OFTEN Testing OpenFlow Networks

Maciej Kuźniar
EPFL

Marco Canini*
TU Berlin / T-Labs

Dejan Kostić
EPFL

***Abstract*—Software-defined networking and OpenFlow in particular enable independent development of network devices and software that controls them. Such separation of concerns eases the introduction of new network functionality; however, it leads to distributed responsibility for bugs. Despite the common interface, separate development entails the need to test an integrated network before deployment. In this work-in-progress paper, we identify the challenges of creating an environment that simplifies and systematically conducts such tests. We discuss optimizations required for efficient and reliable OpenFlow switch black-box testing and present a possible approach to address other challenges. In our preliminary prototype, we combine systematic state-space exploration techniques with real switches execution to explore an integrated network behavior. Our initial results show that such methods help detect previously unrevealed inconsistencies in the network.**

I. Introduction

Software-defined networking (SDN) empowers third-party developers to create control software that tailors the network behavior to specific applications—a vision currently enabled by OpenFlow [1], which is de-facto the standard interface for programmatically managing switches. Like others, we believe SDN provides the opportunity for defining a more principled approach to networking than the current state-of-the-art. However, it is clear that the introduction of greater programmability, despite its many advantages, raises the risk of software faults (or simply bugs). This has undesirable implications for the success of SDN, since developers are relatively inexperienced, especially in this early stage.

The SDN architecture, still undergoing redefinition and refinement through the literature [2]–[4], effectively decouples the computation of network state from its distribution. As such, there are several opportunities for bugs to creep in. That is, faults may take place in any of the components across the SDN stack. To start with, the high-level controller application logic may be buggy. Programming errors may also affect the SDN platform, that is, a software layer that isolates the application from low-level details such as topology discovery, state distribution, fault tolerance, *etc.* The software agents running and implementing OpenFlow on the switches may contain software defects. Additionally, since SDN enables independent development of network equipment and its control software, vendors may misinterpret the OpenFlow specifications or create implementations that are not fully compatible and interoperable.

*Work done while the author was with EPFL.

In an attempt to ensure that network reliability is not impaired by bugs, researchers have recently started working on a rapidly-growing set of tools and techniques for debugging and troubleshooting software-defined networks. For example, our NICE [5] tool finds bugs in controller applications through systematic testing of application behaviors. OFTest [6] is the reference suite of functional tests to check an OpenFlow agent for adherence to protocol specifications. OFLOPS [7] is a switch performance testing framework. Finally, HSA [8] and VeriFlow [9] verify that the forwarding tables satisfy desired network-wide invariants.

Note that these works focus on specific classes of problems and on a particular layer of the SDN stack. However, an important aspect that has not received much attention is that ultimately, it is necessary to test whether all components of an SDN system work correctly when they are integrated together. Our inability to formally verify complex systems or the prohibitive costs associated with such an approach make integration testing a common best practice. Therefore, for SDN as for any complex system, testing is a crucial process to gain confidence about correct intra- and inter-component behavior.

Testing an integrated network is not alternative but rather complementary to individual component testing. In particular, it addresses several shortcomings that are introduced when testing components separately. For instance, testing control software in isolation requires using an OpenFlow switch model, which reduces fidelity [5]. Conversely, testing a switch for OpenFlow compliance [6] validates the implementation at the level of individual protocol features but does not shed light on the cross-feature interactions exercised by a certain controller application.

Conceptually, testing an integrated network is a straightforward process. This form of testing involves setting up a testbed with deployment-like conditions including wiring the switches in a meaningful topology, configuring multiple controllers based on performance and availability requirements, *etc.* Next, by following a testing specification, the testbed is subjected to several exemplary scenarios while the system behavior is compared to the desired outcome (which is expressed in the testing specifications). Unfortunately, writing testing specifications is a tedious, manual process. Beside considering scenarios that account for common cases, one also needs to reason about how to capture corner cases that arise due to unforeseen delays and event reorderings

in the network. Moreover, the testing process requires a precise and detailed specification of desired system behavior. For complex scenarios, specifying correct behavior is often nontrivial and error-prone.

In this work, we experiment with another approach that leverages an observation that existing tools for testing controller applications employ techniques to identify test case scenarios. Moreover, using such tools already requires high-level application correctness properties to be specified. Therefore, we want to explore what is the effort required to take an existing testing tool (we choose the home-grown NICE [5]) and extend it to enable testing an SDN system consisting of one or more controllers and a potentially heterogeneous collection of real switches. In doing so, we check that the system operates without violating a preexisting list of correctness properties.

NICE uncovers bugs in controller applications by systematically exploring the state space of an SDN system. Here, system refers to the composition of an unmodified OpenFlow controller with a model of the network environment. While state-space exploration is essentially a brute-force approach, NICE makes the process more efficient by augmenting model checking (the core technique for systematic testing) with symbolic execution (to automatically derive relevant test inputs). The network model, which includes switches, end hosts and a stub replacement of the NOX controller platform, also contributes to the tool efficiency, though at the expense of testing fidelity

Our goal is to enable systematic testing of an integrated OpenFlow network with the ultimate objective of gaining confidence in whether controllers and real switches work correctly together in deployment-like settings. The high-level idea is to systematically exercise behaviors of control software (as thoroughly as possible), let the controller execution interact with real switches, and observe whether there exist any erroneous conditions.

The main challenge we face is the inclusion of real switches instead of using switch models in the testing harness. In particular, we need to deal with several low-level details such as communicating with the switches and coalescing network state without affecting the representativeness of the underlying state. In addition, we want to keep the number of assumptions we make about switch and controller behaviors to a minimum.

We argue that a possible approach is to create OFTEN: an OpenFlow Testing Environment. OFTEN is a tool for systematic testing of integrated OpenFlow networks. Building upon NICE, it explores possible execution paths of controller applications and network event orderings. It extends NICE by enabling communication between the model checker and real switches. To do so, we construct an environment model for the switches that is capable of emulating controller, end hosts and communication channels. We then add the necessary glue to synchronize the state of the switch model used in NICE with the state of this dual environment model surrounding the real switch. The synchronization works by fetching the flow tables from real switches and by controlling the timing of network events such as processing packets or OpenFlow messages. Finally, OFTEN reports both network-wide correctness issues as well as inconsistencies between switch models and real switches used for testing.

The rest of the paper is organized as follows. In Section II, we discuss the challenges that need to be tackled while creating OFTEN. In Section III, we describe our approach to testing OpenFlow network components. Section IV presents preliminary results of our OFTEN prototype. We place our work in the context of related work in Section V and Section VI concludes the paper.

II. Challenges

The design and development of any network-wide testing tool entails a variety of challenges. In addition to usual requirements such as providing high coverage and being scalable, there are several domain specific problems that need to be addressed.

Switch as a black-box: To provide a general solution, the switches should be treated as black-boxes. We expect the number of different OpenFlow switches to grow rapidly. Therefore, the testing process should rely on a common, standardized interface to minimize the overhead for testing different network gears. Therefore, switches under test are expected to expose the interface following the OpenFlow specification [1]. The specification does not define how the internal mechanisms are implemented, leaving choice to vendors. Thus, only the externally visible behavior of the switch should be checked.

However, the specification does not provide any interface to fetch a device's state. Specifically, there is no preexisting way to reliably check whether a switch has finished processing a test packet. In principle, such a mechanism exists for message processing on the control channel: the barrier request message from the controller forces the switch to process all previous messages before replying with a barrier response. In practice, for data-plane packets no such information is available and there are sources [7] suggesting that the barrier command is not implemented reliably.

Correctness definition: To verify that the system behaves as desired, one needs to specify the expected behavior for each test case. Validating network correctness requires reasoning about correctness at two levels.

First, the network-wide correctness properties need to be defined and checked to ensure there are no implementation and/or logic bugs in the controller causing invalid network behavior. Although there are some common properties that can be predefined (*e.g.*, loop-freedom), preparation of application-specific ones might be a nontrivial and time-consuming process. Moreover, it is difficult to define correctness properties that are never violated even during some

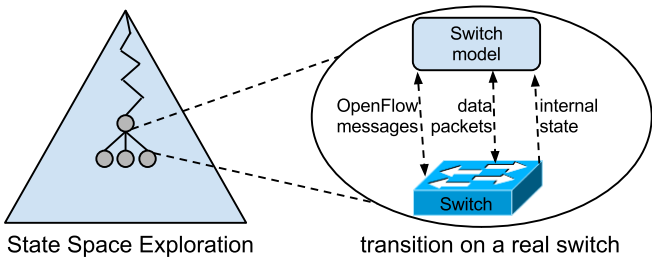

Figure 1: High-level overview of the tool. Combination of systematic state-space exploration of the whole network and execution of specific transitions on a real device.

transient conditions.

Second, to improve testing precision and aid in debugging, low level switch correctness needs to be checked each time it executes any action. Traditionally a developer specifies when the test case succeeds and when it fails. However, because the testing process is automated, the valid response of the switch (accounting for the current state) to a given input should be deduced without human interaction. Moreover, OpenFlow switch behavior may in some cases be nondeterministic. Thus, the checking methodology must take this into account and accept multiple correct outputs.

III. OpenFlow Testing Environment

We start by presenting a high-level overview of OFTEN, our tool for systematic testing of integrated networks.

As mentioned, there are several layers and different parties that may be responsible for incorrect network behavior in an OpenFlow network. Thus, we aim to systematically test interactions between network components before deployment. To do so, we use a model checker to explore possible system execution paths and we create a testing harness that enables us to extend the reach of the model checker to real OpenFlow switches. Fig. 1 presents an overview of the OFTEN architecture.

We use the home-grown NICE [5] tool as the engine driving the testing process. At the core of NICE is a model checker. NICE views an OpenFlow network as a transition system. At each state, the system exposes a set of possible transitions, each of which evolves the system from one state to another. Model checking systematically explores system behaviors by exercising possible system executions, that is, sequences of transitions. To check system correctness, NICE tests after each transition that predefined correctness properties hold in the current state.

However, NICE is designed to test controller applications. Therefore, all system components except for the controller are replaced by simplified models. Thus, we must employ additional techniques that enable us to replace the environment model with real network devices. In the rest of this section, we describe the techniques we adopt to create OFTEN.

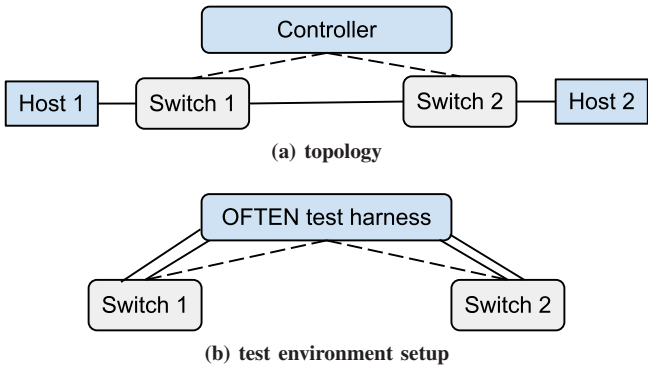

Figure 2: Topology of a tested network (a) and a corresponding test environment setup (b).

First, to make the controller–switch interaction possible, we associate each switch model in the network environment with a physical switch. Communication with the switch happens at the level of network interfaces. The tested device has to expose data-plane network interfaces to which an OFTEN test harness running on a testing machine directly connects. Additionally, the OFTEN test harness contains an emulated controller that is capable of sending and capturing OpenFlow messages in a controlled fashion. We connect all tested switches to this controller. OFTEN supervises the emulated controller, listens on the aforementioned interfaces, and injects packets directly to them. Fig. 2 shows a topology of a tested network (a) and a corresponding testing setup with OFTEN (b). The test harness emulates end hosts and is directly connected to both switches.

To increase usability, we minimize the requirements that the switch must satisfy. The only assumption we make about the switches is that they are compatible to some extent with the OpenFlow protocol. This assumption is reasonable considering that OpenFlow is the functionality we are testing. The required specification subset is not defined and depends on what features are used by the controller under test. However, OFTEN relies on basic, mandatory protocol features to coordinate the testing process.

Moreover, a model checker needs to be aware of the state and possible transitions for each component of the system it is testing. It also needs to execute transitions on the components in a controlled and organized fashion. A purposefully designed model provides an easy access to such data and all features required by a model checker. However, a real device exposing a narrow interface does not guarantee such level of external control as the OpenFlow protocol is not designed with testability in mind.

We choose to check correctness on two levels when testing with real devices. We decide to rely mostly on user-defined network-wide correctness properties, but complement them with a fine-grained comparison of switch outputs. We argue, that the network administrator is not interested in switch-level differences unless they significantly affect the network.

Moreover, the OpenFlow specification is at times ambiguous and allows certain nondeterministic switch behaviors. A low level comparison of outputs for a model and the real switch would unnecessarily report such cases and introduce false positives. Although two switches that temporarily behave differently lead to an inconsistent network state, the final result may be correct for both. In our approach, the user receives a high level report with property violations, and is warned about differences in switch outputs for the same input.

Unfortunately, some correctness properties specified in NICE are too fine grained to be useful in our approach. They require access to internal components of the switch in order to, for example, verify if there are no packets that are permanently left in any buffer. Such properties need to be redesigned taking into account the assumed interface of the switch. However, other, higher level properties (*e.g.*, detection of lost packets in the network) stay unchanged and may be reused in OFTEN.

A. Propagating switch state to a model

We now describe how we gather all required state from switches. A model checker explores a space of system states based on a system model composed of the controller applications and models of switches and end hosts. Each model describes the state and available transitions of its corresponding component. In particular, a switch is defined by: (i) content of its flow table, (ii) communication channels and (iii) available transitions. The switch state kept in the model has to be consistent with the state of the real device. Thus, OFTEN synchronizes both states after finishing each transition affecting the switch.

The first important element of switch state is the flow table. A switch may be queried for the content of its flow table using the OpenFlow interface. Thus, getting such information requires just interpreting the switch response.

Switches use two types of communication channels: one per port for packets and one for OpenFlow messages. There is no standard interface used by switches to expose information about the content of the buffers used to temporarily store packets in the switches while processing them. However, although these buffers are parts of an OpenFlow switch, in our solution they are included only in the model. OFTEN sends a packet to the switch immediately before the switch runs the transition processing the packet. Therefore, the buffers in the switch are empty after each transition finishes and we do not have to query the switch for such information.

There are two types of transitions available for a switch: (i) related to data packets and OpenFlow messages handling, and (ii) time-related flow table modifications. We consider only transitions that have externally visible effects. The transitions that belong to the first group are available only if there are packets in the corresponding communication channels. As the communication channels are modeled externally to the switch, the availability of the aforementioned transitions is implied from the model.

The transitions in the second group represent timer-initiated removal of rules from the flow table. To check interleaving of different time-related events, the model checker explicitly forces the timer event as one of the available transitions. Therefore, the availability of such transitions is not defined by the switch timers, but by the model itself.

B. Executing atomic transitions

A model checker performs a step of state space exploration by executing one of the transitions enabled in current state. Therefore, we need to define a set of transitions and provide mechanisms to execute them. While applying a transition to a model controlled by the model checker is straightforward, it is more complex in the context of OpenFlow switches. An OpenFlow switch is a nonterminating program that runs continuously, handling internal (*e.g.*, timers) and external (*e.g.*, packet arrival) events.

OpenFlow switches run close-sourced, proprietary software that is also difficult to modify in case of hardware switches. Therefore, we assume no access to the source code and treat switches as black-boxes that satisfy only one compulsory requirement: providing the OpenFlow interface. There are two phases (shown in Fig. 3) of the externally controlled execution of transition in the switch: (i) forcing a transition to start and (ii) determining when it ends.

Starting packet processing. We need to induce a transition execution at the right time. In a standard switch mode, packet handling starts soon after the switch receives a new packet. Therefore, transitions used in OFTEN are fired by sending a packet (a data-plane packet or an OpenFlow message) to the switch. Additionally, the order of handling packets that arrive in a short time period and are placed in different queues is nondeterministic. To make this process deterministic, OFTEN keeps external packet queues in the network model. When the model checker executes a packet handling transition, the testing harness sends the first packet in the queue to the switch. As a result, OFTEN can control when the transition starts in a real switch. It is safe to assume that soon after an input is available it is processed by the switch. That is, we do not expect a situation where the switch receives a packet and does not process it for a long time. As we still have no precise guarantees how much time the processing takes, we have to additionally determine when it ends. We present our solution in the next subsection. Additionally, using external queues allows us to apply transitions simulating network events. Such transitions include packets reordering, dropping, duplicating and other events happening in the network.

Determining the end of event processing. As mentioned, the ability to reliably determine when the black-box device finished all internal processing related to the provided input

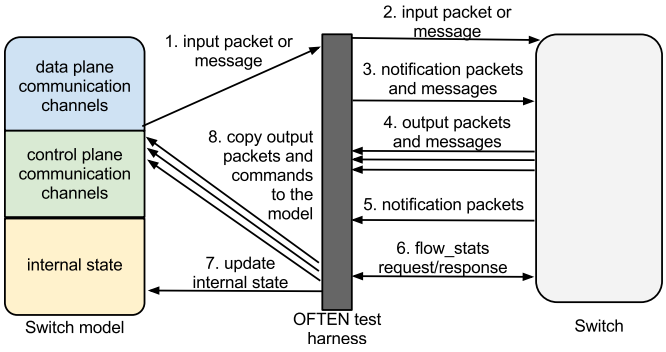

Figure 3: Execution of a handling transition on a real switch. Sequence of communication and data propagation events between a switch model and a real switch supervised by OFTEN.

is a nontrivial problem. The OpenFlow interface does not provide a way to get such information. The simplest solution would be to wait for a predefined time interval after supplying the switch with an input. However, this approach is not reliable and requires choosing an interval sufficiently long to minimize the probability of missing any behavior, which increases the overhead while still offering no certain guarantees. To solve this problem, we propose a few techniques that depend on an additional assumption about the switch capabilities. Although according to the OpenFlow specification [1] a switch is allowed to arbitrarily reorder processing of control-plane messages, our observations of existing software and hardware devices show this not to be the case. Therefore, we assume that events are processed according to their arrival order at the switch. We also assume that all packets arriving to the same data-plane port are processed sequentially. We believe these assumptions are reasonable since we inject inputs in a controlled fashion and at a relatively slow pace. Next, we describe the issues that we tackle in more detail.

• *Control-plane message processing:* The most reliable solution is to follow each message with a barrier request. When a switch receives a barrier message from the controller, it has to finish processing all earlier messages before sending back a barrier response. However, this feature is not widely and reliably supported in hardware switches [7], [10]. As a workaround, OFTEN injects an additional Packet_Out command containing a well-known packet after each message from the controller. When we detect this packet, we assume all processing is finished. Another solution is to follow each control message with a flow install–flow remove pair and wait for the flow removed notification.

• *Data-plane packet processing:* As there is no equivalent to the barrier request for data-plane packets, we have to provide a different way to detect when the switch finishes processing a packet. There is a wide range of possible actions applied to a packet. It can be dropped, forwarded to one or multiple ports, or sent to the controller inside a Packet_In message. Therefore, assuming no knowledge of the forwarding state, it is not possible to know in advance how many packets should appear as output. Thus, we implement a general solution based on the assumption that all packets arriving to the same data-plane port are processed in their arrival order. We first install two high-priority rules that forward special well-known packets to a particular port and to the controller, respectively. Then, OFTEN follows each injected testing packet with two well-known packets matching the aforementioned rules. When both are processed (one appears on the expected port and the other is sent as the Packet_In message to the controller), we assume the original packet was also handled.

C. Specifying root-causes of problems

Finding a problem in the network without pinpointing the misbehaving device and conditions of when it happens does not help administrators and developers sufficiently. To aid in the debugging process, we take advantage of the diversity of multiple OpenFlow switch implementations based on the same specification.

There are two possible correctness properties violation scenarios when testing a network with OFTEN. First, the same correctness properties are violated when running with both model and the real device. Such a result suggests the controller is the source of the problems and need to be investigated first. In the second case, there are differences in correctness properties violated when using a model and a real device. Then, it is unclear which component should be blamed. However, in such a situation, testers may run OFTEN using OpenFlow switches from different vendors. Then, they compare the detected problems and choose the most common behavior as a correct one. This approach is based on the assumption that the same error can be made by many vendors with a very low probability [11]. Finally, if the tested device violates correctness properties others do not violate, this device is probably faulty or does not support features the controller application expects. On the other hand, if multiple real devices uncover failures the model is not able to detect, there is likely a problem with the controller. Additionally, the model should be fixed as it does not simulate a real switch correctly. Another hint that a model is wrong is when it reports problems that real devices do not detect.

D. Discussion

Approach limitations. The approach we take entails a few limitations. Some are due to using model checking, while others are consequences of the level of interaction with the switches we choose. First, model checking forces sequential execution of transitions. Therefore, it does not put tested devices in a high load situation, thus missing problems that may arise under stress conditions or when performance bottlenecks are hit. For the same reason, it

is not possible to observe issues related to many events happening within a short time interval. In fact model checking explores transitions one by one and checks correctness properties after each transition. Thus, there is a time gap between each pair of events in the switch. Additionally, such lack of control over time means that we are not able to exercise timers. When it is possible OFTEN emulates them (for example installing persistent flows and then sending explicit flow remove command instead of relying on timer induced removal). Finally, as all packets and messages stay in the network model, the switch buffers are never full. Thus, OFTEN does not detect issues visible only when many packets are waiting for processing.

Alternative testbed design. In our design each switch is directly connected to the testing harness. This way the network needs to be rewired before testing. However, there are other solutions that improve system scalability for running with an unmodified network but they introduce additional time overhead in exchange. In such a design, switches stay connected to each other on a data-plane level. The control-plane communication channel is directly connected to the OFTEN harness. End hosts are also emulated by the harness. In this setup, each transition on a switch (S) consists of additional preliminary and cleaning phases. During the preliminary phase, OFTEN installs the highest priority send-all-to-the-controller rule at all switches directly connected to S. As a result, all packets leaving S are captured by OFTEN testing harness as Packet_In messages. To inject a data plane packet, OFTEN sends a Packet_Out message containing this packet to the correct neighbor of S. After the transition ends, all rules installed during the preliminary phase are removed. Communication with the controller stays unchanged.

Desirable switch features. Most challenges that OFTEN solves are related to the fact that OpenFlow is not designed with testability in mind. Although, as we show, it is possible to use features specified by OpenFlow to conduct testing, the introduction of just a few additional mechanisms can make the process much simpler. First, like a barrier request forcing synchronization for control messages, there should be a similar way to determine when all data-plane packet buffers are empty and no packets are being processed. Additionally, switches may expose a standardized interface to get detailed information about the internal state (*e.g.*, contents of buffers with packets sent to the controller). Finally, determining when packet processing ends and obtaining the switch state require sending additional OpenFlow messages. This could have side effects on the tested OpenFlow agent, and therefore, should be avoided. As there already is a management and configuration protocol for OpenFlow [12], a similar debugging protocol may be introduced that would define a standardized, but external to OpenFlow, method for acquiring information useful in troubleshooting a switch. Such a protocol, supported by switches and specifying all required features would make testing easier and help to increase the reliability of OpenFlow networks. We will consider this direction as part of our future work.

IV. Preliminary results

Using a preliminary OFTEN prototype, we tested three real OpenFlow applications that were previously tested with NICE [5]: a MAC-learning switch, energy-aware traffic engineering and a server load-balancer. In all cases we used the OpenFlow reference switch implementation (most current version as of Apr 2012), OpenVSwitch (version 1.0.0) and an HP ProCurve E5406zl switch with OpenFlow support (firmware release K.15.05.5001). OFTEN identified two new issues.

Issue #1. OFTEN reveals one issue in the load-balancer application that was previously undetected by NICE. According to the new OpenFlow specifications [1], the maximum length field of the send-to-controller action always means the number of bytes to send. On the other hand, until version 0.8.9, the specification defines zero as a special value with the semantic of sending the whole packet. The switch model ignored this field, always sending the whole packet. When used with real switches, the load-balancer was losing packets (*NoForgottenPackets* property violation) during its reconfiguration phase. Further investigation shows that the load-balancer, during reconfiguration, installs send-to-controller rules with maximum length field set to zero and expects to receive the whole packet. Therefore it is incompatible with OpenFlow specification versions later than 0.8.9.

Issue #2. The in-port field of Packet_In messages set by the switch model was incorrect when following the send to controller action. However, this bug did not affect property violations for any of the controllers. It was reported as a warning and led us to improve the switch model.

V. Related work

Testing OpenFlow switches OFTest [6] is a unified framework used to test correctness of OpenFlow switches. To provide high coverage, developers need to design a large number of test cases each of which targets a specific feature. Additionally, there are usually tests developed with each specific OpenFlow switch version provided by its creators.

OFLOPS [7] is a performance testing tool for OpenFlow switches. It detects problems appearing when the device is heavily loaded, but is also capable of discovering some inconsistencies between data-plane and control-plane undetectable in other ways.

Debugging a network Mininet [13] helps in prototyping the network design. It allows to create a virtual network consisting of a controller and multiple switches on one machine. However, it does not enable systematic testing nor does it provide low level correctness properties. The simulation testbed developed by Google [14] uses a similar approach. It uses real binaries of both controller and OpenFlow agents

while hardware and network are virtualized. Although it can run multiple scenarios covering the entire network, it does not target such a fine-grained systematic scenario exploration as OFTEN.

There are also tools that aid in debugging a deployed OpenFlow network. Such an approach accepts that bugs affect the running network, while OFTEN detects them before deployment. OFRewind [15] records traces of events in a running network. When the network operators realize incorrect behavior, OFRewind helps to locate its root cause by replaying filtered traces. NDB [16] is a similar debugging tool working on a finer granularity level.

VeriFlow [9] also works in a deployed network, but it detects violations of safety properties upon rule modification messages issued by the controller, preventing the update from affecting the network.

Violations of basic network-wide properties may be detected using the static data-plane configuration analysis technique proposed in [8]. Depending on the size and complexity of the network this may be either an online or offline tool.

VI. Conclusion and Future Work

Despite the detailed specification and assumed separation of concerns in OpenFlow networks, the behavior of an integrated network might still be unexpected. In this paper we argued for the necessity of having a network-wide testing tool for OpenFlow. We identified the challenges that have to be addressed when creating such a tool, and proposed the initial approach for tackling them. We developed a prototype of the tool that approaches the problem from one point of view, and concentrates on systematic testing that achieves high state-space coverage. We plan to further develop and improve the tool to utilize other techniques suitable in different scenarios.

OpenFlow is an emerging technology, which gives its designers an opportunity to consider testability as one of the key requirements. Our discussion shows that some problems may be easily solved by including appropriate features in the switch specification.

Acknowledgment

We are grateful to Jennifer Rexford for useful discussions and comments on earlier drafts of this work. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110.

References

[1] "OpenFlow Switch Specification," http://openflow.org/documents/openflow-spec-v1.1.0.pdf.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008.

[3] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, July 2008.

[4] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A Distributed Control Platform for Large-scale Production Networks," *OSDI*, 2010.

[5] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE Way to Test OpenFlow Applications," in *NSDI*, 2012.

[6] "OFTest," http://oftest.openflowhub.org.

[7] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation," in *PAM*, 2012.

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

[9] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide Invariants in Real Time," in *HotSDN*, 2012.

[10] "HP Switch Software, version 1.15.06.5008, OpenFlow Supplement," http://h20000.www2.hp.com/bc/docs/support/SupportManual/c03170243/c03170243.pdf.

[11] L. Hatton, "N-version design versus one good version," *IEEE Software*, vol. 14, no. 6, pp. 71–76, Nov. 1997.

[12] "OpenFlow Management and Configuration Protocol," http://opennetworking.org/images/stories/downloads/of-config/of-config-1.1.pdf.

[13] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *HotNets*, 2010.

[14] "OpenFlow @ Google," Open Networking Summit 2012.

[15] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *USENIX ATC*, 2011.

[16] B. Heller, N. Handigol, V. Jeyakumar, N. McKeown, and D. Mazieres, "Where is the debugger for my software-defined network?" in *HotSDN*, 2012.