Towards Model based Testing for Software Defined Networks

Asma Berriri[1], Jorge López[1], Natalia Kushik[1], Nina Yevtushenko[2,3] and Djamal Zeghlache[1]
[1]*SAMOVAR, CNRS, Télécom SudParis, Université Paris-Saclay, 9 rue Charles Fourier, 91000 Évry, France*
[2]*Department of Information Technologies, Tomsk State University, 36 Lenin street, 634050 Tomsk, Russia*
[3]*Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25 Alexander Solzhenitsyn street, 109004, Moscow, Russia

Keywords: Software Defined Networks, Formal Methods, Testing, Equivalent Classes, Graph Enumeration.

Abstract: Software Defined Networks (SDNs) and corresponding platforms are expected to be widely used in future generation networks and especially deployed and activated on-demand as agile networking control service components. The correct functioning of SDN platforms must be assured, i.e., such platforms should be thoroughly tested before deployment. After thorough verification of SDN controllers and switches, the composition of them still requires additional testing in order to guarantee the absence of critical faults. We propose a model based testing technique for checking SDN platforms that relies on *appropriate graph enumeration*. In particular, we define a fault model where the fault domain contains the wrongly and correctly implemented paths allowed with respect to the underlying resource connectivity graph. We also establish the conditions for deriving a complete test suite with respect to such fault model under black box and white box testing assumptions.

1 INTRODUCTION

Software Defined Networks (SDNs) enable logically centralized control over network devices through a controller that operates independently of the network hardware, and can be viewed as the *network's operating system* (Sezer et al., 2013). As a result, SDN provides a flexible controllability and programmability by separating the control and data planes via introducing an open interface between these planes.

The improvements in the control and management of networks and networking services do not, however, remove the need to address thoroughly the risk of misconfigurations or software bugs that can lead to network failures (Gill et al., 2011). Moreover, due to the flexibility provided by SDNs, it is essential to guarantee the correct (re-)configuration of the paths for network flows. To ensure functional correctness of SDNs and reduce the risk of misconfigurations, programmable devices and components in an SDN framework must be continuously tested. However, even if each component is well debugged in isolation their composition can still face interoperability issues. One of the solutions to check the framework functionality in this case is to define methods and techniques for testing the system as a whole. These methods would allow administrators to estimate the correctness of their SDN solutions and validate them before deployment. Hereafter, we focus on testing functional aspects of SDN frameworks, i.e., in this paper, we do not consider non-functional SDN issues, such as security, trust, etc.

We note that a number of techniques for SDN verification and testing have already been developed. In this paper, we focus on so called active testing when a system under test (SUT) is stimulated by appropriate inputs, i.e., test sequences / cases, and the conclusion about its correctness is made based on observations of its output responses. Formal verification usually does not require any stimulation of the SUT; it is mainly reduced to the creation of a set of properties that are iteratively checked at different phases of the software development / life cycle. In the last decade, SDN verification techniques have been largely investigated and thus, cover a big number of possibilities, starting from the header space analysis of network packets and ending with intricate and complicated model checking problems that can be handled by SAT solving (Mai et al., 2011) or deductive verification and theorem provers (Guha et al., 2013). These approaches differ in the way the verification is performed, namely off-line and on-line (run-time) verification can be considered.

Berriri, A., López, J., Kushik, N., Yevtushenko, N. and Zeghlache, D.
Towards Model based Testing for Software Defined Networks.
DOI: 10.5220/0006805604400446
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 440-446
ISBN: 978-989-758-300-1

Existing testing approaches[1] for SDN infrastructures can be divided into three groups. The first group aims at creating 'problematic' situations for the network or any SDN component. In this case, the inputs to be applied are special (unexpected or rarely applied) requests; the test assessment is given depending on the ability of the controller (or the switch) to process the request correctly or reject it (Scott et al., 2015; Shalimov et al., 2013). The second group of techniques concerns conformance testing when a formal specification of an SDN component is derived (Yao et al., 2014; Zhang et al., 2016) and the test suite is generated on the basis of this specification. The approaches of the last group represent a combination of formal verification and testing techniques, namely model checking solutions are effectively utilized for test suite derivation (see for example, (Canini et al., 2011; Canini et al., 2012)).

We note however, that existing testing techniques and particularly model based testing techniques have been mostly applied to the SDN components, and not to the entire SDN system (framework itself). This motivates us to provide novel formal solutions to comprehensively test the SDN system as a whole.

Consequently, we propose a model based testing technique for checking SDN frameworks that relies on *appropriate graph enumeration*. In particular, we define a fault model where the fault domain contains potential implementations of virtual paths requested by a user. To guarantee the fault coverage, we prove the conditions when under black box and white box testing assumptions a complete test suite with respect to such fault model can be derived.

The paper is structured as follows. Section 2 of this paper presents the necessary background. Novel formal SDN testing approaches based on the appropriate graph enumeration and preliminary experimental results are presented in Section 3. Section 4 concludes the paper and describes future research challenges.

2 BACKGROUND

A fundamental characteristic of the SDN architecture is the separation of the control plane from the forwarding (data) plane (Opennetworking, 2012). A logically centralized control function maintains the state of the network and provides instructions to the data plane. The network devices in the data plane then forward data packets according to these control instructions, more specifically, forwarding and filtering rules (Sezer et al., 2013). A forwarding rule is conformed by three parts: a packet matching part, an action part and a location / priority part. The matching part describes the values which a received network packet should have for a given rule to be applied. The action part is simply the disposition of the matched network packets; the location priority part controls the hierarchy of the rules using tables and priorities.

An SDN controller has a 'global view' of the network and several packet forwarding devices are controlled and configured remotely and dynamically through interfaces using protocols such as the Open Flow (OF) protocol (McKeown et al., 2008). SDN-enabled switches contain one or more forwarding (flow) tables which are ultimately managed by a controller or a cluster of them, logically representing a single controller; SDN switches receive the forwarding rules from controllers to steer network packets in the data plane. An example of an SDN architecture is depicted in Fig. 1. In particular, Fig. 1 presents an example of a network topology consisting of two controllers, four switches and four hosts. Each switch is connected to a single host as well as to one of the SDN controllers.

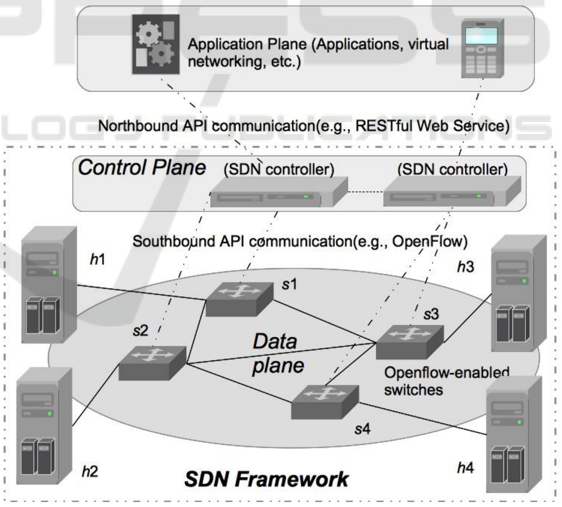
Application Plane (Applications, virtual networking, etc.)
Northbound API communication(e.g., RESTful Web Service)
Control Plane
(SDN controller)
(SDN controller)
Southbound API communication(e.g., OpenFlow)
h1
h2
h3
h4
s1
s2
s3
s4
Data plane
Openflow-enabled switches
SDN Framework

Figure 1: Example of an SDN architecture.

As mentioned before, in order to assure the quality of the SDN architectures, one of the possibilities is to define methods and techniques for testing such complex composite systems.

Hereafter, under testing techniques, we assume an *active* testing mode where test cases or test sequences are applied to a system under test, and the conclusion about the correctness of the SUT is made based on observations of the output responses to the test sequences. In this case, the *observed outputs* are generated

[1]Apart from those, developed for 'classical' communication networks that can be also applied to SDN data planes (see, for example (Zeng et al., 2012)).

upon applying a corresponding test sequence to the system under test and (automatic) traffic generation allows to make necessary observations, such as correct or wrong configurations, correct or wrong flow tables, etc. The quality of a test suite is usually measured by its fault coverage, i.e., the types and number of faults that can be detected with the test suite. Model based testing techniques seek for test suites with guaranteed fault coverage that can be stated as (necessary and) sufficient conditions for a test suite *completeness*. Such conditions can be proven when a corresponding *fault model* is introduced. In this paper, a fault model is represented by a pair $\langle @, FD \rangle$, where @ is a conformance relation (between what is requested and what is really implemented) and FD is a set of potential implementations. Each correct implementation $I_1 \in FD$ passes a *complete* test suite while each faulty implementation $I_2 \in FD$ (with respect to @) fails such a test suite.

3 MODEL BASED TESTING FOR SDN ARCHITECTURES

3.1 Notations

In this paper, we refer to the data plane as the *resource network connectivity topology* (RNCT[2]), depicting the SDN elements in the resource network. An RNCT is represented by an undirected (network links are assumed to be bidirectional) and k-colored graph $G = (V, E, c)$, where the set V of nodes represents network devices (switches, hosts, etc.). Edges of the graph (the set E) are unordered pairs (x, y), $x, y \in V$, representing connections between two nodes (links) in a network; and c is a coloring function $c : V \mapsto \mathbb{N} \cup \{0\}$ such that given a node in the network, a corresponding color is assigned to it as a hashed integer. Note that the colors of adjacent nodes can be the same, differently from the common graph coloring functions. As an example, the previously depicted model can accurately represent the data plane (RNCT) shown in Fig. 1 by the following binary-colored graph: $RNCT = (V, E, c)$, where:

$$V = \{s_1, s_2, s_3, s_4, h_1, h_2, h_3, h_4\}$$

$$E = \begin{Bmatrix} (h_1, s_1), (h_2, s_2), (h_3, s_3), (h_4, s_4), (s_1, s_2), \\ (s_1, s_3), (s_2, s_4), (s_2, s_3), (s_3, s_4) \end{Bmatrix}$$

$$c(v) = \begin{cases} 1, & \textit{if } v = s_1 \vee v = s_2 \vee v = s_3 \vee v = s_4 \\ 0, & \textit{otherwise} \end{cases}$$

[2]Hosting infrastructures can be physical or virtualized.

Note that in the above example, '1' represents a *switch color*, and '0' represents a *host color*, correspondingly.

Issuing a forwarding rule to an SDN-enabled switch creates a virtual link from and to other node(-s) adjacent to the switch if the rule forwards traffic to a given port. For example, assume that for switch s_1 shown in Fig. 1 h_1 is connected to port 1 and s_2 is connected to port 2. The rule "table=0, priority=99, in_port=1, dl_dst=01:80:c2:00:00:00, actions=resubmit(,2)" creates a virtual link $h_1 \to s_1 \to s_2$ if the destination mac address is 01:80:c2:00:00:00. Formally, the application of a forwarding rule creates a virtual link $a \in E^*$ as a sequence of *directed* edges from the RNCT edges. In this paper, we consider a virtual path (simply a path throughout the paper) as a sequence of directed edges whose head and tail nodes are hosts and all other intermediary nodes are switches. The reason is that for testing purposes, observing traffic generated from one host to another is how the resulting configuration is collected as an 'output'. We assume the hosts in the RNCT do not act as switches or relays of network packets, furthermore, we assume switches do not act as hosts in the RNCT. Even if at a physical level the previous cases are possible, the RNCT model must not consider such possibilities. The forwarding rules used to control the traffic in the RNCT construct a virtual partial path or a set of those. Some examples of the paths obtained from an RCNT (as depicted in Fig. 1) are illustrated in Fig. 2. In this case, the RNCT is a dense network with four switches. Each switch is connected to a single host as well as to one controller. We note that edges cannot be repeated in a path, otherwise, infinite loops can be potentially formed. Furthermore, in this paper we consider also that nodes cannot be repeated, studying this possibility is left for future work. Validating the correctness of

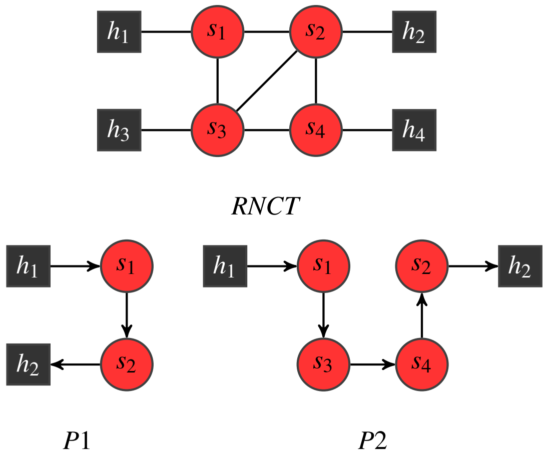

Figure 2: RNCT and examples of its paths.

the requests is also out of the scope of this paper (the interested reader can refer to (López et al., 2017)).

3.2 Introducing an SDN Fault Model

Given the SDN infrastructure in Fig. 1, we propose to test the SDN framework, including the controller(-s), switches, and connections between them. Namely, we propose to derive an application that is responsible for test generation and execution, i.e., a tester that sends specific requests to the SDN controller asking for different paths to be implemented in the RNCT.

The test generation architecture is illustrated in Fig. 1, where the application layer is executing only the tester. According to our assumptions, the inputs that need to be generated by the orchestrator in order to guarantee that the SDN infrastructure is functioning properly are paths limited by the RNCT.

We assume that the SDN infrastructure is *functioning correctly* when each requested path and only it is created. As in fact, mostly connectivity issues are tested, similar to (El-Fakih et al., 2004), the fault model has two items, i.e., the fault model is a pair $\langle =, FD \rangle$ where the conformance relation is the equality. Another issue is about the fault domain FD of the fault model. According to the path definition, the following types of faults can be considered: a requested edge can be directed to a wrong node, additional edges can appear as well as some edges can disappear. Thus, a fault domain FD contains all possible paths of the RNCT. A *test case* is a path of the RNCT and a *test suite* is a finite set of paths. As usual, a test suite is *complete* w.r.t. $\langle =, FD \rangle$ if any difference between a requested and implemented path can be detected.

Checking the output reaction of the SDN infrastructure can be performed through a network traffic initiation. As we try to check that the paths are implemented correctly, we focus on specific traffic generation. For generating traffic, we propose to use the ICMP echo request / echo reply packets through the known ping utility[3]. The ICMP request / reply is performed for each pair of hosts that correspond to the *head* and *tail* nodes of the test cases. Later, the passing traffic is inspected at all node interfaces via a simple network sniffer. The network traffic of all switches can be obtained in different ways, starting from a simple Unix-like sniffer as the tcpdump utility and finishing with non-software-based (physical / vendor) switches via protocols such as NetFlow (Claise, 2004) or sFlow (Phaal et al., 2001). As an example, consider the requested path $(h_1, s_1)(s_1, s_3)(s_3, s_4)(s_4, s_2)(s_2, h_2)$ with respect to the RNCT in Fig. 1. The corresponding traffic generation (through ICMP echo request) and the traffic observation are depicted in Fig. 3. We depict the messages and a timestamp when the message was observed. Hereafter, 't1' denotes the first time instance after traffic generation started. If the network flow follows the requested path, i.e., the expected output response is observed during the traffic generation, we consider that the test case has *passed*. An example of traffic generation and observation is shown in Fig. 3.

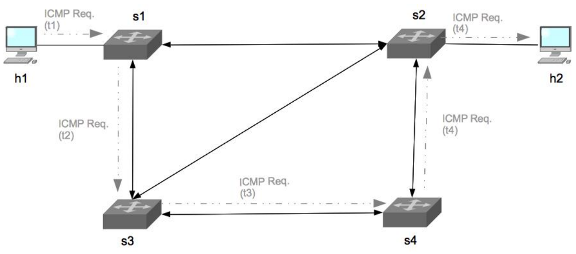

Figure 3: Traffic generation and flow observation.

After the fault model is set, usual testing approaches can be used for deriving test suites with the guaranteed fault coverage. Below, we discuss how black and white test derivation approaches can be used for this purpose.

3.3 Black Box Testing Approach

As the set of all paths of the RCNT is finite, the simplest way to construct a complete test suite w.r.t. $\langle =, FD \rangle$ is to consider the set of all such paths.

Proposition 1. *The set of all RCNT paths is a complete test suite w.r.t.* $\langle =, FD \rangle$.

In general, this test suite is rather long and we propose an approach for reducing its length based on equivalence classes. Indeed, if we assume that each node processes inputs independently of the node where they come from, two paths can be considered (i, j)*-equivalent* if both paths have a directed edge from node i to node j. That is either all the packets that should be directed from i to j are either processed correctly, i.e., are sent from i to j, or are processed wrongly, i.e., are sent anywhere except the j-th node. In Fig. 2, paths $P1 = (h_1, s_1)(s_1, s_2)(s_2, h_2)$ and $P2 = (h_1, s_1)(s_1, s_3)(s_3, s_4)(s_4, s_2)(s_2, h_2)$ are considered to be equivalent w.r.t. the edge (h_1, s_1) as well as w.r.t. the edge (s_2, h_2) .

Proposition 2. *The set of paths that contains a path of each* (i, j)*-equivalent class where* (i, j) *is an edge in the RCNT, is a complete test suite w.r.t. the fault model* $\langle =, FD \rangle$.

[3]We however note the existence of different approaches for automatic traffic generation (see, for example (Zeng et al., 2012; David et al., 2014; Fayaz et al., 2016)).

Indeed, a complete test suite has at least one request where a packet should be sent from node i to node j. If the packet is processed correctly (according to the monitoring results), then due to the testing assumptions, we conclude that each packet directed from node i to node j will be sent to the j-th node. □

By direct inspection, one can assure that the number of all paths for the RNCT example in Fig. 2 equals 36. However, the proposed equivalence classes approach allows to reduce this test suite up to ten paths only.

In order to cover all equivalence classes in an optimal way, an optimization problem should be stated and solved. One option is to consider the Boolean (weighted) matrix and solve the corresponding covering problem (Villa et al., 1997) for which many libraries and scalable software solutions are developed. If the node processes a packet depending on where it comes from then equivalence classes could be considered w.r.t. path subsequences of length $l \geq 2$. Given a sequence γ of RNCT edges of length l between node i and node j, two paths are considered γ-equivalent if they both contain γ. A test suite is complete if it has at least one path of each equivalence class. A minimal cover of a corresponding Boolean matrix can also be used for optimal test generation.

3.4 White Box Testing Approach

In some cases, mainly for reducing testing complexity, it may be desired not to generate test suites w.r.t. all possible edges in the RNCT. The complexity can be reduced if a set of *critical edges* that need to be tested first can be defined; for example, critical edges that include critical services. For this reason, we propose Algorithm 1 that generates a test suite with guaranteed fault coverage w.r.t. a set of critical edges, i.e., if a fault occurs at a given 'critical' edge, it is detected. The algorithm is based on generating a test case $tc = (v_1, v_2) \dots (v_i, v_j)(v_j, v_{j+1}) \dots (v_{n-1}, v_n)$ that traverses a critical edge (v_i, v_j) for all critical edges E'. We consider in the fault domain, implementations that can potentially contain three types of faults that need to be detected, namely:

1. An edge is directed to a wrong node, i.e., from the edge of interest (v_i, v_j) to $(v_i, v_{j'})$ where $j \neq j'$.
2. An edge $e = (v_i, v_j)$ is deleted.
3. A non-existing edge (v_i, v_k) is created for a critical edge (v_i, v_j), where $j \neq k$.

As potential faults are enumerated explicitly, Algorithm 1 returns the test suite under the white box testing assumption. By construction, the following proposition holds.

```
Input  : $RNCT = (V, E, c)$, a binary-colored
         graph where hosts are "0" colored;
         $E' \subseteq E$, a set of critical edges
Output: A test suite $TS_w$
$TS_w \leftarrow \emptyset$;
foreach $f = (v_i, v_j) \in E'$ do
    1. Find (backtrack)
       $p_b = (v_1, v_2) \dots (v_i, v_j)$, the shortest
       sequence of edges such that $c(v_1) = 0$,
       i.e., the shortest sequence that starts in a
       host and finishes at the node $v_j$. Note
       that if $c(v_i) = 0$ then $p_b = (v_i, v_j)$;
    2. Find (forwardtrack)
       $p_f = (v_i, v_j) \dots (v_{n-1}, v_n)$, the shortest
       sequence of edges such that $c(v_n) = 0$,
       i.e., the shortest sequence that finishes in
       a host and starts at node $v_j$. Note that if
       $c(v_j) = 0$ then $p_f = (v_i, v_j)$;
    3. $TS_w \leftarrow TS_w \cup \{p_b p_f\}$;
return $TS_w$
```

Algorithm 1: White box Complete Test Suite Generation

Proposition 3. *Algorithm 1 returns a complete test suite w.r.t. the fault model $\langle =, FD \rangle$ where FD has each path with a critical edge.*

We note that Algorithm 1 is somehow a *naive* algorithm. However, a test minimization can be performed similar to black box testing approach, via solving a covering problem. In this case, a minimal set of paths that cover all critical edges can be identified.

3.5 Preliminary Experimental Results

In order to simulate the resource infrastructure, i.e., to provide the RNCT, we utilized the well known Mininet (Mininet, 2013) simulator executed under an Ubuntu 14.04 LTS virtual machine running on VirtualBox Version 5.1.14 r112924 (Qt5.6.2) for Mac OS X, with 2GB of RAM and 1 core of a 2.3 GHz Intel Core i5. Our preliminary experimental setup is a network containing 4 switches running Open vSwitch version 2.0.2 (also 4 hosts, and 9 links), as depicted in Fig. 1. We note that even if the experiments were not performed at a large scale, the test derivation techniques do not have high complexity and can thus be applied to larger environments. Furthermore, to prove the validity of our approach, real SDN controllers were utilized for the experiments, especially an OpenDaylight (ODL) (Medved et al., 2014) Boron-S3, and an Onos (Berde et al., 2014) 1.10.4 running under a dedicated CentOS 7 virtual machine with 8 cores and 12GB of RAM.

Considering Proposition 2, we derived a test suite TS for (i,j)-equivalence classes, including each edge (i,j) at least once. When the test suite was executed against the SDN framework composed by the experimental setup and the ODL controller, all tests failed. The test suite has been executed by requesting the proper flow instantiation via the ODL REST interface. In fact, none of the requested paths of the test suite were implemented. Nevertheless, the controller gave positive replies (HTTP 201 - created) to the creation of all individual flow entries, and none were installed in the Open vSwitches. Certainly, positively replying to a request for flow creation and not implementing it is not considered to be a correct functionality, independently of any potential misconfigurations. On the other hand, when the test suite was executed against the SDN framework with the experimental setup using the Onos controller all tests successfully passed.

TS is indeed a complete test suite with respect to the presented fault model $\langle =, FD \rangle$, and therefore, its execution against an SDN platform (implementation) provides a guarantee regarding the correct functioning of that SDN platform, if the test suite passes it. Nonetheless, it is interesting to check the fault detection effectiveness of each test sequence α in the test suite TS (test sequence 'power'). For this reason, we deliberately introduced a bug in the Onos controller to provide positive replies to the creation of flows which are not installed on the devices. To obtain a faulty implementation with the previously described bug, a single statement was deleted from the Onos controller's source code [4]; particularly, the statement was deleted in the FlowsWebResource.java file. As Onos compiles with regression tests, a special compilation process ignoring such tests was executed. Given the modified code for the Onos controller, the fault coverage of each test sequence in the test suite was assessed. As a result, all the test sequences failed when being executed against the modified SDN platform, i.e., each test case is 'capable' of detecting the introduced bug by its own. Therefore, these preliminary experiments also showcase the power of the obtained test cases and motivate to consider the test minimization in the future.

Although these experiments were not performed for realistic SDN infrastructures, some conclusions can be drawn. First, the SDN architecture composed by the experimental setup together with the Onos controller *is guaranteed* to be free of faults of the wrong redirection, edge deletion and edge creation faults as considered in the fault domain. Moreover, the obtained test suite can detect other types of bugs (for example, a single statement deletion) for which the relationship with the errors listed above needs to be further investigated. Second, the SDN architecture composed by the experimental setup together with the ODL controller seems not to be free of the bugs under consideration. Note that even if the second conclusion may be considered as trivial, the proposed approach can be seen as a helpful mechanism to 'certify' the correct functionality of a given SDN infrastructure under certain conditions and assumptions.

[4]Statement deletion is often considered in mutation testing (Deng et al., 2013) when the test suite quality is estimated.

4 CONCLUSIONS

In this paper, we focused on testing techniques for checking the functionality of SDN frameworks. As the inputs of the SDN infrastructures are user-defined paths, we proposed a formal approach for effective test generation for such non-trivial inputs, namely, specific graph enumeration techniques have been discussed for testing an SDN framework.

To formally prove the fault coverage, we defined a fault model where the fault domain contains different implementations of the requested paths. We also established the conditions when a complete test suite with respect to such fault model can be derived.

As a future work we plan to investigate other fault models, for example, we would like to check different abstraction levels, i.e., to consider not a single path as a test case but a set of those. Another direction for future work is the test suite minimization for black and white box testing assumptions, namely to choose the minimal number of paths of interest so that the test suite fault coverage is preserved. Finally, we plan to perform experiments on a number of (distributed) SDN architectures for estimating the effectiveness of graph enumeration for detecting real bugs in the code of switches / controllers.

ACKNOWLEDGEMENTS

The results in this work were partially funded by the Celtic-Plus European project SENDATE, ID C2015/3-1 and the Russian Science Foundation (RSF), project # 16-49-03012.

The authors gratefully acknowledge senior researchers Igor Burdonov and Alexander Kossatchev for fruitful discussions.

REFERENCES

Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al. (2014). Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM.

Canini, M., Kostic, D., Rexford, J., and Venzano, D. (2011). Automating the testing of openflow applications. In *Proceedings of the 1st International Workshop on Rigorous Protocol Engineering (WRiPE)*.

Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. (2012). A nice way to test openflow applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, number EPFL-CONF-170618.

Claise, B. (2004). Cisco systems netflow services export version 9.

David, L., Stefano, V., and Olivier, B. (2014). Towards test-driven software defined networking. In *2014 IEEE Network Operations and Management Symposium*, pages 1–9.

Deng, L., Offutt, J., and Li, N. (2013). Empirical evaluation of the statement deletion mutation operator. In *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pages 84–93.

El-Fakih, K., Trenkaev, V., Spitsyna, N., and Yevtushenko, N. (2004). FSM based interoperability testing methods for multi stimuli model. In *Testing of Communicating Systems, 16th IFIP International Conference, TestCom 2004, Oxford, UK, March 17-19, 2004, Proceedings*, pages 60–75.

Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V. (2016). BUZZ: testing context-dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation*, pages 275–289.

Gill, P., Jain, N., and Nagappan, N. (2011). Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA.

Guha, A., Reitblatt, M., and Foster, N. (2013). Machine-verified network controllers. In *ACM SIGPLAN Notices*, volume 48, pages 483–494. ACM.

López, J., Kushik, N., Yevtushenko, N., and Zeghlache., D. (2017). Analyzing and validating virtual network requests. In *The 12th International Conference on Software Technologies (ICSOFT), Madrid, Spain*, pages 441–446.

Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. (2011). Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

Medved, J., Varga, R., Tkacik, A., and Gray, K. (2014). Opendaylight: Towards a model-driven sdn controller architecture. In *Proceedings of the IEEE 15th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–6.

Mininet (2018). Mininet: An instant virtual network on your laptop (or other pc)-mininet.

Opennetworking (2012). Software-defined networking: The new norm for networks. *ONF White Paper*.

Phaal, P., Panchen, S., and McKee, N. (2001). Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks.

Scott, C., Wundsam, A., Raghavan, B., Panda, A., Or, A., Lai, J., Huang, E., Liu, Z., El-Hassany, A., Whitlock, S., et al. (2014). Troubleshooting blackbox sdn control software with minimal causal sequences. *ACM SIGCOMM Computer Communication Review*, 44(4):395–406.

Sezer, S., Scott-Hayward, S., Chouhan, P. K., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., Miller, M., and Rao, N. (2013). Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43.

Shalimov, A., Zuikov, D., Zimarina, D., Pashkov, V., and Smeliansky, R. (2013). Advanced study of sdn/openflow controllers. In *Proceedings of the 9th central & eastern european software engineering conference in russia*. ACM.

Villa, T., Kam, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1997). Explicit and implicit algorithms for binate covering problems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(7):677–691.

Yao, J., Wang, Z., Yin, X., Shiyz, X., and Wu, J. (2014). Formal modeling and systematic black-box testing of sdn data plane. In *The IEEE 22nd International Conference on Network Protocols (ICNP)*, pages 179–190.

Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. (2012). Automatic test packet generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, pages 241–252.

Zhang, Z., Yuan, D., and Hu, H. (2016). Multi-layer modeling of OpenFlow based on EFSM. In *4th International Conference on Machinery, Materials and Information Technology Applications*, pages 524–529.