COCOON: CORRECT-BY-CONSTRUCTION NETWORKS USING STEPWISE REFINEMENT

Leonid Ryzhyk, Nikolaj Bjorner, Marco Canini, Jean-Baptiste Jeannin
Cole Schlesinger, Douglas Terry, George Varghese
RUNNING EXAMPLE: CAMPUS NETWORK

subnet 1
subnet 2
ACL
core
zone 1
zone 2
zone 3
RUNNING EXAMPLE: CAMPUS NETWORK

subnet 1 gateway router
subnet 2 gateway router
subnet 1
subnet 2
switch
router (not assigned to a subnet)

zone 1
zone 2
zone 3
core
NETWORK VERIFICATION: CURRENT PRACTICES

**Option 1: Dataplane verification**
(NetPlumber, HSA, Veriflow)
- Fixing bugs in a deployed network takes time; may not avoid the downtime

Check for:
- ✔ Loop freedom
- ✔ Black holes
- ✔ Reachability
- ✔ Isolation

**Option 2: Controller verification**
(Vericon, FlowLog)
- ✔ Limited scalability

Common to both approaches:
- ✔ Property-based verification does not guarantee correctness
Option 1: Dataplane verification (NetKAT)
- Limited scalability
REQUIREMENTS

Ideally, network verification should be:

1. Scalable
   (works at DC scale)

2. Static
   (verifies all possible configurations)

3. Exhaustive
   (misses no bugs)

➤ State of the art: pick 1 out of 3
OBSERVATIONS

- Simple top-level description: the *what*, not the *how*
- Design by hierarchical decomposition
DECOMPOSING A WAN

Data center 1

Data center 2

WAN router

WAN link

Local fabric

Global fabric

ToR layer

Core layer

Local fabric

DC3
MORE EXAMPLES

- Virtual network is decomposed into
  - Physical fabric
  - Virtual fabric
- Cellular network is decomposed into
  - Edge (base stations)
  - Core
  - Internet gateway

Exposing this structure enables efficient compositional verification
We propose Cocoon:

- SDN design method
- Programming language
- Verifier

Cocoon achieves *scalable, static, exhaustive* verification (3 out of 3!) via a network design process that focuses on correctness.
EXAMPLE COCOON SPECIFICATIONS

```
role HostOut[IP4 addr] | cHost(addr) = 
  filter ip2subnet(pkt.srcIP)==ip2subnet(pkt.dstIP)
  or acl(pkt);
  filter cHost(pkt.dstIP);
  send HostIn[pkt.dstIP]
```

Runtime-Defined Functions:

- function cHost(IP4 addr)
- function cSubnet(int vid)
- function acl(Packet p)
- function ip2subnet(IP4 ip)

Assumption:

```
∀ addr. cHost(addr) → cSubnet(ip2subnet(addr))
assume(IP4 addr) cHost(addr)=>cSubnet(ip2subnet(addr))
```
**REFINEMENT EXAMPLE**

```
role HostOut[IP4 addr] | cHost(addr) =
  filter ip2subnet(pkt.srcIP) == ip2subnet(pkt.dstIP)
  or acl(pkt);
  filter cHost(pkt.dstIP);
  send HostIn[pkt.dstIP]

refine HostOut {
  role HostOut[IP4 addr] | cHost(addr) =

  ... 
  ... 
  send RouterZoneIn[zone(addr)]

  role RouterZoneIn[zid_t] =
  ...
  ...
}
```
2-PHASE VERIFICATION

- Refinements + assumptions specify static network design
  - Verified statically

- RDFs encapsulate runtime configuration
  - Checked at runtime against assumptions
COCOON ARCHITECTURE

Output: A diagram illustrating the COCOON architecture, with components labeled as follows:
- **Cocoon spec**
- **Verifier**
- **Assumption checker**
- **Compiler**
- **Cocoon runtime**
- **OpenFlow/P4**
- **SDN controller**
- **External apps**
- **RDF definitions**

The diagram shows the flow of information and interaction between these components, highlighting how external apps interact with the Cocoon runtime, which in turn interfaces with SDN controllers through OpenFlow/P4 protocols.
IMPLEMENTING VERIFICATION

- Role semantics: $R : LPkt \rightarrow 2^{LPkt}$
- Role refinement: $\hat{R}(pkt) \subseteq R(pkt)$
- We convert this program to Boogie and use the Corral model checker
  - Enforce static bound on the number of network hops to achieve completeness
- Assumptions are converted to SMT and checked using Z3
CASE STUDIES

- **B4-style WAN**
  [Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN]

- **NSX-style network virtualization framework**
  [Koponen et al. Network Virtualization in Multi-tenant Datacenters]

- **Enterprise network**
  [Sung et al. Towards Systematic Design of Enterprise Networks]

- **F10**

- **Stag**
  [Lopes et al. Automatically verifying reachability and well-formedness in P4 Networks]

- **iSDX**
  [Gupta et al. An Industrial-Scale Software Defined Internet Exchange Point]
# PERFORMANCE (static verification)

<table>
<thead>
<tr>
<th>case study</th>
<th>LOC total</th>
<th>LOC high-level</th>
<th>#refines</th>
<th>verification time (s)</th>
<th>verification time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compositional</td>
<td>monolithic</td>
</tr>
<tr>
<td>WAN virtualization</td>
<td>305</td>
<td>18</td>
<td>6</td>
<td>10</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>enterprise</td>
<td>678</td>
<td>97</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>F10</td>
<td>342</td>
<td>50</td>
<td>4</td>
<td>16</td>
<td>&gt;3600</td>
</tr>
<tr>
<td>stag</td>
<td>283</td>
<td>47</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>iSDX</td>
<td>190</td>
<td>21</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**Compositional:**

- [✔️] [✔️] [✔️] [✔️] [✔️]

**Monolithic:**

- [✔️]
PERFORMANCE (runtime verification)

![Graph showing performance over network scale](image)

<table>
<thead>
<tr>
<th>Scale</th>
<th>Hosts</th>
<th>Switches</th>
<th>NetKAT Policy</th>
<th>Flowtable Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>11</td>
<td>1,559</td>
<td>830</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>23</td>
<td>5,149</td>
<td>3,299</td>
</tr>
<tr>
<td>15</td>
<td>47</td>
<td>63</td>
<td>46,094</td>
<td>31,462</td>
</tr>
<tr>
<td>25</td>
<td>77</td>
<td>103</td>
<td>151,014</td>
<td>89,216</td>
</tr>
<tr>
<td>40</td>
<td>122</td>
<td>163</td>
<td>496,268</td>
<td>212,925</td>
</tr>
</tbody>
</table>
COCOON VS TRADITIONAL NETWORK

VERIFICATION

HSA/Veriflow/

Correctness spec
PERFORMANCE (Cocoon + HSA)

![Graph showing performance over network scale.

<table>
<thead>
<tr>
<th>Scale</th>
<th>Hosts</th>
<th>Switches</th>
<th>NetKAT Policy</th>
<th>Flowtable Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>11</td>
<td>1,559</td>
<td>830</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>23</td>
<td>5,149</td>
<td>3,299</td>
</tr>
<tr>
<td>15</td>
<td>47</td>
<td>63</td>
<td>46,094</td>
<td>31,462</td>
</tr>
<tr>
<td>25</td>
<td>77</td>
<td>103</td>
<td>151,014</td>
<td>89,216</td>
</tr>
<tr>
<td>40</td>
<td>122</td>
<td>163</td>
<td>496,268</td>
<td>212,925</td>
</tr>
</tbody>
</table>
CONCLUSION

- Design-by-refinement works well for networks:
  - Allow concise high-level specifications
  - Well-defined module boundaries
  - Verification is feasible for a single refinement: no pointers, concurrency, dynamic memory allocation, etc.

Source code, case studies:

https://github.com/ryzhyk/cocoon