</tr>
<tr>
<td>3</td>
<td>$\gamma$</td>
</tr>
</tbody>
</table>
Properties of the Semantics

• Compositional
• Allows free packets
• Assume that (controller) actions executed atomically
  – Ignores delays in switch rule instantiations
Useful Programming Language Tools

• Interpreter
• Compiler
• Parser
• Type Checker
• Static Program Analysis
• Verification Tool
• Dynamic Program Analysis
  – Model checker
Verification Process

- Topology $T$
- Controller Code
- Required properties $\varphi$

Verification Condition

$T \land [P] \Rightarrow \varphi$

- VC gen
- SAT Solver

Counterexample

Proof
## Interesting Network Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectivity</td>
<td>Every packet eventually reaches its destination</td>
</tr>
<tr>
<td>No forwarding loops</td>
<td>A switch sw never receives a packet sent by sw</td>
</tr>
<tr>
<td>No black holes</td>
<td>No packet should be dropped in the network</td>
</tr>
<tr>
<td>Access control</td>
<td>No paths between certain hosts of certain packets</td>
</tr>
<tr>
<td>Direct paths</td>
<td>Once a packet reached its destination future packets are not going to the controller</td>
</tr>
<tr>
<td>Strict direct paths</td>
<td>Once two packets travel both ways between a source and destination</td>
</tr>
<tr>
<td>Data structure integrity</td>
<td>The controller data structures <code>correctly</code> records the network states</td>
</tr>
</tbody>
</table>