Dynamic Test Packet Generation for P4 Programs
(Work in Progress)

Praveen Kumar ¹  Hossein Hojjat ¹  Changhoon Kim ²  Nate Foster ¹

¹Department of Computer Science, Cornell University
²Barefoot Networks

November 23, 2015
P4: Programming Protocol-Independent Packet Processors
A high-level language for programming protocol-independent packet processors
Traditional Networking

End-To-End Arguments in System Design

J. H. SALTZER, D. P. REED, and D. D. CLARK
Massachusetts Institute of Technology Laboratory for Computer Science

This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit-error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgment. Low-level mechanisms to support these functions are justified only as performance enhancements.

CR Categories and Subject Descriptors: C.0 [General] Computer System Organization—system architectures; C.2.2 [Computer-Communication Networks]: Network Protocols—protocol architecture; C.2.4 [Computer-Communication Networks]: Distributed Systems; D.4.7 [Operating Systems]: Organization and Design—distributed systems

General Terms: Design

Additional Key Words and Phrases: Data communication, protocol design, design principles

1. INTRODUCTION

Choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer. Design principles that provide guidance in this choice of function placement are among the most important tools of a system designer. This paper discusses one class of function placement argument that has been used for many years with neither explicit recognition nor much conviction. However, the emergence of the data communication network as a computer system component has sharpened this line of function placement argument by making more apparent the situations in which and the reasons why it applies. This paper articulates the argument explicitly, so as to examine its nature and to see how general it really is. The argument appeals to application requirements and provides a rationale for moving a function upward in a layered system closer to the application that uses the function. We begin by considering the communication network version of the argument.

This is a revised version of a paper adapted from End-To-End Arguments in System Design by J. H. Saltzer, D. P. Reed, and D. D. Clark from the 2nd International Conference on Distributed Systems (Paris, France, April 8–10 1981, pp. 509–512. © IEEE 1981

This research was supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense and monitored by the Office of Naval Research under contract N00014-75-C-0661.


Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0734-207X/84/1100-0277 $00.75


- Classic design principle in network: End-to-End principle
- Dumb minimal network with smart endpoints
New Cisco ASIC has a programmable data plane

The new Cisco ASIC, Unified Access Data Plane, is a programmable chip that will accelerate product

Programmable Parsing
The Intel® Ethernet FM6000 series switch provides unprecedented parsing flexibility to support the ever growing portfolio of networking protocols, making the switch an ideal building block for administrators that want to increase productivity and introduce the latest advances and standards. The switch is a programmable data plane to support line rate performance.
Emerging switch architectures are programmable
P4 design goal: provide a vendor agnostic, standardized and unified way to program a switch

Programmable switch does not know how to process packets (Protocol Independence)

P4 program configures the switch to know how to process packets
P4 design goal: provide a vendor agnostic, standardized and unified way to program a switch

Programmable switch does not know how to process packets (Protocol Independence)

P4 program configures the switch to know how to process packets

A P4 program can have **internal mutable state**

Forwarding a packet can have side effects to local state that can potentially affect the forwarding of the next packets
Stateful Firewall

- $H_1$ should be allowed to communicate with $H_2$
- $H_2$ should only be allowed to communicate with $H_1$ if $H_1$ has previously initiated a connection
Stateful Firewall

- $H_1$ should be allowed to communicate with $H_2$
- $H_2$ should only be allowed to communicate with $H_1$ if $H_1$ has previously initiated a connection
Stateful Firewall

- $H_1$ should be allowed to communicate with $H_2$
- $H_2$ should only be allowed to communicate with $H_1$ if $H_1$ has previously initiated a connection

$f_{H_1 \rightarrow H_2} = \text{false}$
- $H_1$ should be allowed to communicate with $H_2$
- $H_2$ should only be allowed to communicate with $H_1$ if $H_1$ has previously initiated a connection
Stateful Firewall

- $H_1$ should be allowed to communicate with $H_2$
- $H_2$ should only be allowed to communicate with $H_1$ if $H_1$ has previously initiated a connection
Stateful Firewall

- $H_1$ should be allowed to communicate with $H_2$
- $H_2$ should only be allowed to communicate with $H_1$ if $H_1$ has previously initiated a connection
Stateful Firewall

- $H_1$ should be allowed to communicate with $H_2$
- $H_2$ should only be allowed to communicate with $H_1$ if $H_1$ has previously initiated a connection
Our Goals

- Generate test packets for P4 programs that cover all entries of forwarding tables
  - Challenge: P4 switches are stateful
- Provide formal semantics for P4 programs by translation to SMT
- Formal verification for P4 programs
  - Proving or disproving a safety property
**Test packet generation:** Produce a minimum number of test packets to exercise every forwarding rule in the network
Test Packets

Test packet generation: Produce a minimum number of test packets to exercise every forwarding rule in the network

Stateless: Model each switch as a transfer function in header space
Stateful Test Packets

- **Stateful packet generation:** Maybe a series of packets are required to test an entry
Stateful packet generation: Maybe a series of packets are required to test an entry
For expressions, the value with largest bit width is identified and all other values are converted to this width according to their own signedness. The expression is then evaluated and the result is converted as necessary according to its use.

- **Table definitions**: the type of lookup to perform, the input fields to use, the actions that may be applied, and the dimensions of each table.
- **Action definitions**: compound actions composed from a set of primitive actions.
- **Pipeline layout and control flow**: the layout of tables within the pipeline and the packet flow through the pipeline.

P4 addresses the configuration of a forwarding element. Once configured, tables may be populated and packet processing takes place. These post-configuration operations are referred to as "run time" in this document. This does not preclude updating a forwarding element's configuration while it is running.

1.1 The P4 Abstract Model

The following diagram shows a high level representation of the P4 abstract model. The P4 machine operates with only a few simple rules.

- Unclear statements: “as necessary”, “according to its use”
- Need a formal semantics for P4
Overall Process

P4 program

Translate

Decision Diagram

Traverse

Set of Paths

Pick a path

Symbolic Execution

Feasible + Model (test)

Infeasible

Pick another path (until coverage)
```p4
count := 0
drop

register count { width: 4 }
control ingress {
    apply (read_count);
    if (count < 8)
        apply (ipv4_policy_A);
    else apply (ipv4_policy_B);
}
control egress {
    apply (write_count);
}
action set_outport (port) {
    modify_field (standard_metadata.egress_spec, port);
    add_to_field (count, 1);
    add_to_field (ipv4.ttl, -1);
}
table ipv4_policy_A {
    reads {ipv4.dstAddr: lpm;}
    actions {
        set_outport;
        drop;
    }
}
table ipv4_policy_B {
    reads {ipv4.dstAddr: lpm;}
    actions {
        set_outport;
        drop;
    }
}
```

### Policy A
<table>
<thead>
<tr>
<th>Addr</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1</td>
</tr>
</tbody>
</table>

### Policy B
<table>
<thead>
<tr>
<th>Addr</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>2</td>
</tr>
</tbody>
</table>

Policy A: 1010 (Port 1)  
Policy B: 1111 (Port 2)
Symbolic Execution: P4 Example

\[
\begin{align*}
\text{count} & := 0 \\
\text{addr} & := \text{read} \\
\text{count} & < 8 \\
\text{addr} = 1010 & \quad \text{addr} = 1111 \\
\text{drop} & \\
\text{port} & := \begin{cases} 1 & \text{if } \text{addr} = 1010 \\ 2 & \text{if } \text{addr} = 1111 \end{cases} \\
\text{count} & := \text{count} + 1 \\
\text{count} & := \text{count} + 1
\end{align*}
\]
Symbolic Execution: P4 Example

- \( \text{count} := 0 \)
- \( \text{addr} := \text{read} \)
- \( \text{count} < 8 \)
- \( \text{addr} = 1010 \)
  - \( T \): \( \text{drop} \)
  - \( F \): \( \text{port} := 1 \)
- \( \text{addr} = 1111 \)
  - \( F \): \( \text{drop} \)
  - \( T \): \( \text{port} := 2 \)
- \( \text{count} := \text{count} + 1 \)

<table>
<thead>
<tr>
<th>count</th>
<th>addr</th>
<th>port</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>
Symbolic Execution: P4 Example

\[
\begin{align*}
count &:= 0 \\
addr &:= \text{read} \\
count &< 8 \\
\begin{array}{l}
addr = 1010 \\
\quad T \\
port &:= 1
\end{array} & \begin{array}{l}
\quad F \\
\quad \text{drop}
\end{array} \\
\begin{array}{l}
addr = 1111 \\
\quad F \\
\quad \text{drop}
\end{array} & \begin{array}{l}
\quad T \\
port &:= 2
\end{array}
\end{align*}
\]

\[
\begin{array}{c|c|c|c}
\text{count} & \text{addr} & \text{port} & \text{PC} \\
\hline
- & - & - & \text{true} \\
0 & - & - & \text{true} \\
0 & x_0 & - & \text{true}
\end{array}
\]

\((x_0 \cdots x_8 \text{ are fresh variable symbols})\)
Symbolic Execution: P4 Example

\[
\begin{align*}
\text{count} &:= 0 \\
\text{addr} &:= \text{read} \\
\text{count} &< 8 \\
\text{addr} = 1010 &\rightarrow \text{drop} \\
\text{addr} = 1111 &\rightarrow \text{port} := 2 \\
\text{port} &:= 1 \\
\text{count} &:= \text{count} + 1 \\
\text{count} &:= \text{count} + 1
\end{align*}
\]

<table>
<thead>
<tr>
<th>count</th>
<th>addr</th>
<th>port</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>—</td>
<td>—</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>$x_0$</td>
<td>—</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>$x_0$</td>
<td>—</td>
<td>(0 &lt; 8)</td>
</tr>
</tbody>
</table>

($x_0 \cdots x_8$ are fresh variable symbols)
Symbolic Execution: P4 Example

\[\text{count} := 0\]
\[\text{addr} := \text{read}\]
\[\text{count} < 8\]
\[\begin{align*}
\text{addr} = 1010 & \quad T \quad \text{drop} \quad F \\
\text{addr} = 1111 & \quad F \quad \text{drop} \quad T \\
\end{align*}\]
\[\text{port} := 1\]
\[\text{port} := 2\]
\[\text{count} := \text{count} + 1\]
\[\text{count} := \text{count} + 1\]

<table>
<thead>
<tr>
<th>count</th>
<th>addr</th>
<th>port</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>(x_0)</td>
<td>-</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>(x_0)</td>
<td>-</td>
<td>(0 &lt; 8)</td>
</tr>
<tr>
<td>0</td>
<td>(x_0)</td>
<td>-</td>
<td>(0 &lt; 8) &amp; (x_0 \neq 1010)</td>
</tr>
</tbody>
</table>

\((x_0 \cdots x_8 \text{ are fresh variable symbols})\)
Symbolic Execution: P4 Example

How can we generate test packets to examine $count \geq 8$?
How can we generate test packets to examine $\text{count} \geq 8$?
Symbolic Execution: P4 Example

How can we generate test packets to examine $\text{count} \geq 8$?
Symbolic Execution: P4 Example

How can we generate test packets to examine $\text{count} \geq 8$?
Symbolic Execution: P4 Example

How can we generate test packets to examine $count \geq 8$?
Symbolic Execution: P4 Example

How can we generate test packets to examine $\text{count} \geq 8$?
Symbolic Execution: P4 Example

How can we generate test packets to examine \( \text{count} \geq 8 \)?
Symbolic Execution: P4 Example

How can we generate test packets to examine $\text{count} \geq 8$?
Symbolic Execution: P4 Example

How can we generate test packets to examine $\text{count} \geq 8$?
Challenges

- **Path explosion**: major bottleneck
Challenges

- **Path explosion**: major bottleneck
- Compositional Symbolic Execution:

\[
\begin{align*}
\text{paths} & = n \times m + n \times r \\
\end{align*}
\]
Challenges

- **Path explosion:** major bottleneck
- **Compositional Symbolic Execution:**
- Compute a table summary and re-use it later

\[
\begin{align*}
\text{paths} &= n \times m + n \times r \\
n &\left\{ \begin{array}{c}
\begin{array}{cc}
\text{pat}_1 & \text{act}_1 \\
\vdots & \vdots \\
\text{pat}_n & \text{act}_n
\end{array}
\end{array} \right. \\
m &\left\{ \begin{array}{cc}
\begin{array}{cc}
\text{pat'}_1 & \text{act'}_1 \\
\vdots & \vdots \\
\text{pat'}_m & \text{act'}_m
\end{array} \\
\begin{array}{cc}
\text{pat''}_1 & \text{act''}_1 \\
\vdots & \vdots \\
\text{pat''}_r & \text{act''}_r
\end{array}
\end{array} \right. \\
r &\left\{ \begin{array}{c}
\begin{array}{cc}
\text{pat'}_1 & \text{act'}_1 \\
\vdots & \vdots \\
\text{pat'}_m & \text{act'}_m
\end{array}
\end{array} \right. \\
\end{align*}
\]
Challenges

Table Summary

\[
\begin{align*}
&\neg pat_1 \land \neg pat_2 \land (v'_1 = v_1 \land v'_2 = v_2) \\
\lor (\neg pat_1 \land pat_2) \land (v'_1 = v_1 \land v'_2 = expr_2) \\
\lor (pat_2) \land (v'_1 = expr_1 \land v'_2 = v_2)
\end{align*}
\]

<table>
<thead>
<tr>
<th>pat_1</th>
<th>act_1 : v_1 := expr_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat_1</td>
<td>act_2 : v_2 := expr_2</td>
</tr>
</tbody>
</table>

Diagram:

- pat_1
- pat_2
- act_1
- act_2
Challenges

- **Path explosion**: major bottleneck
- Compositional Symbolic Execution:
  - Compute a table summary and re-use it later
- **Incremental Solving**: We push and pop the computed summaries

\[
\begin{align*}
\phi_n & \quad (\text{not used}) \\
\begin{array}{c}
pat_1 \quad act_1 \\
\vdots \\
pat_n \quad act_n \\
\end{array} \\
\begin{array}{c}
pat'_{1} \quad act'_{1} \\
\vdots \\
pat'_{m} \quad act'_{m} \\
\end{array} \\
\begin{array}{c}
pat''_{1} \quad act''_{1} \\
\vdots \\
pat''_{r} \quad act''_{r} \\
\end{array}
\end{align*}
\]
## Experiments

<table>
<thead>
<tr>
<th># Rules per table: 64</th>
<th>basic_routing</th>
<th>l2_switch</th>
<th>simple_router</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time (s)</td>
<td>14.8</td>
<td>9.38</td>
<td>36.89</td>
</tr>
<tr>
<td>Z3 time (s)</td>
<td>11.95</td>
<td>7.37</td>
<td>32.57</td>
</tr>
<tr>
<td>#packets</td>
<td>43</td>
<td>193</td>
<td>219</td>
</tr>
<tr>
<td>#entries</td>
<td>390</td>
<td>390</td>
<td>780</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Rules per table: 128</th>
<th>basic_routing</th>
<th>l2_switch</th>
<th>simple_router</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time (s)</td>
<td>55.66</td>
<td>33.29</td>
<td>96.82</td>
</tr>
<tr>
<td>Z3 time (s)</td>
<td>45.49</td>
<td>27.35</td>
<td>81.63</td>
</tr>
<tr>
<td>#packets</td>
<td>64</td>
<td>385</td>
<td>465</td>
</tr>
<tr>
<td>#entries</td>
<td>774</td>
<td>774</td>
<td>1548</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Rules per table: 256</th>
<th>basic_routing</th>
<th>l2_switch</th>
<th>simple_router</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time (s)</td>
<td>217.3</td>
<td>115.76</td>
<td>450.68</td>
</tr>
<tr>
<td>Z3 time (s)</td>
<td>177.34</td>
<td>94.16</td>
<td>389.91</td>
</tr>
<tr>
<td>#packets</td>
<td>88</td>
<td>769</td>
<td>940</td>
</tr>
<tr>
<td>#entries</td>
<td>1542</td>
<td>1542</td>
<td>3084</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Rules per table: 512</th>
<th>basic_routing</th>
<th>l2_switch</th>
<th>simple_router</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time (s)</td>
<td>806.06</td>
<td>611.64</td>
<td>1580.01</td>
</tr>
<tr>
<td>Z3 time (s)</td>
<td>656.4</td>
<td>523.14</td>
<td>1359.82</td>
</tr>
<tr>
<td>#packets</td>
<td>149</td>
<td>1538</td>
<td>1764</td>
</tr>
<tr>
<td>#entries</td>
<td>2942</td>
<td>3078</td>
<td>6011</td>
</tr>
</tbody>
</table>
Experiments

Verification time

Time (s) vs. Number of entries per table

- simple_router
- l2_switch
- basic_routing
Future Work

- Automatic synthesis of forwarding table entries instead of getting as input
- Incorporate different coverage criteria
- P4-specific reduction techniques in addition to general symbolic execution heursitics