Extended Finite State Machine based test generation for an OpenFlow switch

Asma Berriri, Natalia Kushik, Djamal Zeghlache

Extended Finite State Machine based Test Generation for an OpenFlow Switch

Asma Berriri[1] , Natalia Kushik[1] and Djamal Zeghlache[1]
[1]*SAMOVAR, CNRS, Télécom SudParis, Université Paris-Saclay, 9 rue Charles Fourier, 9100 Évry, France*
{*asma.berriri, natalia.kushik, djamal.zeghlache*}*@telecom-sudparis.eu*

Keywords: Software Defined Networking (SDN), OpenFlow Switch, Formal Methods, Testing, Extended FSM.

Abstract: Implementations of an OpenFlow (OF) switch, a crucial Software Defined Networking (SDN) component, are prone to errors caused by developer mistakes or/and ambiguous requirements stated in the OF documents. The paper is devoted to test derivation for related OF switch implementations. A model based test generation strategy is proposed. It relies on an Extended Finite State Machine (EFSM) specification that describes the functional behaviour of the switch-to-controller communication while potential faults/misconfigurations are expressed via corresponding mutation operators. We propose a method for deriving a test suite that contains distinguishing sequences for the specification EFSM and corresponding mutants. The proposed approach is implemented as a testbed to automatically derive and execute the test suites against different versions of an OF implementation. Preliminary experimental evaluation has shown the effectiveness of the proposed approach. Further on, the derived test suites have been able to detect a number of functional inconsistencies such as erroneous responses to the $Flow\_Mod$ adding rules with specific 'action' values in an available Open vSwitch implementation.

1 INTRODUCTION

Software Defined Networking is a new networking paradigm where a logically centralized controller orchestrates a distributed set of switches to provide high level networking services to end-host applications. An OpenFlow switch acts as a forwarding device receiving and sending network packets in accordance with a set of configured rules through the data plane interface. It also sends events such as traffic statistics, topology changes and acknowledgments to the controller. The controller can (re-) configure the switches through network commands such as installing new rules. This switch-to-controller interaction is performed through OF protocol, it enables the implementation of network applications' policies such as routing and load balancing. Nonetheless, the OF specification is extremely complex and lacks uniform standardization. For example, just the rule installation command ($Flow\_Mod$) is more than two pages long (Open Networking Foundation, 2014). This might increase misinterpretation or cause conflicts due to multiple or duplicated requirements.

To address this challenge, in this paper, we propose an EFSM based technique for test generation that aims to verify the OF switch-to-controller interaction with respect to requirements described in the OF specification. Correspondingly, the EFSM model is derived based on the OF requirements. Potential *incorrect* implementations are also modelled as EFSMs that are obtained by injecting specific types of (user-driven) faults into the model, i.e., through mutants generation. For each mutant, a *distinguishing sequence* (*DS*) is sought that separates the original specification from the mutated one. We present an effective algorithm that derives a test suite TS formed by the corresponding *DS*s.

Several approaches have been proposed in the literature for deriving conformance tests when the system specification is represented by an EFSM. On one hand, a group of these works have proven to be effective in deriving tests with guaranteed fault coverage. For example, these approaches have been proposed in the works by Bochmann et al. (Bochmann and Petrenko, 1994), El-Fakih et al. (El-Fakih et al., 2003), and Petrenko et al. (Petrenko et al., 2004). However, the authors are not aware of the corresponding techniques being applied to tackle the correctness of SDN-enabled components. On the other hand, another group of works relying on EFSM for test generation have been proposed for SDN application area (Yao et al., 2014b), (Zhang et al., 2016), however, no

related fault model has been introduced and thus, no fault coverage has been proven (see Section 2).

In this paper, we propose to lessen this gap and advance the state of the art research on model based testing applied to SDN. An effective heuristic approach to derive distinguishing sequences for the specification EFSM and its mutants is presented for testing SDN switch-to-controller communication.

The main contributions of this paper are the following. Firstly, we formalize a part of the OF requirements and propose an EFSM based testing approach applied to an OF switch. Secondly, to demonstrate the effectiveness of the proposed approach, an experimental evaluation is performed which aims at the assessment of the derived test suites fault coverage on one hand and at the execution of the derived tests against an OF implementation under test, on the other hand. To this end, a testbed is developed which aims at the automation of the generation and execution of the derived tests against an OF implementation under test.

The conducted evaluation has shown on the one hand the effectiveness in terms of fault coverage of the derived test suites. Indeed, compared to randomly generated test suites, the ones derived by our approach have shown an average mutation score of 60.07% against 21.75% for the randomly generated TSs. On the other hand, experiments have revealed several implementation faults and specification ambiguities when we have tested a switch implementation, namely Open vSwitch 2.5.0. Examples of detected faults include several misbehaviours when rules with some specific values of the 'action' field of the $Flow\_Mod$ input have been installed, in addition, a misbehaviour in updating the statistics about the installed rules has been observed.

The paper is structured as follows. Section 2 presents related work. Section 3 provides the background information. Section 4 gives an overview of the EFSM model of the switch. The test generation process is described in Section 5. Section 6 presents the experimental evaluation and results. Section 7 concludes the paper.

2 RELATED WORK

A large number of contributions have applied both verification and testing methods to the SDN data plane in general. Several research efforts have focused on verification techniques where different properties can be checked, e.g. reachability, loop freedom, etc. For example, SAT solving (McGeer, 2012), (Zhang and Malik, 2013), symbolic execution/SMT solvers (Dobrescu and Argyraki, 2015), model checking over temporal logic (Ruchansky and Proserpio, 2013) and theorem proving (Chen et al., 2014) have been utilized for this purpose.

Despite formal verification can guarantee that the properties hold for the model, some implementation faults might still manifest. Therefore, *active* testing has been applied as well to the data plane (and the switch) where the latter is stimulated by test inputs. These works can be divided into two main groups. The approaches of the first group propose testing based on mechanisms of formal verification (Perešíni et al., 2015), (Bu et al., 2016), (Zeng et al., 2014), (Fayaz et al., 2016), (Stoenescu et al., 2016); in particular, Kuzniar et al. (Kuzniar et al., 2012) have proposed interoperability testing for the switch-to-controller interaction. The approaches of the second group rely on model based testing techniques when formal models are used to describe the desired behaviour of the data plane/switch under test (Zhao et al., 2017), (Fayaz and Sekar, 2014), (Alsmadi et al., 2015), (López et al., 2018), (Yao et al., 2014a), (Yao et al., 2017). In particular, the closest research to our work has been conducted in (Yao et al., 2014b) and (Zhang et al., 2016). Indeed, the authors in these works have also investigated the possibility of modelling the switch behaviour via a state machine, however, in contrast to our approach, they rather model the pipeline processing in a switch. Moreover, their approaches rely on a composition of finite state models and verify the packet processing while we are interested in testing the switch-to-controller communication.

To the best of our knowledge, model based testing approaches for checking the OF switch-to-controller communication are not yet present in the literature. We therefore propose one of them where the underlying model is expressed in terms of an Extended Finite State Machine.

3 BACKGROUND

3.1 OpenFlow Switch

An OF switch reacts to OF messages and data plane packets it receives. Therefore, it exposes two interfaces, i.e., the southbound interface (SBI) to handle the communication with the controller and the data plane interface taking care of the packets forwarding. The OF specification (Open Networking Foundation, 2014) describes the behaviour of the switch and its communication with the controller. It specifies OF messages handling via the SBI. Exam-

ples of messages received/sent by the switch include OFPT_HELLO message for connection establishment; OFPT_FEATURE for advertising the supported capabilities; FLOW_MOD for handling modification of rules in the switch; OFPT_BARRIER related to when a given command is applied; OFPT_MULTIPART for reporting statistics and OFPT_ECHO for sensing the liveness.

Before any messages can be exchanged, the connection establishment process takes place implying OF version and capability negotiation. Both ends of the connection exchange HELLO messages immediately after the lower layer (TCP/TLS) connection establishment. Afterwards, to be aware of the capabilities of the switch, FEATURE is exchanged. In case this message is not received by the controller and after a timeout, the latter disconnects the switch. Once the connection is successfully established, different messages can be exchanged, e.g., ECHO, FLOW_MOD, BARRIER and MULTIPART.

A rule installed by the FLOW_MOD message contains a matching part and an action part. The OF requirements define actions which can be applied to incoming packets and including modification of IP and VLAN values. If an OF implementation complies with the requirements, the exchange of these messages should be performed correctly with the specified parameters.

3.2 EFSMs

Let X and Y be finite sets of inputs and outputs; IN_p, OUT_p and C_v be finite disjoint sets of input/output parameters and context variables respectively. Some inputs (outputs) are related to subsets of parameters. For $x \in X$, let $IN_{px} \subseteq IN_p$ be the set of input parameters of x and let $D_{IN_{px}}$ be the set of input vectors, each component of an input vector corresponds to an input parameter associated with x. The set of output parameters and vectors are similarly defined. Let D_{C_v} be the set of context vectors $\mathbf{v}$. Given an input x and a (possibly empty) set of input vectors, a parameterized input is a tuple $(x, \mathbf{px})$ where $\mathbf{px}$ is an input parameter vector. A sequence of parameterized inputs is called a parameterized input sequence. Parameterized outputs and their sequences are defined similarly.

An EFSM (Petrenko et al., 2004) $\mathcal{S}$ over X, Y, IN_{px}, OUT_{py}, C_v, $D_{IN_{px}}$, $D_{OUT_{py}}$ and D_{C_v} is a pair (S,T) of a finite set of states S and a finite set of transitions T between states in S, such that each transition $t \in T$ is a tuple $(s,x,P,op,y,up,s\prime)$, where

- $s, s\prime \in S$ are the initial and final states of the transition, respectively;
- $x \in X$ is the input of the transition;
- $y \in Y$ is the output of the transition;
- P, op and up are functions, defined over input parameters and context variables
  - $P : D_{IN_{px}} \times D_{C_v} \rightarrow \{True, False\}$ is the predicate of the transition;
  - $op : D_{IN_{px}} \times D_{C_v} \rightarrow D_{OUT_{py}}$ is the output parameter function of the transition;
  - $up : D_{IN_{px}} \times D_{C_v} \rightarrow C_v$ is the context update function of the transition.

If a transition t has a predicate, the latter must be satisfied in order for t to be enabled. A configuration of $\mathcal{S}$ is a pair $(s, \mathbf{v})$.

An EFSM $\mathcal{S}$ is

•*Deterministic* if any two transitions outgoing from the same state with the same input have mutually exclusive predicates;

•*Complete* if for each pair $(s,x) \in S \times X$, there exists at least one transition at state s with the input x, otherwise $\mathcal{S}$ is called *partial*;

•*Initially connected* if each state of $\mathcal{S}$ is reachable from the initial state. Hereafter, the specification EFSM modelling the behaviour of an SDN switch is deterministic, partial and initially connected.

A *path* in $\mathcal{S}$ is the set of (parameterized) inputs of the successive transitions that are enabled from one configuration to another. Let $\mathcal{S}$ have the initial configuration $(s_0, \mathbf{v}_0)$. A *test sequence* is the sequence of input/output pairs of $\mathcal{S}$ that starts from $(s_0, \mathbf{v}_0)$ to a given configuration $(s, \mathbf{v})$ (i.e., a path from $(s_0, \mathbf{v}_0)$ to $(s, \mathbf{v})$).

Given a (parameterized) input sequence α, if α is defined at the initial states for machines $\mathcal{S}$ and $\mathcal{M}$, initial configurations $(s_0, \mathbf{v}_0)$ of $\mathcal{S}$ and $(s_0\prime, \mathbf{v}_0\prime)$ of $\mathcal{M}$ are distinguishable by α (*DS*) if the (parameterized) output sequences produced respectively by $\mathcal{S}$ and $\mathcal{M}$ in response to α are different, i.e., $out(\mathcal{S}, \alpha) \neq out(\mathcal{M}, \alpha)$. We furthermore refer to $\mathcal{S}$ as the specification machine describing the desired behaviour of an OF switch, while $\mathcal{M}$ denotes a potential faulty implementation of it. $\mathcal{M}$ is *distinguishable* from $\mathcal{S}$ (can be detected) if there exists such a *DS* α. Otherwise, $\mathcal{M}$ is *quasi-equivalent* to $\mathcal{S}$.

3.3 User-Defined Mutation and Fault Model

In this work, we assume that the specification machine $\mathcal{S}$ has a set of selected transitions, referred to as *suspicious* that are defined by a 'user' (can be an expert, tester, developer, etc.). We focus on output, transfer, predicate and update function faults at these transitions. Given a 'suspicious' transition $t = (s_i, x, P, op, y, up, s_j)$ of $\mathcal{S}$, t has an output fault if its (parameterized) output is distinct from that specified in $\mathcal{S}$. t has a transfer fault if its final state is differ-

ent from that specified by $\mathcal{S}$. t has an update function fault if its update function is omitted. t has a predicate fault if its predicate is negated [1].

Note that the introduced faults could be more sophisticated. For example, one can consider altering the operators of an update function. Nonetheless, even with these types of faults, our approach has proven to detect faults in an OVS implementation. Also, we note that in this work, only first-order mutants are considered.

As usual, a fault model is a tuple of the specification, conformance relation and fault domain, $\langle \mathcal{S}, \simeq, \mathcal{FD} \rangle$ (Petrenko et al., 2016). In this paper, the specification machine $\mathcal{S}$ is represented by an EFSM, and the conformance relation $\simeq$ is the *quasi-equivalence*, i.e., an implementation $\mathcal{M}$ conforms to the specification $\mathcal{S}$ if for each input sequence for which $\mathcal{S}$ is defined, $\mathcal{M}$ produces the same output sequence as defined by $\mathcal{S}$. $\mathcal{FD}$ is a set of implementation machines; faulty implementations are simulated by the mutants of interest. As usual, we are interested in deriving *exhaustive* test suites for $\langle \mathcal{S}, \simeq, \mathcal{FD} \rangle$, i.e., such test suites that detect each non-conforming (faulty) implementation $\mathcal{M} \in \mathcal{FD}$.

4 OF SWITCH MODEL

Formal models may be used as the basis for automating parts of the testing process and can lead to more efficient and effective testing (Hierons et al., 2009). Moreover, FSMs/EFSMs are widely used and have proven their effectiveness in various application domains, such as modeling and testing communication protocols, and other reactive systems. These formal models can be successfully adopted in specifying the properties of the OF switch and in capturing its functioning, particularly its communication with the controller. It is therefore of paramount importance to have a model which can capture the main interaction part between the controller and the switch, is able to model the main communication messages going from switch to controller (and vice versa), and can be easily extended to include additional parts of the OF specification or even the entire specification. The model proposed in this paper is an attempt in that direction.

[1]The specification can stay deterministic if other outgoing transitions are mutated accordingly. In our model, no more than two transitions with the same (parameterized) input (and different predicates) are defined at a state, thus when introducing a predicate fault we can simply change the predicates assigned to such two outgoing transitions. We assume this mutant is still of first order because it can correspond to a single fault in an implementation, e.g., unintentional swapping of the 'if-then-else' statements.

The proposed EFSM model (the specification $\mathcal{S}$) for the switch is derived from the OF requirements (Open Networking Foundation, 2014) only considering its interaction with the controller because our goal is to test a switch at the OF interface (and not at the data plane interface). The model is partially illustrated in Fig. 1 and is composed of five states, two non parameterized inputs and nine parameterized inputs, eleven non parameterized and seven parameterized outputs. It contains also two context variables, seven output parameter functions, fifteen predicates and three context update functions. Note a level of abstraction in this model, for example not all inputs/outputs of the original protocol are modelled. As the requirements do not describe precisely what the switch should reply in case of the success of a request received from the controller (e.g., response to a successful FLOW_MOD), in our model, we assume that the reply is a non-parameterized NULL_o output.

The sets of states S, inputs X and outputs Y are as follows:

$$S = \{\text{CLOSED}, \text{WAIT\_HELLO}, \text{WAIT\_FEATURE}, \text{CONNECTION\_ESTABLISHED}, \text{FAIL\_MODE}\};$$

$$X = \{\text{connected}, \text{HELLO}_i, \text{NULL}_i, \text{disconnected}, \text{FEATURE\_REQ}, \text{ADD}, \text{MULTIPART\_REQ}, \text{DELETE}, \text{BARRIER\_REQ}, \text{ECHO\_REQ}, \text{PACKET\_OUT}\};$$

$$Y = \{\text{HELLO}_o, \text{E}\mathit{rror}, \text{ERROR}_1, \text{ERROR}_2, \text{ERROR}_3, \text{ERROR}_4, \text{ERROR}_5, \text{ERROR}_6, \text{ERROR}_7, \text{ERROR}_8, \text{ERROR}_9, \text{ERROR}_{10}, \text{MULTIPART\_REP}, \text{NULL}_o, \text{FEATURE\_REP}, \text{ECHO\_REP}, \text{BARRIER\_REP}, \text{FLOW\_REMOVED}\}.$$

The EFSM reflects that the switch supports connection version negotiation (parameterized input HELLO_i) and preserves the behaviour of correct exchange of FEATURE, ADD and DELETE (modelling the FLOW_MOD message), BARRIER, ECHO, PACKET_OUT and MULTIPART messages. The handshake and version negotiation are handled by predicates of t_0, t_1 and t_2. For example, transition t_2 in Fig. 1 is written as

$$t_2 = (\text{WAIT\_HELLO}, \text{HELLO}_i, P_2, -, \text{NULL}_o, -, \text{WAIT\_FEATURE})$$

where WAIT_HELLO and WAIT_FEATURE are the initial and final states of t_2 respectively, HELLO_i is the parameterized input. P_2 is the predicate checking the values of parameters *type* and *version* of the input. NULL_o is the output sent by the switch representing a success of the HELLO_i request. Once in t_2's final state and upon receipt of the parameterized input FEATURE_REQUEST, if the predicate of t_5 evaluates to True (indicating non expiry of a timeout), the machine outputs the parameterized

FEATURE_REPLY (containing the switch capabilities) and moves to the state CONNECTION_ESTABLISHED. In case FEATURE_REPLY is not sent after a timeout (predicate of t_3 evaluates to True), the machine moves to the initial state CLOSED indicating a disconnection. Some of the exchanged messages are modelled as self-looping transitions once the connection is successfully established and the machine being in the CONNECTION_ESTABLISHED state.

We note that the values of input/output parameters have rather small domains. For example, the input parameter *version* of HELLO$_i$ input takes only the single value $0x04$ and the input parameters for the input ADD are $[type \quad table \quad match \quad action \quad flags]$ and their values have small domains as well. The same for output parameters.

The machine has the set of two context variables; $C_v = \{nbFlows, \text{TLS\_}timeout\}$ denoting respectively the number of rules in the switch and the TLS session timeout. *nb*F*lows* can take distinct values in the range $[0..max\_entries]$ where *max_entries* is a constant that denotes the maximal number of rules the switch can insert and depends on the switch characteristics (takes the value of 10^6 in our model). In case TLS_*timeout* expires, the switch loses the connection with the controller and the machine moves to state CLOSED (t_4). The OF requirements specify that the connection maintenance is done by the underlying TLS/TCP connection mechanisms and since currently supported protocols have the same default timeout value of 300 seconds, we set TLS_*timeout* to this value.

Note that the finite state model proposed in this paper can be extended to model additional parts of the OF requirements, or even the entire specification. For example, the configuration messages handling groups, queues, and meters can be added as self-looping transitions after the connection establishment, or other states can be added as well. The model allows the detection of potential faults and misinterpretations as it will be shown by our experiments. However, its expanding would eventually require more significant upfront time. Naturally, an evaluation of such expanding is needed and hence forms a direction of future work.

5 TEST GENERATION

As mentioned above, we are interested in deriving test cases that check that the suspicious transitions of $\mathcal{S}$ are correctly implemented in the switch. Our approach is a depth first search based heuristic that progressively constructs a test suite TS. First, a set of

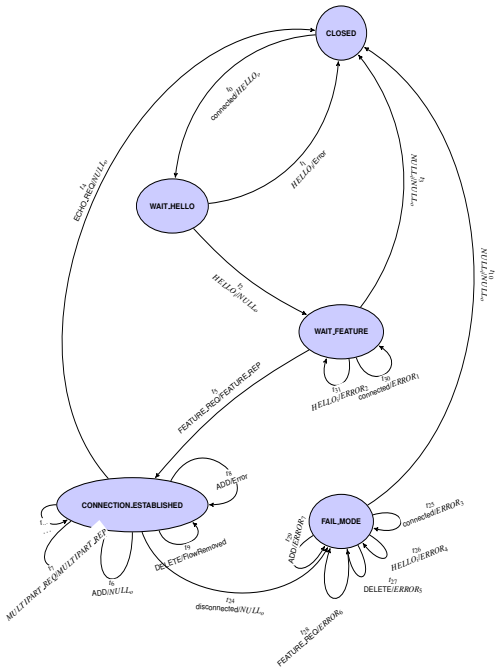

Figure 1: Part of the specification EFSM of the switch.

user-defined mutants for suspicious transitions is derived. Then for each mutant $\mathcal{M}_i$ of the set, for checking the distinguishability between a reached configuration $(s, \mathbf{v})$ where the suspicious transition is defined in $\mathcal{S}$ and the corresponding configuration reached in $\mathcal{M}_i$, the approach tries to append the *DS*s if they exist up to a certain predefined positive depth l.

The construction of a *DS*, say σ, of $\mathcal{S}$ and a given $\mathcal{M}_i$ is performed in the following way. It is formed of a preamble α and a postamble that is appended to α to form the *distSeq*. The preamble α is an input sequence that takes $\mathcal{S}$ from the initial configuration $(s_0, \mathbf{v}_0)$ to the configuration where the suspicious transition is defined. The *distSeq* is constructed by a depth first search that will repeatedly expand deeper configuration nodes in the EFSM and explore the successive configurations. In other words, configurations at depth l (which corresponds to length $|\alpha| + l$ starting from the initial configuration) are treated as if they have no outgoing transitions (successors). The algorithm therefore progressively increases the depth until it finds that the outputs of $\mathcal{S}$ and $\mathcal{M}_i$ are different or that the limit l for the depth is reached (i.e., the length $|\alpha| + l$ is reached).

The approach is formalized in the following algorithms. Algorithm 1 captures the main flow of the proposed approach. It first assigns TS to an empty set. Then, for each mutant $\mathcal{M}_i$ in the set $\mathcal{FD}$, it looks for the corresponding *DS*. Since our EFSM is initially connected, we are only interested in configu-

rations that are reachable from the initial configuration. A preamble α is generated to first reach the suspicious transition. The specification $\mathcal{S}$ is then simulated over α and the sequence σ is first empty and is later extended with the (parameterized) sequence DISTSEQ if it exists to satisfy the distinguishability between the two configurations reached in $\mathcal{S}$ and the mutant respectively. Therefore, for a *length* going from $|\alpha|$ to $|\alpha|+l$, i.e., from the depth of the starting suspicious transition to the defined depth limit l, the algorithm repeatedly calls DISTINGUISHINGSEQUENCEAPPEND illustrated in Algorithm 2 which takes care of suitable extensions. Eventually, Algorithm 1 will find a *DS* if one exists up to depth l (i.e., length $|\alpha|+l$ from the initial configuration). Finally, if such sequence exists, it is added to TS while the mutant is marked as distinguishable, i.e., added to the set $\mathcal{FD}\prime \subseteq \mathcal{FD}$. When all derived mutants are considered, TS is returned along with $\mathcal{FD}\prime$ containing distinguished mutants.

Proposition 1. *If Algorithm 1 returns a test suite TS then this test suite is exhaustive w.r.t. the fault model* $\langle \mathcal{S}, \simeq, \mathcal{FD}\prime \rangle$. □

Proof.
Indeed, $\mathcal{FD}\prime$ contains only distinguishable mutants. Moreover, the derived test suite TS is formed by all the input sequences that distinguish the specification $\mathcal{S}$ and each mutant of the set of mutants $\mathcal{FD}\prime$. Therefore, each faulty implementation/mutant is detected by the derived test suite TS.

□

Below, we show an example of a mutant and the corresponding *DS* returned by Algorithm 1.

Consider the following transition t_6

$$t_6 = (\text{CONNECTION\_ESTABLISHED}, \text{ADD}, P_6, op_6, \text{NULL}_o, up_6, \text{CONNECTION\_ESTABLISHED})$$

where up_6 indicates that *nb*F*lows* is increased by one. A mutant M_0 considers an issue that sometimes developers forget to update the value of a variable. The idea here is that M_0 is derived by omitting the update function up_6 of t_6.

$$up_6 : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$$
$$nb\text{F}lows \leftarrow nb\text{F}lows + 1$$

It suggests the switch will not update its statistics after adding a new rule. The corresponding *DS* returned by Algorithm 1 is shown below.

Algorithm 1: Algorithm that derives a test suite TS and a set $\mathcal{FD}\prime$ of distinguishable mutants.

Input : EFSM $\mathcal{S}$, initially connected and deterministic, a set $\mathcal{FD}$ of mutants, an upper bound positive integer $l > 0$
Output: A test suite TS and a set of mutants $\mathcal{FD}\prime$ that can be detected or *NoSolution*

```
1  TS ← ∅
2  FD' ← ∅
3  foreach M_i ∈ FD representing a fault at a
     'suspicious' transition
     t_i = (s_i, x_i, P_i, op_i, y_i, up_i, s_i') do
4      Let α be a (parameterized) input sequence
         that takes S from the initial configuration
         (s_0, v_0)^S to the configuration (s_i, v_i)^S
         where the predicate P_i of t_i can be
         evaluated to True and a (parameterized)
         input x_i is defined
5      Simulate S applying α and reach
         configuration (s_i, v_i)^S
6      σ ← empty
7      for length ← |α| to (|α|+l) do
8          DISTSEQ ← DISTINGUISHINGSE-
             QUENCEAPPEND(S, M_i, (s_i, v_i)^S,
             length)
           // depth limited search
9          if DISTSEQ ≠ NoSolution then
10             σ ← σ.DISTSEQ
11     if |σ| > |α| then
12         TS ← TS ∪ {σ}
13         FD' ← FD' ∪ {M_i}
14 if |TS|==0 then
15     return NoSolution
16 return TS and FD'
```

$$\begin{aligned} DS = &\text{connected}(\text{OFPT\_HELLO}, 0x04) \\ &\text{HELLO}_i(\text{OFPT\_HELLO}, 0x04) \\ &\text{FEATURE\_REQ}(\text{FEATURE\_REQUEST}, 0x04) \\ &\text{BARRIER\_REQ}(\text{BARRIER\_REQUEST}) \\ &\text{ADD}(\text{OFPT\_FLOW\_MOD}, 0, any, 1, 0) \\ &\text{BARRIER\_REQ}(\text{BARRIER\_REQUEST}) \\ &\text{MULTIPART\_REQ}(\text{OFPMP\_TABLE}) \\ &\text{BARRIER\_REQ}(\text{BARRIER\_REQUEST}). \end{aligned}$$

Algorithm 2: DISTINGUISHINGSEQUENCEAPPEND($\mathcal{S}$,$\mathcal{M}$, $(s_i, \mathbf{v_i})$, *length*).

Input : Initially connected and deterministic EFSMs $\mathcal{S}$ and $\mathcal{M}$, a configuration $(s_i, \mathbf{v_i})$, an upper bound positive integer *length*

Output: A test sequence or *NoSolution*

```
NoSolutionHappen ← False
seq ← input sequence of the path up to (s_i, v_i)
if out(S, seq) ≠ out(M, seq) then
    return seq
if |seq|==length then
    return NoSolution
foreach transition t = (s_i, x_i, P_i, op_i, y_i, up_i, s_i') outgoing from the configuration (s_i, v_i) with P_i evaluating to True do
    successor ← (s_i', v_i') // v_i' is the result of up_i applied to v_i
    path ← DISTINGUISHINGSEQUENCEAPPEND(S, M, successor, length)
    if path == NoSolution then
        NoSolutionHappen ← True
    else
        return path
if NoSolutionHappen then
    return NoSolution
```

6 EXPERIMENTAL EVALUATION AND RESULTS

6.1 Evaluation of the Approach

First, given the depth l to which Algorithm 1 is allowed to explore the generated transitions tree starting from the suspicious transition of interest, we have investigated the impact of this depth on the effectiveness of the generated test suites (fault coverage). For this purpose, we have varied the depth l and we have generated corresponding test suites, then we have measured the mutation score of each generated test suite. In the second part, we have further assessed the effectiveness of the test suites derived by the proposed approach by comparing them to test suites generated randomly, i.e., an approach that randomly selects inputs from the input set. We refer to this approach as RG (Random Generation). The goal is to compare the *TS*s derived by our test generation method with size-equivalent test suites that do not follow any systematic test generation strategy. Though this provides only a baseline and a comparison with alternative testing methods is definitely relevant, it is a necessary starting point.

In order to perform an experimental evaluation, a number of software tools have been developed. The experimentation process is composed of four main steps. The first step is to generate mutants of different kinds. The second step is to generate the test suites based on the two approaches aforementioned. The third step of the experiment is to produce JUnit files that can run the test suites. The fourth and final step is running the generated test suites against the generated mutants.

Further on, automation of test suite generation is paramount. Similarly, mutant generation should be automated. However, in our experiments, we have generated two sets of mutants. A first set of user-defined mutants have been manually generated and added to the second set of automatically (randomly) generated ones. We have therefore developed a testing framework. The framework is composed of two main modules. The first allows the test engineer to produce test suites according to two different test generation methods; using our proposed approach and using RG. The mutants have been generated and the test suites have been automatically constructed. The second module of the testing framework helps to execute the randomly generated test suites against the different generated mutants. Note that the random test suites generation is based on the random function provided by the Java programming language.

We discuss our measurement of performance and effectiveness (at finding faults) using two metrics as follows. On one hand, concerning the *performance*, we focus precisely on the execution CPU time metric. On the other hand, concerning the effectiveness of our testing technique, we focus on mutation analysis to compare the capability (of fault detection) of the derived test suites based on our approach against the capability of randomly generated test suites in terms of mutation score.

One issue to be addressed is the detection of equivalent mutants, i.e., mutants that have the same behaviour as the specification machine and therefore cannot be killed by test sequences. There are studies proposing techniques to automate the detection of equivalent mutants, and a commonly used heuristic is to consider survived mutants not killed by any test suite in the overall test pool (i.e., in test suites being compared) as equivalent mutants (DeMillo et al., 1978). In this work, we have used this heuristic. Since we compare test suites between one another, this assumption should not introduce a significant threat to validity of our results. In our experiments, we have

produced output, transfer, predicate and update function mutants.

We have created a total of 288 mutants where 70 have been manually generated. 20 of them (not killed by any of our test suites) were equivalent. Overall, we have therefore used 268 mutants where 67 are output, 67 are transfer, 67 are predicate, and 67 are update function. 10 test suites have been generated using our approach. This has been performed by increasing the depth l until 10 and for each fixed depth $l = \{1,\ldots,10\}$, a corresponding test suite has been generated. To generate size-equivalent random test suites, for each generated test suite using our approach, we have measured the average length of all of its test sequences. For each of these length values, a corresponding test suite of that same length has been randomly generated using the RG approach. Hence, 10 test suites have been randomly generated.

Figure 2 illustrates the mutation score as the depth increases using our approach. The mean mutation score and the execution CPU time (in minutes) for both our approach and RG are shown in Figure 3.

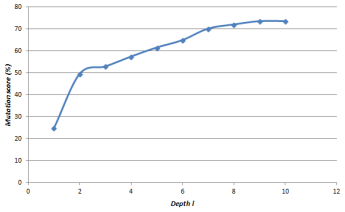

Figure 2: Mutation score as the depth increases

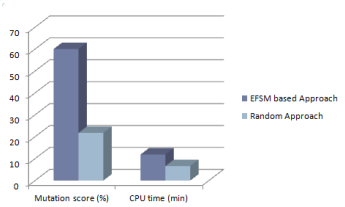

Figure 3: Average mutation score and execution time for TSs derived by the proposed approach Vs TSs randomly generated

The results for randomly generated test suites show an average mutation score of 21.75% (Figure 3). This is significantly lower than the 60.07% average that has been obtained with the proposed EFSM based method (with the depth l varying from 1 to 10). The average mutation score is even significantly higher than the maximal score with random test suites. We can then conclude that, factoring out the cost of testing, the EFSM based testing technique is rather relevant. It is able to outperform the effectiveness of randomly generated test suites by achieving the highest mutation score. We conjecture that this is due to not only the ability of our technique to cover more transitions in the EFSM model, but also to its ability to distinguish between configurations which the random approach does not do.

However, as shown in Figure 3, the CPU time that our proposed technique requires is higher compared to the RG approach. Therefore, there is a trade-off between the performance and the quality of the generated test suites using the two methods.

6.2 Experiments on an OVS Switch Implementation

To evaluate the effectiveness of the EFSM based approach in revealing OF switch implementation faults, we have developed another module of our testing framework that includes two main components. First, there is a translation module that maps the generated test sequences described in Section 5 into OF-syntax messages. It uses an input file specifying the derived inputs to fill the different input parameter values and construct a TS ready to be executed against the SUT. A second component takes care of the comparison of observed and expected outputs specified in the input file as well. The framework allows to send TSs to an SUT, collect the observed outputs and compare them against the specified expected ones to finally print a conclusion displaying 'OK' or 'Fail' message to the user.

Experiments have been performed on the OVS version 2.5.0. To emulate a 'close-to-real' test environment, Mininet (De Oliveira et al., 2014) tool has been utilized. In the testing set up, Mininet has been executed under Mint 18.1 with 8GB of RAM and 1 core of a 2.4 GHz Intel Core i7. The Floodlight 1.1 controller has been used. The SUT has been connected to a testing engine containing an emulated controller that is capable of sending and capturing OF messages in a controlled manner. Figure 4 shows the SUT (VM_1) and the testing engine (VM_2). The source code of OVS 2.5.0 has been directly cloned from its Git repository on VM_1. The SUT may receive any of the inputs from the input alphabet X emitted from the controller interface of $VM2$.

The test suites TSs of interest have been executed against the OVS implementation of interest. In the following, we present the bugs detected in OVS 2.5.0 implementation along with the related DSs and their lengths.

The existence of a fault in OVS 2.5 implementation has been revealed by a test sequence (DS) of one of the generated TSs. It concerns the installation of a new rule with parameter *action* set to 'OFPActionPushVlan' which pushes a new VLAN tag

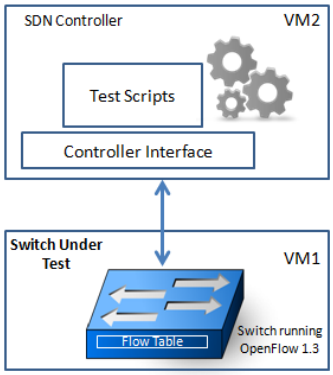

Figure 4: Test architecture.

onto the packet matching the entry. In this case, the SUT has replied with an Error message with parameters $type$ = OFPET_BAD_ACTION and $code$ = OFPBAC_BAD_ARGUMENT. The observed reply reveals a fault in the SUT in association with installing a new rule having an action that pushes a new VLAN tag to the matching packet. The switch has behaved as if an action in the $Flow\_Mod$ message ADD had a value that is invalid, however the 'OFPActionPushVlan' action is specified to be supported. The length of the *DS* capable of detecting such fault is 7.

Another fault has been detected by another test sequence and concerns the installation of a new rule with parameter *action* set to OFPP_ALL which is supposed to forward the matching packet to all ports of the switch under test. In this case, the SUT has replied with an Error message as well instead of a NULL_o reply. The parameters of the observed Error reply are $type$ = OFPET_BAD_ACTION and $code$ = OFPBAC_BAD_ARGUMENT. In this case, the length of the *DS* capable of detecting this fault is 7 as well.

A similar fault has been detected which concerns the installation of a new rule with parameter *action* set to OFPP_IN_PORT. When the SUT receives a $Flow\_Mod$ message with the indicated parameter, it is supposed to reply with a NULL_o reply and install the rule which forwards the matching packet to all ports except the input one. However, the observed reply has been an Error message similar to the previously mentioned one. The length of the *DS* capable of detecting such fault is also 7.

The next detected fault concerns the MULTIPART_REQ for reporting statistics about installed rules. The input sequence $\alpha \in TS$ is intended to add a rule to the SUT (when $nbFlows$ is less than $max\_entries$) and request the SUT about statistics. The expected reply should show increment of the total number of rules $nbFlows$ by one. However, the SUT reply contains the exact same number of rules than before applying the *DS*. This means that the switch when adding the rule, does not update statistics. The *DS* capable of detecting such fault is derived based on an update function mutant and it contains a $Flow\_Mod$ input and a MULTIPART_REQ input for adding a new rule and then getting the statistics. The $type$ parameter of the MULTIPART_REQ is set to OFPMP_TABLE. The reply to this *DS* contains the active number of installed rules. The *DS* in this case has length 8.

Another type of fault has been detected as well. It concerns a table overflow. The *DS* of interest includes the parameterized input ADD 10^6 times followed by another input ADD. In this case, the SUT has replied with 10^6 NULL messages followed by another NULL and has added the 'extra' rule. Note that in section 6.4 of the OpenFlow requirements (Open Networking Foundation, 2014), it is stated that "If a switch cannot find any space in the requested table in which to add the incoming flow entry, the switch must send an ofp_error_msg with OFPET_FLOW_MOD_FAILED type and OFPFMFC_TABLE_FULL code". This indeed reports a fault in the OVS 2.5 under test and confirms our intuition that in the implementation, the variable $nbFlows$ is not checked for reaching its extreme value $max\_entries$. The *DS* has length $10^6 + 6$. Thus it has not been derived by our implementation but rather derived manually using our model. Once the configuration of interest is reached in the model, the transition with the parameterized input ADD and having the predicate $[nbFlows < max\_entries]$ has been simulated to be traversed 10^6 times.

Discussion

There are several threats that could potentially affect the validity of our study. One of the threats susceptible to affect our study is the one referring to generalizability of our findings. In this preliminary study, it is clear that the results we have obtained are a priori limited as they are based on one case study (as a baseline) involving the comparison of our approach to the RG approach. Therefore, more investigations and comparisons are necessary in order to be able to generalize. Another point worth mentioning is related to checking the scalability of the proposed approach w.r.t. the size and complexity of the built model. In other words, if the proposed model is extended to incorporate additional parts of the OF requirements, it would be interesting to investigate how the complexity of the proposed test derivation algorithm will grow accordingly. This question also forms a direction of planned future work. In contrast, as we have used mutation score being a surrogate metric of evaluating the effectiveness at detecting faults and as a test

suite mutation score has proven to be correlated with its real fault detection rate (Aichernig et al., 2014), we believe there is little threat to the validity in general. Besides, we have experimented on an OF switch implementation which has allowed us to detect some faults.

7 CONCLUSION

In this paper, an EFSM based testing technique for an OF switch-to-controller communication has been presented. Given an EFSM model of the switch-to-controller interaction derived from OF requirements, and a set of mutants representing faults defined by a user at suspicious transitions, the method derives a test suite formed by *distinguishing sequences* aiming at detecting the mutants of interest.

We have evaluated the proposed test derivation technique by comparing it to a random generation approach. The results have shown that the designed algorithm is relevant. Further, preliminary experiments with an Open vSwitch have confirmed the effectiveness of the proposed approach in revealing some implementation faults. However, the findings presented in this work should be interpreted in the context of limitations related to the model design decisions and the number of *user-driven* mutants. In the future, we plan to improve the operation efficiency of the approach and apply it to larger SDN models on one hand and compare it to other (EFSM based) test generation techniques on the other hand.

Finally, the testing framework features are planned to be enhanced so that interested testers / developers could execute their own test suites against SDN switches in a fully automated manner. The issues and challenges listed above form the directions for the future work.

REFERENCES

Aichernig, B. K., Auer, J., Jöbstl, E., Korošec, R., Krenn, W., Schlick, R., and Schmidt, B. V. (2014). Model-based mutation testing of an industrial measurement device. In *International Conference on Tests and Proofs*, pages 1–19. Springer.

Alsmadi, I., Munakami, M., and Xu, D. (2015). Model-based testing of sdn firewalls: a case study. In *Trustworthy Systems and Their Applications (TSA), 2015 Second International Conference on*, pages 81–88. IEEE.

Bochmann, G. V. and Petrenko, A. (1994). Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124. ACM.

Bu, K., Wen, X., Yang, B., Chen, Y., Li, L. E., and Chen, X. (2016). Is every flow on the right track?: Inspect sdn forwarding with rulescope. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE.

Chen, C., Jia, L., Zhou, W., and Loo, B. T. (2014). Proof-based verification of software defined networks. In *ONS*. USENIX.

De Oliveira, R. L. S., Schweitzer, C. M., Shinoda, A. A., and Prete, L. R. (2014). Using mininet for emulation and prototyping software-defined networks. In *Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6. IEEE.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

Dobrescu, M. and Argyraki, K. (2015). Software dataplane verification. *Communications of the ACM*, 58(11):113–121.

El-Fakih, K., Prokopenko, S., Yevtushenko, N., and Bochmann, G. v. (2003). Fault diagnosis in extended finite state machines. In *Proceedings of the IFIP International Conference on Testing of Software and Communicating Systems*, pages 197–210. Springer.

Fayaz, S. K. and Sekar, V. (2014). Testing stateful and dynamic data planes with flowtest. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 79–84. ACM.

Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V. (2016). Buzz: Testing context-dependent policies in stateful networks. In *NSDI*, pages 275–289. USENIX.

Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., et al. (2009). Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):9.

Kuzniar, M., Peresini, P., Canini, M., Venzano, D., and Kostic, D. (2012). A soft way for openflow switch interoperability testing. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 265–276. ACM.

López, J., Kushik, N., Berriri, A., Yevtushenko, N., and Zeghlache, D. (2018). Test derivation for sdn-enabled switches: A logic circuit based approach. In *IFIP International Conference on Testing Software and Systems*, pages 69–84. Springer.

McGeer, R. (2012). Verification of switching network properties using satisfiability. In *The IEEE International Conference on Communications (ICC)*, pages 6638–6644. IEEE.

Open Networking Foundation (2014). Openflow Switch Specification Version 1.3.4. https://www.opennetworking.org.

Perešíni, P., Kuźniar, M., and Kostić, D. (2015). Monocle: Dynamic, fine-grained data plane monitoring. In *Proceedings of the 11th ACM Conference on Emerg-*

ing Networking Experiments and Technologies, pages 1–13. ACM.

Petrenko, A., Boroday, S., and Groz, R. (2004). Confirming configurations in efsm testing. *IEEE Transactions on Software Engineering*, 30(1):29–42.

Petrenko, A., Timo, O. N., and Ramesh, S. (2016). Test generation by constraint solving and fsm mutant killing. In *IFIP International Conference on Testing Software and Systems*, pages 36–51. Springer.

Ruchansky, N. and Proserpio, D. (2013). A (not) nice way to verify the openflow switch specification: Formal modelling of the openflow switch using alloy. *SIGCOMM Comput. Commun. Rev.*, 43(4):527–528.

Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. (2016). Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 314–327. ACM.

Yao, J., Wang, Z., Yin, X., Shi, X., Li, Y., and Li, C. (2017). Testing black-box sdn applications with formal behavior models. In *25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 110–120. IEEE.

Yao, J., Wang, Z., Yin, X., Shi, X., Wu, J., and Li, Y. (2014a). Test oriented formal model of sdn applications. In *Performance Computing and Communications Conference (IPCCC)*, pages 1–2. IEEE International.

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

Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. (2014). Automatic test packet generation. *IEEE/ACM Trans. Netw.*, 22(2):554–566.

Zhang, S. and Malik, S. (2013). Sat based verification of network data planes. In *International Symposium on Automated Technology for Verification and Analysis*, pages 496–505. Springer.

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.

Zhao, Y., Zhang, P., Wang, Y., and Jin, Y. (2017). Sdn-enabled rule verification on data plane. *Communications Letters*, pages 1–1.