# Simon: Scriptable Interactive Monitoring for SDNs

Tim Nelson Da Yu Yiming Li Rodrigo Fonseca Shriram Krishnamurthi
Brown University
{tn, dyu, yli, rfonseca, sk}@cs.brown.edu

## Abstract

Although Software-Defined Networking can simplify network management, it also poses new testing and debugging challenges for operators. Debugging is often an interactive process that involves stepping through data- and control-plane events and performing actions in response. Sometimes, however, this interactive process can become highly repetitive; in such cases, we should be able to script the activity to reduce operator overhead and increase reusability.

We introduce SIMON, a Scriptable Interactive Monitoring system for SDN. With SIMON, operators can probe their network behavior by executing scripts for debugging, monitoring, and more. SIMON is independent of the controller platform used, and does not require annotations or intimate knowledge of the controller software being run. Operators may compose debugging scripts both offline and interactively at SIMON's debugging prompt. In the process, they can take advantage of the rich set of reactive functions SIMON provides as well as the full power of Scala. We present the design of SIMON and discuss its implementation and use.

## Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network management

## General Terms

Management

## Keywords

Software-Defined Networking, OpenFlow, Debugging

## 1. INTRODUCTION

Software-Defined Networking greatly increases the power that operators have over their networks. Unfortunately, this power comes at a cost. While logically centralized controller programs do simplify network control and configuration, the network itself remains an irrevocably distributed system. Bugs in network behavior are not eliminated in an SDN, but are merely lifted into controller programs. SDNs also introduce new classes of bugs that do not exist in traditional networks, such as flow-table consistency issues [16, 19].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*SOSR2015,* June 17 – 18, 2015, Santa Clara, CA USA
Copyright 2015 ACM ISBN 978-1-4503-3451-8/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2774993.2774994 .

Even in a relatively simple environment such as Mininet [11], it can be frustrating to understand SDN controller behavior. Simple errors can be deceptively subtle to test and debug. For instance, if an application sometimes unnecessarily floods traffic via packetOut messages, the network's performance can suffer even though connectivity is preserved. Similarly, Perešíni et al. [16] note a class of bugs that result in a glut of unnecessary rules being installed on switches, without changing the underlying forwarding semantics. Efforts that focus only on connectivity, such as testing via ping, can disguise these and other problems.

Fortunately, SDN also offers opportunities for improving how we test, debug, and verify networks. Invariant checking tools such as VeriFlow [9] and NetPlumber [6] are a good first step. However these tools, while powerful, are limited in scope to the network's forwarding information base, and errors involving more (such as the above unnecessary-flooding error) will escape their notice. Even total knowledge of the flow-table rules installed does not suffice to fully predict network behavior; as Kuźniar, et al. [10] show, different switches have varying foibles when it comes to implementing OpenFlow, with some even occasionally disregarding barrier requests and installing rules in (reordered) batches. Thus, operators need tools that can inspect more than just forwarding tables and can determine whether the network (on all planes) respects their goals.

SIMON (Scriptable Interactive Monitoring) is a next step in that direction. SIMON is an interactive debugger for SDNs; its architecture is shown in Figure 1. SIMON has visibility into data-plane events (e.g., packets arriving at and being forwarded by switches), control-plane events (e.g., OpenFlow protocol messages), northbound API messages (communication between the controller and other services; e.g., see Section 5), and more, limited only by the monitored event sources (Section 4). Since SIMON is interactive, users can use these events to iteratively refine their understanding of the system at SIMON's debugging prompt, similar to using traditional debugging tools. Moreover, SIMON does not presume the user is knowledgeable about the intricacies of the controller in use.

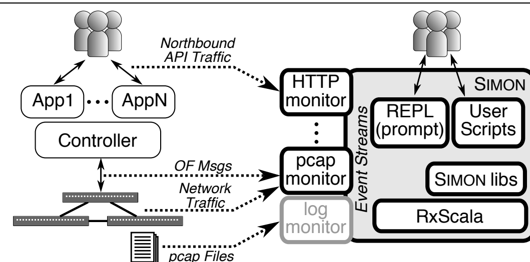


Figure 1: SIMON's workflow. Events are captured from the network by monitors, which feed into Simon's interactive debugging process.

The downside of an interactive debugger is that its use can often be repetitious. SIMON is thus *scriptable*, enabling the automation of repetitive tasks, monitoring of invariants, and much else. A hallmark of SIMON is its reactive scripting language, which is embedded into Scala (`scala-lang.org`). As we show in Section 2, reactivity enables both power and concision in debugging. Furthermore, having recourse to the full power of Scala means that SIMON enables kinds of *stateful* monitoring not supported in many other debuggers. In short, SIMON represents a fundamental shift in how we debug networks, by bringing ideas from software debugging and programming language design to bear on the problem.

# 2. SIMON IN ACTION

For illustrative purposes, we show SIMON in action on an intentionally simplistic example: a small stateful firewall application. (We discuss more complex applications in Section 5.) A stateful firewall should allow all traffic from internal to external ports, but deny traffic arriving at external ports unless it involves hosts that have previously communicated.

## 2.1 Debugging With Events

Consider a basic implementation that performs three separate tasks when an OpenFlow packetIn message is received on an internal port:

1. It installs an OpenFlow rule to forward internally-arriving traffic with this packet's source and destination;
2. It installs an OpenFlow rule forwarding replies between those addresses arriving at the external port; and
3. It replies to the original packetIn with a packetOut message so that this packet will be properly forwarded.

If the programmer forgets (3), packetOuts are never sent, and packets that arrive before FlowMod installation will be dropped. Since the OpenFlow rules are installed correctly, this bug will not be caught by flow-table invariant checkers like VeriFlow. Connections eventually work, but the bug is noticeable when pinging external hosts. Faced with this issue, an operator may ask: *What happened to the initial ping?* To investigate, we first enter the following at SIMON's prompt:

```
val ICMPStream=Simon.nwEvents().filter(isICMP);
showEvents(ICMPStream);
```

The first line reactively *filters* SIMON's stream of network events (`Simon.nwEvents()`) to remove everything but ICMP packet arrivals and departures. All streams produced are constantly updated by SIMON to maintain consistency, and so new ICMP packet arrivals will automatically be directed to `ICMPStream`. We might also achieve this effect with callbacks (Section 3), but at the cost of far more verbose code where we most want concision: an interactive prompt. Although the `filter` operation here is analogous to a pcap filter, we will show that SIMON provides far more flexibility.

In the second line, the `showEvents` function spawns a new window that prints events arriving on the stream it is given, allowing further study without cluttering the prompt. The window displays the following[1] when `h1` pings `h2` twice on a linear, 1-switch, 2-host topology with `h1` and `eth1` internal:

```
ICMP packet from h1 to h2 arrives at s1 on eth1
ICMP packet from h1 to h2 arrives at s1 on eth1
ICMP packet from h1 to h2 departs from s1 on eth2
```

[1]Edited for clarity; SIMON displays events as JSON by default.

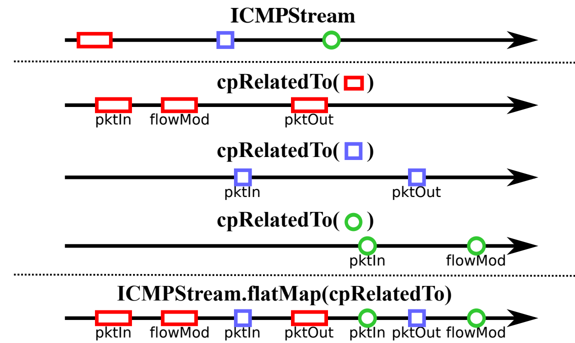


Figure 2: General illustration of finding related packets with `cpRelatedTo` and using `flatMap` to combine the resulting streams. Red rectangle, blue square, and green oval represent incoming ICMP packets. Each has related PacketIn, PacketOut, and/or FlowMod messages (indicated by text under the original symbol). `cpRelatedTo` places these messages into a separate stream for each original packet. `flatMap` then merges the streams into one stream of ICMP-related OpenFlow messages, keeping their relative ordering intact.

We conclude that the initial packet is dropped by the firewall (or, at least, delayed until well after the second ping is received). The next question is: *Did the program send the appropriate OpenFlow messages to configure the firewall?*

To answer this question, we need to examine OpenFlow events that are *related* to ICMP packets. To do this, we use SIMON's powerful `cpRelatedTo` function ("control-plane related to"). It takes a packet and produces a stream of all future OpenFlow messages *related* to that packet: PacketIn and PacketOut messages containing the packet, as well as FlowMod messages whose match condition the packet satisfies. We also use the `flatMap` stream operator; here it invokes `cpRelatedTo` on each ICMP packet and merges the results into a single stream (Figure 2 illustrates this operation on a separate, more general example). We write:

```
showEvents(ICMPStream.flatMap(cpRelatedTo))
```

SIMON shows that, while the expected pair of FlowMods are installed for the initial packet, no corresponding packetOut is received. This explains the incorrect behavior.

## 2.2 Debugging via Ideal Models

The operator could have also detected the bug by describing what a stateful firewall *should* do, independent of implementation, and having SIMON monitor the network and alert them if their assumptions are violated. There are three high-level *expectations* about the forwarding behavior of a stateful firewall:

1. It allows packets arriving at internal ports;
2. It allows packets with source $S$ and destination $D$ arriving at external ports if traffic from $D$ to $S$ has been previously allowed; and
3. It drops other packets arriving at external ports.

Note that none of these expectations are couched in terms of packetOut events or flow tables; rather they describe actual forwarding behavior that users expect to see. Also, they are stateful, in that the second and third expectations refer to the history of packets previously seen.

We can implement monitors for these expectations in SIMON. To keep track of the firewall's state, we create a mutable set of pairs of addresses called `allowed`. We then use SIMON's built-in `rememberInSet` operator to keep the set up-to-date:

```
Simon.rememberInSet(ICMPStream, allowed,
  {e: NetworkEvent =>
    if(isInInt(e))
      Some(new Tuple2(e.pkt.eth.dl_src,
                      e.pkt.eth.dl_dst))
    else None});
```

We pass in a stream of events (here, ICMP packet events), the set to be mutated (`allowed`), and a function that says what, if anything, to add to the set for each event in the stream. Now, as ICMP packets arrive on the internal interface, their source-destination pairs will be automatically added to the set. The `isInInt` ("is incoming on internal") function is just a helper defined in SIMON that returns true on packets arriving on internal interfaces. `{e: NetworkEvent => ...}` is Scala syntax for defining an anonymous function over network events.

We are now free to write the three expectations using SIMON's `expectNot` function, which takes a source stream (here, the ICMP stream), along with a function that says whether or not an event violates the expectation, and a duration to wait. It then returns a stream that emits an `ExpectViolation` if a violating event is seen before the duration is up and otherwise emits an `ExpectSucceed` event. The third expectation is most interesting. First we define a helper that recognizes external packets whose destination and source are not in `allowed`[2]:

```
def isInExtNotAllow(e: NetworkEvent): Boolean = {
  e.direction == NetworkEventDirection.IN &&
  e.sw == fwswitchid && fwexternals(e.interf) &&
  !allowed((e.pkt.eth.dl_dst, e.pkt.eth.dl_src))
}
```

`fwswitchid` and `fwexternals` are configurable parameters that indicate which switch acts as a firewall and which interfaces are considered external. Now we create a stream for this expectation, saying that whenever a packet should be dropped, we expect not to see it exiting the switch:

```
val e3 = ICMPStream.filter(isInExtNotAllow).flatMap(
  e => Simon.expectNot(ICMPStream, isOutSame(e),
         Duration(100, "milliseconds")));
```

We elide the first and second expectations for space, but they are similar. The `isOutSame` function accepts an *incoming* packet event and produces a new function that returns true on events that are *outgoing* versions of the same packet.

After defining all three expectation streams, we merge them into a single stream that emits events whenever any expectation is violated. As before, we can call `showEvents` on this stream. Other code can also use the stream to modify state, trigger actions on the network, or even feed into new event streams. Moreover, as we discuss in Section 5, once expectations have been written they can be re-used to check multiple applications.

# 3. WHY REACTIVE PROGRAMMING?

We now explain the advantages of reactive programming for network monitoring in more detail. Recall this expression from earlier:

```
showEvents(ICMPStream.flatMap(cpRelatedTo))
```

Now consider how one might implement it without reactivity. A natural solution is to use synchronous calls:

[2] SIMON's event field names (e.g., `pkt.eth.dl_dst`) follow the standard pcap format (`www.tcpdump.org`).

```
val seen = new Set();
while(true) {
  val e = getEvent();
  if(isICMP(e))
    seen += e;
  for(orig: seen) {
    if(relatedTo(orig, e))
      println(e);
  }
}
```

Not only is this much more involved, it also quickly becomes untenable because synchronous calls will *block*. Instead, asynchronous communication is needed. Callbacks are a standard way to implement asynchrony, but even with library support for callback registration, we get something like:

```
def eventCallback(e: Event) {
  if(isICMP(e))
    Simon.nwEvents().subscribe(
      makeRelatedToCallback(e));
}
def makeRelatedToCallback(e: Event): Event => Unit {
  e2 => if(relatedTo(e, e2)) println(e2);
}
Simon.nwEvents().subscribe(eventCallback);
```

In general, this approach can produce a tangle of callbacks registering even more callbacks ad nauseum. We could avoid this with careful maintenance of state throughout a single callback. However, that only moves the complexity into the callback's internal state, which harms reusability, compositionality, and clarity of the debugging script.

Callbacks also force an inversion of a program's usual control logic: external events *push* to internal callbacks; this can be confusing, especially when integrating with existing code. Instead, reactive programs maintain the perspective that the program *pulls* events. The necessary callbacks to maintain this abstraction are handled automatically by the language, and consistency between streams is maintained without any programmer involvement. (The canonical example of this is the way a spreadsheet program automatically updates a cell if other cells it depends on change. The same is true of streams in reactive programs.) In effect, reactive programming lets the programmer structure their program *as if* they were writing with synchronous calls that return values subsequent computations can consume, leading to a compositional programming style (which has already been used successfully in SDN controllers, most notably by Voellmy, et al. [24]).

Moreover, the ability to name streams, compose them with other streams, and re-use them as values is not something callbacks can easily provide. For these reasons, reactive programs tend to be concise, which makes monitoring—with the full power of a programming language behind it—brief enough to use at the debugging prompt.

There are at least two other applications of SIMON's streams that are worth mentioning:

**Interacting with the Network** As we alluded to in Section 2, streams can be easily fed into arbitrary code. For instance, suppose we wanted to write a monitor that sends continuous pings to host `10.0.0.1`. We create a stream that detects ICMP traffic exiting the network, and feed that stream into code that sends the next ping via `subscribe`:

```
ICMPStream.filter(
  e => e.sw == fwswitchid &&
  e.direction == NetworkEventDirection.OUT)
    .subscribe( _ => "ping 10.0.0.1" ! );
```

(The `!` operation, from Scala's `process` library, executes the preceding string as a shell command.)

**Caching Stream History** By default, SIMON's event streams do not remember events they have already dealt with, but sometimes visibility into the past is important. Users can apply the `cache` operator to obtain a stream that caches events as they are broadcast, so that new subscribers will see them. While this incurs a space overhead, it also enables *omniscient* debugging scripts that can see into the past of the network, before they were run.

## 4. A SIMON PROTOTYPE

As depicted in Figure 1, SIMON's architecture separates the sources of network events from event processing. We implement a fully functional prototype of SIMON's event processing component, and in this paper evaluate it by monitoring a Mininet-emulated SDN running different networks and controller applications. The current sources of events in our prototype rely on the visibility into the network afforded by Mininet, but the event processing framework and scripts do not. While an immediate contribution is the ability to monitor and debug arbitrary SDN environments running in Mininet—our original motivation was the frustration of doing just this—in Section 7 we discuss other potential sources of events that would enable SIMON in real networks.

We implement SIMON event processing atop Scala, using Scala's ReactiveX library (`reactivex.io`) to manage streams and events. SIMON's debugging prompt is the Scala command-line interface[3] plus a suite of additional functions we wrote for processing and reacting to events. Users can either use the prompt by itself, or load external scripts from the REPL.

Figure 3 contains a selection of ReactiveX operators (top), as well as a selection of built-in helper functions for network debugging and monitoring (bottom). These functions are built from ReactiveX operators and are sufficient for the examples of Sections 2 and 5. If needed, additional functions can be written the same way; SIMON's helper functions are themselves scriptable.

**Prototype Monitors** SIMON's event processing is independent of the types of events it receives, but of course the scripts and debugging power depend on the specific input event streams. SIMON receives events from monitor components through a JSON interface.

Our current prototype uses two types of monitors: a pcap monitor that captures both data plane and OpenFlow events, and an HTTP monitor that we use to capture REST API calls to a firewall running atop the Ryu controller (Section 5). Both monitors use JnetPcap 1.4 (`jnetpcap.com`), a Java wrapper for libpcap, and exploit the fact that we can capture all packets from Mininet. We use the APIs provided by JnetPcap and Floodlight (`www.projectfloodlight.org/floodlight/`) to deserialize data-plane data and control-plane data. The HTTP monitor currently assumes that the API calls will be contained in the first data packet of the TCP connection, which holds true for our tests.

The monitors use multiple threads to capture packets, which are timestamped by the kernel when captured. Due to the scheduling order of the threads, however, events may become accessible to SIMON out of order, so we implement a holding buffer to allow reordering of the packets. Empirically, a buffer of 50ms is enough to provide more than 96% of the packets in order. Because of the way the buffer is implemented, we change the interarrival time distribution of the packets seen by SIMON slightly, which has (minor) implications to reactive operators that depend on timeouts, such as `expect` and `expectNot`. In Section 7 we discuss a more general handling of time needed to apply SIMON to real networks.

## 5. ADDITIONAL CASE-STUDIES

We evaluate SIMON's utility by applying it to three real controller programs: two that implement shortest-path routing and a firewall application with real-time configurable rules. The firewall and one of the shortest-path applications are third-party implementations that are used in real networks. All are necessarily more complex than the basic stateful firewall of Section 2. However, unlike the previous example, none of these applications sends data-plane packets to the controller. Rather, the controller responds to northbound API events, network topology changes, etc.

**Shortest-Path Routing** We first examine a pair of shortest-path routing controllers. The first was the final project for a networking course designed by two of the authors. The second is RouteFlow [20], a complex application that creates a virtual Quagga (`quagga.net`) instance for each switch and emulates distributed routing protocols in an SDN.

The shortest-path ideal model in SIMON keeps up-to-date knowledge of the network's topology and runs an all-pairs-shortest-path algorithm to determine the ideal path length for each route. When the user sends a probe, the model starts a hop-counter appropriate to the probe's source and destination, and decrements the counter as the probe traverses the network. A non-zero counter at the final hop indicates a path of unexpected length, and the ideal model issues a warning. Note that the ideal model does not determine a *distinct* shortest path that packets must follow. Rather, the model is tolerant of variations in the exact paths computed by each application, so long as they are indeed shortest (by hop count). The shortest-path computation takes roughly 100 lines of code; the remaining model uses under 80 lines.

This example highlights an advantage of SIMON's approach: we were able to re-use the same ideal model for both implementations; once a model is written, its assumptions can be applied to multiple programs. Also, creating the ideal model did not require knowledge of RouteFlow or Quagga, but merely a sense of what a shortest-path routing engine should do. Of course, our model is a proof of concept, and assumes a single-area application of OSPF with known weights.

**Ryu Firewall** We also created an ideal model for the firewall module released with the Ryu controller platform (`osrg.github.io/ryu`). This module accepts packet-filtering rules via HTTP messages, which it then enforces with corresponding OpenFlow rules on firewall switches. These OpenFlow rules are installed proactively (i.e., the application installs them without waiting for packets to appear), but the rule-set is modified as new messages arrive.

To capture these rule-addition and -deletion messages, we took advantage of SIMON's general monitor interface to add a second event source, one that listens for HTTP messages to the controller. By creating a model aware of management messages, rather than depending on the OpenFlow messages created by the program, our SIMON model was able to check whether traffic-filtering respected the current firewall ruleset.

## 6. RELATED WORK

We relate SIMON to other work along the four axes that characterize it: *interactivity*, *scriptability*, *reactivity*, and *visibility into network events*.

**Scriptable Debugging** Scriptable debuggers are not new; many have been proposed, starting with Dalek [15], which can automate

[3] Also called a REPL, short for "read-eval-print loop". A REPL is an interactive prompt where expressions can be evaluated and programs run. Though sometimes called an "interpreter" loop, it can equally well interface to a compiler, as it does here.

| | |
|---|---|
| filter | Applies a function to every event in the stream, keeping only events on which the function returns true. |
| map | Applies a function to every event in the stream, replacing that event with the function's result. |
| flatMap | Like map, the function given returns a stream for each event, which flatMap then merges. |
| cache | Cache events as they are broadcast, allowing subscribers to access event history. |
| timer | Emit an event after a specified delay has passed. |
| merge | Interleave events on multiple streams, producing a unified stream. |
| takeUntil | Propagate events in a stream until a given condition is met, then stop. |
| subscribe | Calls a function (built-in or user-defined) whenever a stream emits an event. |

| | |
|---|---|
| expect | Accepts a stream to watch, a delay, and a function that returns true or false on events. Produces a stream that will generate exactly one event: an ExpectViolation or the first event on which the function returned true. |
| expectNot | Similar to expect, but the function describes events that violate the expectation. |
| cpRelatedTo | Accepts a packet arrival event and returns the stream of future PacketIns and PacketOuts that contain the packet, as well as FlowMods whose match condition the packet would pass. |
| showEvents | Accepts a stream and spawns a new window that displays every event in the stream. |
| isICMP | Filtering function, recognizes ICMP traffic. (Similar functions exist for other traffic types.) |
| isOutSame | Accepts an incoming-packet event and produces a function that returns true for outgoing-packet events with the same header fields. |

Figure 3: Selection of Reactive Operators and built-in SIMON helper functions. The first table contains commonly-used operators provided by Reactive Scala (`reactivex.io/documentation/operators.html`). The second table contains selected SIMON helper functions we constructed from reactive operators to support the examples shown in Sections 2 and 5.

repetitive tasks in gdb. Dalek's scripts are event-based, as in SIMON, but Dalek is callback-centric rather than reactive. MzTake [13] brought reactive programming to scriptable debugging. SIMON's use of the Scala command-line interface is partly inspired by MzTake's use of the DrScheme interface. Expositor [8] adds *time-travel* features to scriptable debugging, i.e., it allows users to view (and act on) events that have occurred in the past. SIMON has the capability to do the same, but we have not yet fully explored this direction. Expositor is also interactive, with a reactive programming style. All of these tools are designed for traditional program debugging, and so their notion of event visibility is different from SIMON's (e.g. method entrance and exit rather than network events).

**Data-Plane Invariant Checking** There has been significant work on invariant checking for the data plane. Anteater [12] provides off-line invariant checking about a network's forwarding information base. VeriFlow [9] extends the ideas of Anteater with specialized algorithms to allow *real-time* verification, ensuring invariants are respected as the rules in a live network changes. Beckett et al. [2] use annotations in controller programs to enable dynamic invariants in VeriFlow. Although powerful, these tools are limited to checking invariants about the rules installed on the network. For instance, the example of Section 2 would not be expressible in these tools, since the forwarding rules installed by the buggy program violate no invariants. SIMON does not require knowledge or annotation of controller program code to function, and its visibility is not limited to flow-tables.

Batfish [4] checks the overall behavior of network configurations, including routing and other factors that change over time. Like the above tools, it uses data-plane checking techniques, but the invariants it checks are not limited only to the data-plane. Batfish provides off-line configuration analysis, rather than on-line monitoring and debugging as SIMON does.

**Network Monitoring and Debugging** The NetSight [5] suite of tools has several goals closely aligned with SIMON. Chief among these tools is an interactive network debugger, `ndb`, which provides detailed information on the fate of packets matching user-provided filters on packet history. `ndb` is not scriptable, however, and its filters are limited to describing data-plane behavior, although control-plane context is attached to packet histories it reports. NetSight's postcard system allows it to differentiate between packets based on payload hashes, rather than using only packet header information (as our prototype monitor does); this means it provides more fine-grained packet history information than SIMON currently can. The matching invariant checker, `netwatch`, contains a library of invariants, such as traffic isolation and forwarding loop-freedom, and raises an alarm if those invariants are violated, along with the packet-history context of the violation. These invariants are limited in general to data-plane behavior; in contrast, SIMON provides visibility into all planes of the network.

Narayana, et al. [14] accept regular expressions ("path-queries") over packet behavior and encode them as rules on switches, avoiding the collection of packets that are not interesting to the user. This is in contrast to both NetSight and SIMON, which process all packets. However, the path-queries tool is neither interactive nor scriptable, and has visibility only into data-plane behavior.

OFRewind [26] is a powerful, lightweight network monitor that records key control plane events and client data packets and allows later replay of complete subsets of network traffic when problems are detected. While it is neither scriptable nor interactive to the level SIMON provides, its replay can be an excellent source of events for analysis with SIMON.

FortNOX [17] monitors flow-rule updates to prevent and resolve rule conflicts. It provides no visibility into other types of events, is not scriptable and has no interactive interface.

Y! [25] is an innovative tool that can explain why desired network behavior did *not* occur. Such explanations take the form of a branching backtrace, where every possible cause of the desired behavior is refuted. Obtaining such explanations requires program-analysis as well as monitoring, whereas SIMON has utility even if the controller is treated as a black box. Y! does not provide interactivity or scriptability.

**Other Network Debugging Tools** A number of other tools provide useful debugging information without monitoring. Automated test packet generation [28] produces test-cases guaranteed to fully exercise a network's forwarding information base. SDN traceroute [1] uses probes to discover how hypothetical packets would be forwarded through an SDN. The tool functions similarly to traditional traceroute, although it is more powerful since it allows arbitrary packet-headers to be tested. These also lack either an interactive environment or scriptability, and none leverage reactive programming.

SIMON's ideal-model description bears some resemblance to the example-based program synthesis approach of NetEgg [27]. However, NetEgg synthesizes applications from individual examples of correct behavior; ideal models fully describe the shape of correctness.

STS [21] analyzes network logs to find *minimal* event-sequences that trigger bugs. Although logs may include OpenFlow messages and other events, STS's notion of invariant violation is limited to forwarding state. Thus SIMON is capable of expressing richer invariants, although it does not attempt to minimize the events that led to undesired behavior. As STS focuses on log analysis, it provides neither scriptability nor interactive debugging.

# 7. DISCUSSION

**Reactive Programming** Section 3 discusses how reactive programming is a natural fit to deal with the inherent streaming and concurrent nature of network events. However, not all programmers and operators will feel comfortable with it. SIMON is flexible in this regard and allows any observable to invoke event-processing callbacks at any point in a script, and progressively incorporate reactive features.

**SIMON beyond Mininet** SIMON as presented is agnostic to, but only as useful as, the source of events that it sees as input. In this paper we prototyped SIMON using omniscient packet capturing enabled by Mininet. Given that Mininet allows the faithful reproduction of many networking environments and SDN applications, this is already valuable.

There are, however, many other potential sources of events that can make SIMON applicable to real networks. On a live network, port-mirroring solutions such as Big Switch's Big Tap [3] can serve as sources of events, and an OpenFlow proxy like FlowVisor [23] can intercept OpenFlow messages. It is also straightforward to feed SIMON with events from logs, such as pcap traces, which are routinely captured in test and production networks. NetSight's [5] packet history files can also be used a source of events to SIMON. Finally, SIMON's interactivity and scriptability offer an excellent complement to OFRewind [26]'s replay capabilities, which offer a hybrid between online and offline monitoring.

SIMON can also compile portions of debugging scripts that involve flow-table invariants to existing checkers such as VeriFlow [9] or HSA [7].

Narayana et al. [14] make a distinction between two types of monitors: "neat freaks," which record a narrow range of events but support correspondingly narrow functionality, and "hoarders," which record all events available. Our prototype monitor is a hoarder; it captures all network events, which are then filtered at the prompt or in a script. While this is practical in a prototype deployment it is less so in a real network under load. A solution would be to "neaten" SIMON by analyzing how scripts process event streams and, where possible, proactively circumscribing what traffic must be captured.

**Incomplete Information** Some sources of events will not provide all packets in the network. Mirroring ingress and egress ports only, for example, allows for end-to-end checks in SIMON programs, but not hop-by-hop. Sampling (e.g., sFlow [22] and Planck [18]) makes it infeasible to witness the same packet be forwarded by different switches along a path, as sampling is uncoordinated. Incorporating these with SIMON (e.g., via inference) is an interesting future challenge.

**Dealing with Time** Most networks can synchronize clocks to acceptable accuracy, but SIMON has to be prepared to deal with occasional timing inconsistencies. Our prototype naïvely orders packet events by their timestamps after a small reordering buffer, but in real networks we will need to extend SIMON's notion of time. Some scripts are only concerned with logical time; for these, SIMON only needs to potentially reorder events to be consistent with causal order. For these, SIMON has to maintain an internal notion of time, driven by the timestamps in the input streams, but properly corrected to be consistent with causal ordering. By observing pairs of causally related events in both directions among two sources, SIMON can compute correction factors and bounds for the different time sources in the network.

**Other Application Areas** SIMON applies to a wide range of situations beyond the illustrative examples seen here. For instance, SIMON could monitor a load-balancing application, sending events on a warning stream whenever balancing failed.

More broadly, networks that are not entirely controlled by a logically centralized program—e.g., networks with middleboxes—cry out for black-box methods that are nevertheless stateful. SIMON can even be used to debug problems in a non-SDN network, although it may be harder to pinpoint the cause of observed anomalies. SIMON also allows stateful debugging at the border between networks, even when one or more are not SDNs. Because it does not assume that flow-tables suffice to fully predict behavior, it can also be useful in detecting consistency errors [16, 19] or switch behavior variation [10].


### *Acknowledgments.*

We thank Aditya Akella, Greg Cooper, Minlan Yu, and the anonymous reviewers for useful feedback and discussions. This work was partially supported by the NSF.


# 8. REFERENCES


[1] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN traceroute: Tracing SDN forwarding without changing network behavior. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[2] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An assertion language for debugging SDN applications. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[3] BigSwitch Big Tap Monitoring Fabric. `http://www.bigswitch.com/products/big-tap-monitoring-fabric`.

[4] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Networked Systems Design and Implementation*, 2015.

[5] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Networked Systems Design and Implementation*, 2014.

[6] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Networked Systems Design and Implementation*, 2013.

[7] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Networked Systems Design and Implementation*, 2012.

[8] Y. P. Khoo, J. S. Foster, and M. Hicks. Expositor: Scriptable time-travel debugging with first class traces. In *International Conference on Software Engineering*, May 2013.

[9] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Networked Systems Design and Implementation*, April 2013.

[10] M. Kuźniar, P. Perešíni, and D. Kostić. What you need to know about SDN flow tables. In *Passive and Active Measurement*, 2015.

[11] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks*, 2010.

[12] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2011.

[13] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering Journal*, 2006.

[14] S. Narayana, J. Rexford, and D. Walker. Compiling path queries in software-defined networks. In *Workshop on Hot Topics in Software Defined Networking*, 2014.

[15] R. A. Olsson, R. H. Crawford, and W. W. Ho. Dalek: A GNU, improved programmable debugger. In *Usenix Technical Conference*, 1990.

[16] P. Perešíni, M. Kuźniar, N. Vasić, M. Canini, and D. Kostić. OF.CPP: Consistent packet processing for OpenFlow. In *Workshop on Hot Topics in Software Defined Networking*, 2013.

[17] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[18] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2014.

[19] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2012.

[20] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[21] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. B. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2014.

[22] sFlow. `http://sflow.org/about/index.php`.

[23] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *Operating Systems Design and Implementation*, 2010.

[24] A. Voellmy, H. Kim, and N. Feamster. Procera: A language for high-level reactive network control. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[25] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, 2014.

[26] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference*, 2011.

[27] Y. Yuan, R. Alur, and B. T. Loo. NetEgg: Programming network policies by examples. In *Workshop on Hot Topics in Networks*, 2014.

[28] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.