Leveraging SDN Layering to Systematically Troubleshoot Networks

Brandon Heller[†] Colin Scott[*] Nick McKeown[†] Scott Shenker[◇*]
Andreas Wundsam[‡◇] Hongyi Zeng[†] Sam Whitlock[◇] Vimalkumar Jeyakumar[†]
Nikhil Handigol[†] James McCauley[◇*] Kyriakos Zarifis[§] Peyman Kazemian[†]
[†]*Stanford University* [*]*UC Berkeley* [‡]*Big Switch Networks* [◇]*ICSI* [§]*USC*

ABSTRACT

Today's networks are maintained by "masters of complexity": network admins who have accumulated the wisdom to troubleshoot complex problems, despite a limiting toolset. This position paper advocates a more structured troubleshooting approach that leverages architectural layering in Software-Defined Networks (SDNs). In all networks, high-level intent (policy) must correctly map to low-level forwarding behavior (hardware configuration). In SDNs, intent is explicitly expressed, forwarding semantics are explicitly defined, and each architectural layer fully specifies the behavior of the network. Building on these observations, we show how recently-developed troubleshooting tools fit into a coherent workflow that detects mistranslations between layers to precisely localize sources of errant control logic. Our goals are to explain the overall picture, show how the pieces fit together to enable a systematic workflow, and highlight the questions that remain. Once this workflow is realized, network admins can formally verify that their network is operating correctly, automatically troubleshoot bugs, and systematically track down their root cause – freeing admins to fix problems, rather than diagnose their symptoms.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer - Communication Networks—*Network Communications*

Keywords

SDN, Software-Defined Networks, OpenFlow, Network Architecture, Troubleshooting

1. INTRODUCTION

Debugging a network is notoriously difficult. Network failures could be caused by acts of humans (invalid configuration, careless backhoe operators, malicious attacks), acts of protocols (route flapping, protocol interactions), or acts of nature. These problems occur frequently: according to a recent survey of network operators, 35% of networks generate more than 100 tickets per month [27].

To diagnose these problems, the majority of network operators employ `ping`, `traceroute`, and SNMP agents "often" or "very often" [27]. These tools deserve credit for keeping our networks up; they provide crucial visibility into end-to-end connectivity, routes, and traffic totals. Admins use them to diagnose the most difficult problems, such as loops caused by undefined interaction between spanning tree protocols, by combining their knowledge of protocol operation with the partial view provided by each tool.

Nevertheless, network troubleshooting today is still a largely ad hoc process. Given the critical role of networks, one might expect networking to have a more well-developed toolkit and methodology for tracking down bugs. This does not appear to be the case, especially when compared with the well-honed toolchains used to verify silicon and software. Diagnosing problems takes time: 24.6% of admins reported that tickets take over an hour on average to resolve [27]. Diagnosing problems also takes skill: understanding protocol interactions and driving the tools requires domain expertise. The requisite time and domain expertise for the current troubleshooting workflow is substantial.

When asked to describe their ideal tool, most admins said "automated troubleshooting" [27]. If such a tool could (1) detect errant behavior and (2) precisely locate its cause, admins could focus on (3), *actually fixing (or working around) bugs*. For example, the tool might point the admin toward a low-level issue, like a switch adding the wrong VLAN tag for a subnet, or it might point the admin higher, toward an incorrectly enforced access control policy. Ideally, the tool would even catch errors *before* they cause an issue (e.g., detecting a potential loop before it occurs).

Why are automated, complete troubleshooting tools absent, despite the desire for them? As explained in §6, such tools are impractically difficult to build within a traditional network; the limitations of our current ad hoc troubleshooting toolset are a direct consequence of the architecture in which these tools must operate. Fortunately, the recent development of Software-Defined Networks (SDNs) changes the architecture of the network control plane, creating a rare opportunity to re-think the troubleshooting workflow.

In this position paper, we articulate how a shift to SDN enables a more systematic troubleshooting workflow. We observe that most errors in an SDN are mistranslations between architectural layers. By bringing together a range of SDN troubleshooting tools from the research community [5,

HotSDN'13, August 16, 2013, Hong Kong, China.

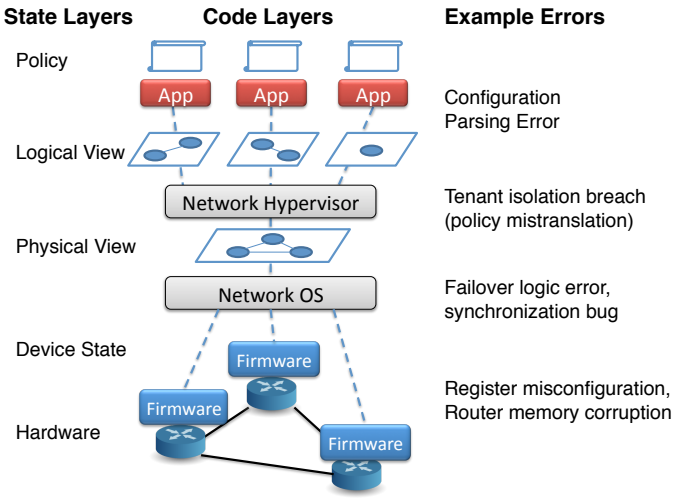

Figure 1: The SDN Stack: state layers are on the left, code layers are in the center, and example bugs are on the right.

10, 13, 14, 16, 17, 22, 25, 27], operators can readily detect problems and pinpoint their root cause, potentially in a fully automated way (similar in goals, but not approach, to the Knowledge Plane [7]).

This paper does not propose any new system; instead, it builds an overall picture of systematic troubleshooting in SDNs. We show how an expanding suite of SDN tools can diagnose or prevent problems across layers – from high-level policy errors to low-level hardware bugs.

2. THE SDN STACK

SDN factors the network control plane into functionally distinct layers, as depicted in Figure 1. *State layers* hold a representation of the network's configuration, while *code layers* implement logic to maintain the mapping between two state layers. State changes propagate in two directions: policy changes from above map to configurations (i.e., forwarding entries) of lower layers, while network events from below, such as link failures, map to view changes above.

At the highest layer are "control applications" (e.g., OpenStack Quantum [2]) that specify routing, access control, or quality of service policies by configuring the logical view. The logical view is a simplified, abstract representation of the network (often a single logical switch) designed to make it easy for the control application to specify policies. The network hypervisor maps the configuration of the logical entities to one or more physical entities in the physical view. The physical view has a one-to-one correspondence with the physical network, and it is the job of the network operating system (e.g., Onix [15], NOX [9]) to configure the corresponding network devices through a protocol such as OpenFlow [20]. Finally, the firmware in each network device maps forwarding tables to hardware configuration.

A key observation is that the *purpose* of the entire architecture is to translate human intent to the low-level behavior of the network, and each layer performs one piece of this translation process. Therefore, at any point in time, every state layer should be correctly mapped to every other state layer, a property we loosely call "equivalence". *Errors within the SDN control stack always result in broken equivalence between two or more layers.* For example, a breach of tenant isolation manifests itself as an inconsistency between a policy specified at the logical view ("Network A should not be able to see network B's packets") and the state of the physical network ("One of the switches is sending packets to the wrong destination network"). If all state layers are equivalent, but unwanted behavior persists, it must be the case that the configured policy does not match the operator's intent, or hardware beyond the control plane's view, such as an optical repeater, is broken.[1]

When the behavior of the network does not match an operator's intent, the operator must localize the symptoms to the system components that are responsible. As we show in §3, the layered architecture of SDN enables both humans and programs to reason about the fault localization process in a straightforward manner. Since errors manifest themselves as mistranslations between state layers, localizing the faulty component involves identifying the specific two state layers that are inconsistent. And because the interfaces between layers are well-defined (e.g., OpenFlow between the network operating system and switches [20], or OpenStack Quantum's web API between the control application and the network hypervisor [2]), we can readily build tools to check for equivalence between any two layers.

3. TROUBLESHOOTING WORKFLOW

Figure 2 shows a workflow for troubleshooting SDNs based on these observations. At a high level, this workflow consists of two phases: (1) a binary search through the control stack to identify the first code layer where a mistranslation occurs, followed by (2), a search within that layer to reduce the scope of those elements responsible for the invariant violation (e.g. ports, tables, rules, and code paths). Our goal in this section is to present a general search process; the exact search order depends on the available tools, and could differ from the order shown in Figure 2.

3.1 Find the Code Layer

Each step checks for equivalence between two state layers. If the state layers are equivalent, the problem is outside the covered layers; if not, we continue on a subset of the covered layers. The search starts by checking whether *any* network behavior appears broken.

A: Does the Actual Network Behavior Match the Policy? If yes, then the policy itself is the problem, and a human must resolve the discrepancy. If no, we continue.

B: Does the Device State Match the Policy? If yes, then the control plane is behaving properly; the problem must be lower in the stack, either in firmware code, or a hardware component itself (e.g., a link, port, or table), and Question D follows. If no, the problem is somewhere in the control plane, and we continue to Question C.

C: Does the Physical View Match the Device State? If yes, the actual device state is correctly synchronized with the physical view, then the problem must be above; go to Question E. If no, the network operating system is to blame; example bugs that could cause these layers to diverge include incorrect link event handling during a controller failover [22], unimplemented synchronization messages [11], parsing errors (e.g. a silent MAC address drop [3]), and single-bit memory corruption [1],

D: Does the Device State Match the Hardware? If yes, the network is not functioning correctly though the hard-

[1]Unless, of course, the monitoring tool itself is broken.

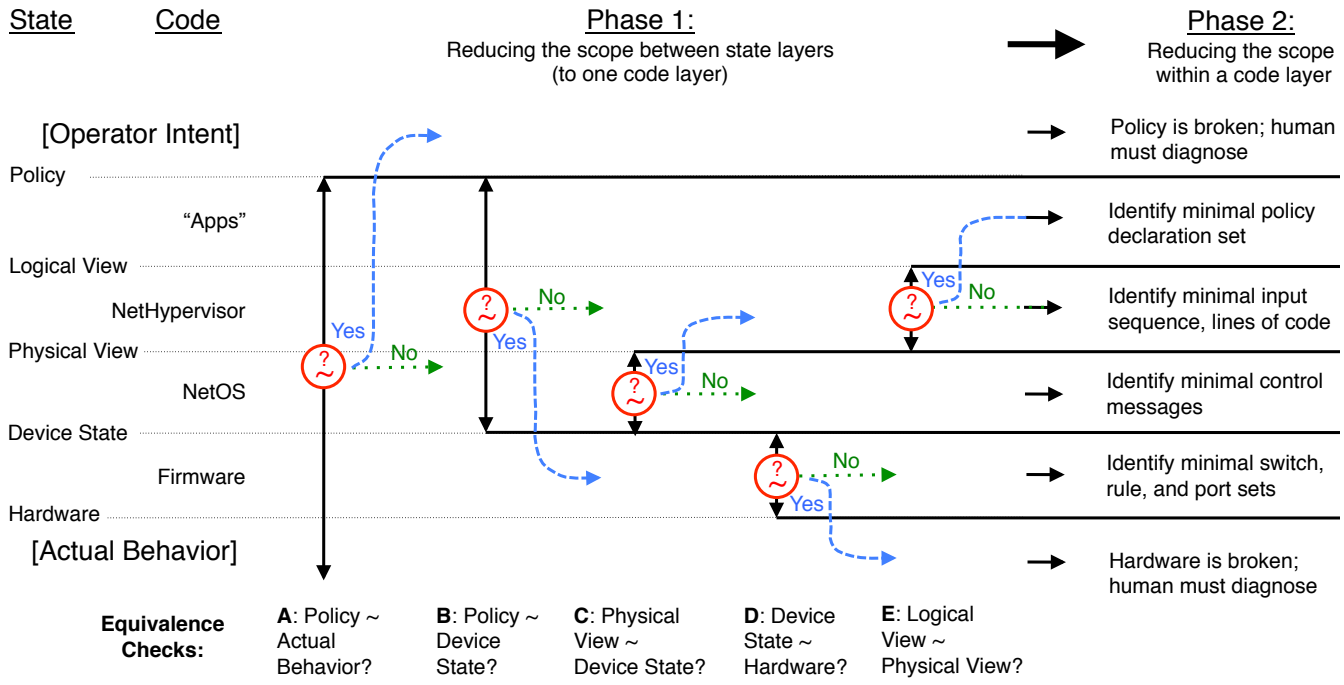

Figure 2: Bugs manifest themselves as non-equivalence (an improperly maintained mapping) between state layers. Our troubleshooting procedure: binary search to reduce the scope of a bug, by checking state layers for equivalence, then further localize within the identified code layer.

ware is configured correctly, which strongly indicates buggy hardware. If no, the firmware is to blame; example bugs include incorrectly mapping a single virtual forwarding table to multiple physical forwarding tables or incorrectly configuring TCAM or port group registers.

E: Does the Logical View Match the Physical View? If yes, the Apps layer is specifying the policy incorrectly to the logical view. If no, the network hypervisor is incorrectly translating from a logical to a physical mapping, possibly because of an incorrect view of the physical network. For example, undesirable behavior such as a tenant isolation breach can occur because the network hypervisor overlooks a link in the physical network topology, or path allocations computed by a traffic engineering application might fail to obey bandwidth constraints because the network hypervisor reports incorrect link capacities.

3.2 Localize Within a Code Layer

With the code layer identified, the next phase aims to further reduce the scope of the problem. The goal here is to be as precise as possible when localizing the root cause; for example, a firmware error would ideally be narrowed down to a single incorrectly configured forwarding rule.

Two general techniques help to localize within a code layer. Model checking automatically finds buggy code paths before they occur in practice, but can be computationally expensive. Alternately, once a problem has been observed, logging (coupled with post-processing) can programmatically hone in on its root cause.

As will become clear in §5, the tools used to localize within a layer differ, because the code may have wildly different implementations: firmware tends to use a lower-level language with a relatively constrained state space, while controller software tends to use a higher-level language with an effectively unbounded state space.

3.3 Workflow Examples

The workflow we have described naturally lends itself to automation. It suggests a path toward SDN troubleshooting that automatically localizes the source of many network problems, by leveraging the structure of control-plane layers, rather than the expertise of a master network admin.

We now show an example to help clarify the workflow steps. Network admin Alice has specified that no packets are allowed from the guest subnet to database backends. In the first scenario, Alice's controller application translates the policy incorrectly. In the second scenario, a new vendor switch has a firmware bug. For both scenarios, Alice observes that some of the background test traffic from the guest network is reaching the databases, which answers question A, "Does the Actual Network Behavior Match the Policy?"

Scenario One. For question B, "Does the Device State Match the Policy?", Alice uses a tool that answers "no"; the problem must be somewhere in the network OS or hypervisor. Next, asking question C, "Does the Physical View Match the Device State?", Alice finds out the answer to this question is "yes". She proceeds to question E, "Does the Logical View Match the Physical View?", and realizes that the network hypervisor is not causing the error; she concludes that her controller app must be the source of the problem. Her tools further localize the problem to a sequence of input messages. With a detailed, repeatable bug report, she verifies her bug fix and adds it to a regression suite for every controller build in the future, so that this specific input sequence will never break her security policy.

Scenario Two. For question B, "Does the Device State Match the Policy?", Alice uses a tool that answers "yes"; the problem must be somewhere below the interface to the switches. Moving to question D: "Does the Device State Match the Hardware?", her tool answers "no", so Alice con-

cludes that the new firmware has a bug somewhere. Her tools further localize the problem to one switch that received a specific input message that configured the hardware incorrectly. With a detailed, repeatable bug report, she contacts her vendor and verifies that their firmware update correctly handles this specific input sequence.

In both scenarios, Alice followed steps that eliminate a class of hypotheses, making continuous forward progress in her search, regardless of the type of bug.

4. TOOLS FOR FINDING THE CODE LAYER

This section and the next describe implemented systems that can help realize the troubleshooting workflow. Due to space limits, we focus on SDN-specific tools.

4.1 A: Actual Behavior ~ Policy?

This check requires the observation of real packets passing through the network, to verify that their behavior is compliant with the high-level policy. The tools shown here do not currently check directly against the high-level policy, but this is a natural next step for them.

Automatic Test Packet Generation (ATPG) verifies full reachability by sending packets to and from test terminals [27]. ATPG employs a model of the complete network forwarding state to build a list of test packets, where each test packet corresponds to a forwarding equivalence class at an input port. Rather than send the full list, ATPG finds a smaller test packet set that covers all forwarding rules, using an heuristic solver for the *min-set-cover* problem. Then ATPG sends test packets, looking for consecutive drops indicative of persistent software or hardware errors.

A second method is passive, in that it generates no new packets at test terminals. The **Network Debugger** (NDB) [10] is like a network-wide, path-aware `tcpdump`. NDB builds *packet backtraces*, which in addition to the packet headers, reveal the network path of a packet, any modifications along the path, and the full state of each switch when it forwarded a packet. Network analysis applications atop the stream of NDB-created packet histories can verify whether each observed path fits the policy, such as checking that each packet exited at an egress port.

4.2 B: Device State ~ Policy?

Example errors at this layer include loops, blackholes, and disconnectivity; each tool below uses a different technique to verify these configuration correctness properties.

Anteater translates connectivity invariants into boolean satisfiability problems, then checks them against the data-plane state using a general solver [17]. If there is configuration error, the approach returns a counterexample. Using a general solver simplifies the implementation but prevents network-specific algorithm optimizations.

Header Space Analysis (HSA) defines a "network algebra" where packet headers are represented by n-dimensional bit fields and switches and routers become functions on bit fields [13]. Atop this general representation for network forwarding state, HSA implements connectivity, isolation, and loop checking algorithms by enumerating the expected path of every group of packets that should traverse the same forwarding state. By employing lazy evaluation and compression methods, this all-pairs computation becomes feasible, and a recent extension makes the computation distributed and incremental [12].

VeriFlow is a related approach to incrementally verify dynamically changing network configurations [14]. VeriFlow constructs a model of the network state by monitoring control plane messages. It employs a multi-dimension prefix tree (trie) representation for network state, which is also used in packet classification.

4.3 C: Physical View ~ Device State?

We next check if the control plane's view of the network state matches the actual state of the forwarding devices.

OFRewind provides a start; it records all control plane messages in the network for offline processing and helps to detect control plane bugs such as OpenFlow parsing errors or non-compliant switch behavior, for a single controller [25]. Of all the stages, this is the most unexplored; we expect to see other methods built on distributed systems concepts like periodic verification of causally consistent snapshots [6].

4.4 D: Device State ~ Hardware?

The next check is whether the abstract configuration of each network device, e.g., a single match-action table, matches the low-level, hardware-specific configuration, e.g., registers to configure a forwarding pipeline. With knowledge of hardware internals, one could check for equivalence between simulated low-level hardware and a higher-level functional model (e.g., a Python-based switch) for any switch configuration and sequence of packet events (such as those captured by NDB).

However, a more general, proactive, black-box approach is the one taken by **SOFT** [16], which symbolically executes inputs to switches to identify where implementations' outputs diverge; this approach captures both implementation bugs and ambiguity in the specification of switch behavior.

4.5 E: Logical View ~ Physical View?

The last check is whether the abstract logical view seen by the control applications properly maps to the physical view configured by the network hypervisor. One example, **Correspondence Checking**, verifies that each path in the logical view has a corresponding path in the physical view by computing the transitive closure of possible headerspace transformations in the network configuration [23].

5. TOOLS FOR LOCALIZING WITHIN A CODE LAYER

At this point, the problem has been narrowed to a single code layer. The tools to further localize problems fall into two categories: those designed for controller software, and those designed for firmware running at forwarding devices.

5.1 Localizing Within the Controller

Localization within the controller generally involves two tasks: (1) identifying the events that lead up to the bug, and (2) finding the responsible line(s) of code.

Replay techniques such as **OFRewind** [25] support the first task, helping to retroactively infer OpenFlow events that triggered a bug observed in logs. A more programmatic approach is **Retrospective Causal Inference** (RCI) [22], which automatically infers the minimal set of inputs to control software that is sufficient for triggering a bug observed in

a trace of network events. To find the *minimum causal set*, RCI iteratively prunes events to find the smallest sequence that causes the same buggy outcome.

Traditional source-level debuggers are a common tool for the second task, locating the responsible code line(s). Model inference tools such as **Synoptic** [4] can help developers build a state-transition model to identify errant state transitions. Another approach is model-checking: **NICE** [5] enumerates all code paths (where computationally tractable) taken by OpenFlow control software and identifies message sequences that lead to invalid network configurations.

5.2 Localizing Within the Switch Firmware

Localization within the switch generally involves (1) identifying the network device responsible for the problem, and (2) finding an errant device parameter or line of code.

NDB's tracing mechanism and **ATPG's** triangulation mechanism naturally reduce the scope of a bug to a single node, link, or forwarding entry [10, 27]. Model checking is particularly amenable for localizing bugs within firmware, where the state space is more limited than general software. **SOFT** [16] model checks OpenFlow agents, helping to catch and localize tricky OpenFlow compliance bugs.

6. IS SDN NECESSARY?

This section contrasts requirements to realize our workflow on an SDN versus a more traditional network.

Knowing the operator's intent: Automated troubleshooting requires a clear description of "correct operation": the network operator's intent. In an SDN, policy can be a first-class citizen; intent can be *explicitly* expressed at the policy layer, while code compiles the policy description into lower-level configuration. To express a tenant isolation policy, like "A should be able to talk to B, but not C", an admin can define a virtual networks for A, B and C with connectivity between A and B [2]. Their intent is clear.

In a traditional network, it is practically impossible to unambiguously discern the operator's intent; it is *implicitly* expressed as the combination of all protocol configurations over all nodes. For example, an admin might isolate tenants by inserting A and B on the same VLAN with a static rule to prevent the IP prefix range of C's VLAN communicating with A and B's VLAN.[2] To infer intent, one must gather this state from every node, understand multiple vendor-specific and protocol-specific formats (different protocol configs), understand the *composition* of each protocol config (the precedence rules between protocols operating at the same data-plane layer), and implement logic to infer intended network behavior from configuration. This is no trivial exercise.

Checking network behavior against intent: Automated troubleshooting requires a way to compare operator intent against state in forwarding elements. In an SDN, forwarding state is protocol-independent, accessible, and simply described; it is *explicitly* exposed as the device state and used directly by many tools.

In a traditional network, forwarding state is determined by independent switches and routers running distributed protocols, which communicate until they converge on stable forwarding state. Reading the complete distributed forwarding state requires either generating a consistent snapshot [6] or centrally collecting logs with precisely synchronized timestamps. These features are missing today for individual protocols, let alone *every* protocol together. As a consequence, the troubleshooting process must analyze the network with no guarantee that its view of the network state is consistent. Contrast this with SDN, where the job of the controllers is to maintain a consistent view of the network state; this architectural layer avoids the need to understand the combined operation of every interacting network protocol. Even if traditional networks were somehow augmented to provide state snapshots, and administrators had some way to convey their intent, checking network behavior would still require making sense of the combined operation of every active protocol.

To summarize, one *could* augment a traditional network to automate troubleshooting, but you would have to overcome daunting practical challenges. Although SDN was not specifically conceived to simplify troubleshooting, a simplified troubleshooting workflow seems to naturally follow from the layering inherent to the architecture.

[2]Actually, this description leaves out a number of steps [18].

7. UNANSWERED QUESTIONS

Many aspects of SDN troubleshooting remain unexplored; this section describes related questions that we find exciting.

How can we integrate program semantics into network troubleshooting tools? Since SDNs are programs operating on a network view, they might benefit from network-specific versions of standard software engineering tools like debuggers (e.g., `gdb`), memory leak checkers (i.e., `valgrind`), profilers, (e.g., `gprof`), and race condition checkers, all of which use program semantics to execute a modified version of the code. Extending troubleshooting tools to understand network program semantics enables new possibilities, like bug reports that detail the program states and inputs that triggered an errant network state, as well as debuggers that handle breakpoints at each SDN state layer.

How can we integrate troubleshooting knowledge into network control programs? Going the other direction, providing troubleshooting state to network programs might enable automatic recovery from software crashes [21] or safely divert traffic away from a switch whenever an input sequence triggers an implementation bug [26].

How can we improve invariant checkers? Invariant checkers now run in real-time [13, 14], but the evaluations use relatively static IP and VLAN rule sets with few packet modifications, one controller, no stateful middleboxes, and few topologies. We should understand and improve their costs for larger, more dynamic networks, plus those not using destination-based IP forwarding, such as scale-out data centers or WANs with integrated load balancing. Additionally, checkers could integrate invariants for QoS, traffic engineering, fault tolerance, and security.

How can we troubleshoot mixed networks with legacy devices? Commercial SDN deployments will include legacy devices, complicating the task of obtaining a consistent network view and checking layer equivalence.

What abstractions are useful for troubleshooting? Abstractions for programming SDNs can eliminate certain classes of bugs such as network loops, as well as simplify the creation of network control programs [8, 19]. Yet, aside from the packet backtrace construct in NDB, we are unaware of similar abstractions that simplify the creation of new tools explicitly designed to aide troubleshooting [10]. Extracting

simplicity out of complex networks and into abstractions leads to better systems that we can more easily build and reason about [24].

8. DISCUSSION

Other industries have well-developed methodologies and tools for tracking down and finding the root cause of bugs, based on formal models and clear abstractions. SDN brings the same opportunity to networking, by clarifying the role of each layer in the control hierarchy and enabling us to compare the operator's intent with the actual network operation in a way that was not previously practical. Over the next few years, we expect a proliferation of tools to emerge for testing different parts of the SDN hierarchy. Our systematic approach provides a framework to bring current systems, as well as the ones we expect to see soon, into a coherent whole.

9. ACKNOWLEDGMENTS

This research is supported by NSF grants CNS-1040838, CNS-1015459, CNS-1040190 (NEBULA), CNS-0832820 (POMI), and CNS-1040593 (FIA), as well as the Open Networking Research Center (ONRC).

10. REFERENCES

[1] Amazon S3 Availability Event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`.

[2] OpenStack Quantum. `http://wiki.openstack.org/Quantum`.

[3] NOX commit c3fa89a8e5. `http://noxrepo.org/git/nox-classic/commit/c3fa89a8e5`, 2010.

[4] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE, 2011.

[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[6] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1985.

[7] D. Clark, C. Partridge, J. Ramming, and J. Wroclawski. A knowledge plane for the internet. In *SIGCOMM*, 2003.

[8] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 2011.

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *CCR*, 38, 2008.

[10] B. Heller, N. Handigol, V. Jeyakumar, N. McKeown, and D. Mazières. Where is the debugger for my Software-Defined Network? In *HotSDN*, August 2012.

[11] HP Switch Software OpenFlow Supplement. `http://bizsupport2.austin.hp.com/bc/docs/support/SupportManual/c03170243/c03170243.pdf`, 2012.

[12] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, 2013.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. In *NSDI*, 2012.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Brighton Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. OSDI '10, 2010.

[16] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A soft way for openflow switch interoperability testing. In *CoNEXT*, 2012.

[17] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. SIGCOMM '11, 2011.

[18] The Modular Network-in-a-Box: What Could Happen if SDN Thinks Big. `http://packetpushers.net/the-modular-network-in-a-box`.

[19] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walke. Composing software-defined networks. NSDI, 2013.

[20] The OpenFlow Switch. `http://www.openflowswitch.org`.

[21] F. Qin, J. Tucek, and Y. Zhou. Treating bugs as allergies: a safe method for surviving software failures. HotOS, 2005.

[22] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker. How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software. Technical Report UCB/EECS-2013-8, EECS Department, University of California, Berkeley, 2013.

[23] C. Scott, A. Wundsam, K. Zarifis, and S. Shenker. What, Where, and When: Software Fault Localization for SDN. Technical Report UCB/EECS-2012-178, EECS Department, University of California, Berkeley, 2012.

[24] S. Shenker. The Future of Networking, and the Past of Protocols. In *Open Networking Summit*, October 2011.

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

[26] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: predicting and preventing inconsistencies in deployed distributed systems. NSDI, 2009.

[27] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *CoNEXT*, 2012.