Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 160 (2019) 23–29

Procedia Computer Science

www.elsevier.com/locate/procedia

The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2019)
November 4-7, 2019, Coimbra, Portugal

Validation of SDN policies: a property-based testing perspective

Laura M. Castro[a,*], Nicolae Paladi[b,c]

[a] *Universidade da Coruña, Centro de Investigación CITIC, Spain*
[b] *Lund University, Sweden*
[c] *RISE Research Institutes of Sweden, Sweden*

Abstract

Software-defined networks are being widely adopted and used in large and complex networks supporting critical operations. Their increasing importance highlights the need for effective validation of SDN topologies and routing policies both prior and during operation. The policies that configure an SDN deployment come from several, possibly conflicting sources. This may lead to undesired effects such as node isolation, network partitions, performance drops and routing loops. Such effects can be formulated as automatically testable reusable conditions using property-based testing (PBT). This approach allows to automatically determine and formulate as a counterexample the minimum set of conflicting rules. The approach is especially useful when policies are configured in an incremental manner. PBT techniques are particularly good at automatic counterexample *shrinking* and have the potential of being extremely effective in this area.

© 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Conference Program Chairs.

Keywords: software-defined networks (SDN); property-based testing (PBT); MiniNET; QuickCheck; Hypothesis

1. The challenge of SDN policies validation

Software Defined Networks (SDNs) have become popular in enterprise and research settings. Some of the reasons for this are: (a) they establish a functional separation of concerns amongst application, control and infrastructure layers; (b) they facilitate network virtualisation and allow seamless cloud support; and (c) they enable network automation and in-network processing through programmability.

With an SDN, an administrator can supervise and update networking rules during operation, change their priorities, steer traffic, and perform other network management actions. This is done via a centralised, unified SDN controller instead of tampering directly with heterogeneous hardware switches. In multi-tenant, cloud-based architectures like

* Corresponding author. Tel.: +34-881-01-1269 ; fax: +34-981-167-160.
E-mail address: lcastro@udc.es

1877-0509 © 2019 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)
Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2019.09.439

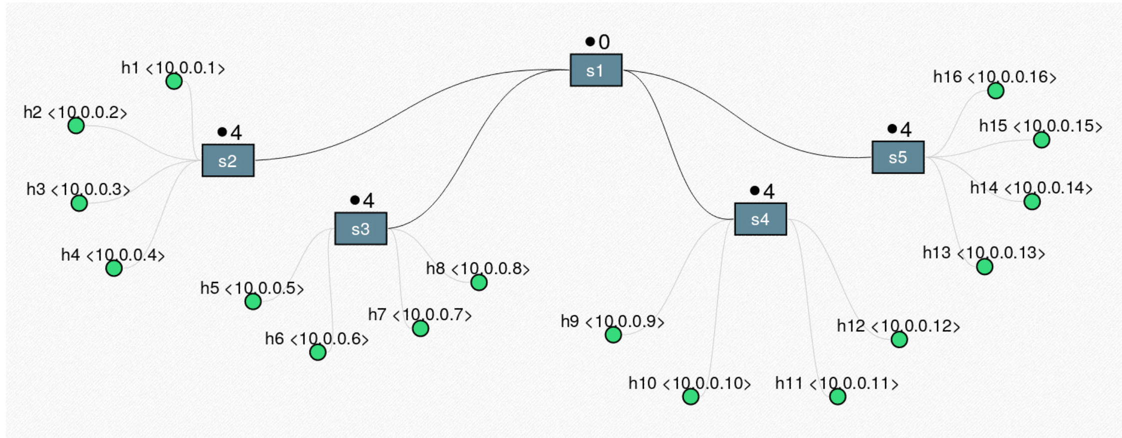

Fig. 1: Example of SDN with a tree topology

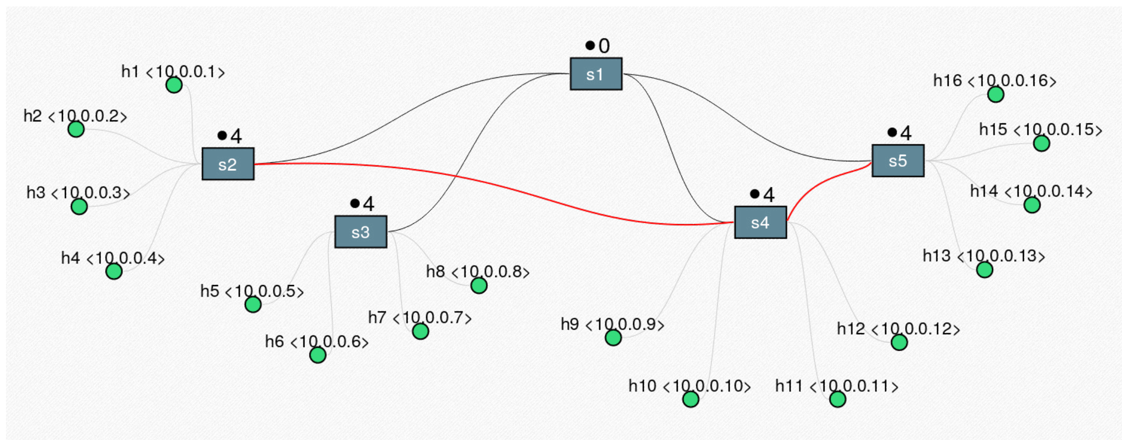

Fig. 2: Evolved SDN topology

data-centres or large Internet-of-Things (IoT) systems, this allows to manage system load in a controlled, flexible and efficient manner. Considering that SDN are a cornerstone of the upcoming 5G, it is reasonable to think the adoption of SDNs has hardly seen its peak [14].

Building larger and more complex SDN deployments results in increasingly complex policies to operate them. Policies in an SDN deployment may be issued from several sources - by network applications, by the network administrator, or by the network controller based on a configuration file. The policies may be added, removed, and changed during operation and in arbitrary order. For example, in the SDN depicted on Fig. 1, requests from hosts $h_1 - h_4$ to hosts $h_{13} - h_{16}$ could be easily dropped at s_1 with a rule like:

```
DROP REQUEST WHERE origin = s2 AND destination = s5    (1)
```

Later on, if there are too many packets flowing between hosts $h_9 - h_{12}$ and $h_{13} - h_{16}$, the SDN administrator could decide to add a new link between s_4 and s_5 to improve latency. A similar situation could later require a new link between s_2 and s_4, resulting in a network topology similar to the one in Fig. 2. As a result, we inadvertently added a path between hosts connected to s_2 and s_5 via s_4. Requests amongst hosts $h_1 - h_4$, and hosts $h_{13} - h_{16}$ routed through this new path violate aforementioned rule 1, installed only in s_1 where it was needed at the time.

Admittedly, this is a simple scenario with few network components and a very straightforward rule. However, the situation looks different when we consider cloud-size deployments with complex, dynamic and possibly conflicting rules: the need and complexity of detecting potential violations and the interest of doing so before their effects propagate in the system both become more evident.

The SDN controller can verify the policies before applying them. There are several tools that address this issue from several perspectives [13]. However, many such tools focus on verifying *reachability* and loop absence; moreover, most of them do not scale to large networks with hundreds of thousands of nodes.

A different angle to the situation described in our previous example would be adding rule 1 when the network has already reached the configuration shown on Fig. 2. Whether we would consider this an ineffective measure because traffic from s_2 would still be able to reach s_5 through s_4, or a purposeful measure aimed to just alleviate s_1 load, it would be difficult to assess in a fully-automatic way, and it is, to the best of our knowledge, outside of the scope of the existing approaches.

In this work-in-progress, we explore whether property-based testing (PBT) techniques can fill in the above gaps and become a potent tool for SDN policy validation. PBT allows for a semi-formal description of requirements that suits perfectly how SDN policies are defined. We have successfully run experiments showing that SDN network deployments can be translated into easy-to-reuse PBT stateful models with invariants describing its enforced policies. We then used these descriptions to effectively test for rule conflict situations like those described above. We implemented the experiments using popular open-source tools: Hypothesis [7], a PBT tool for Python; and MiniNET [6], a simulator implementing OpenFlow [8], a widely used SDN protocol.

2. Property-based testing for SDN validation

Property-based testing [2] (PBT), and in particular stateful property-based testing [1], is a testing technique that automates the validation of invariants, called *properties*, given a model of the subject under test (SUT) and rules for modelling its *behaviour* (evolution and change during operation). Given a stateful model, a PBT tool can automatically assess its behaviour generating, running, and evaluating large numbers of sequences of interactions. The random element introduced during generation in both test data and test sequence composition and length, accounts for an effective complement, or even replacement, of more traditional example-based testing (i.e. unit testing) [3].

One of the most appreciated features of PBT techniques is *shrinking*. Upon fault detection (i.e. inconsistency between SUT expected behaviour according to its model, and SUT actual behaviour), shrinking represents the automatic simplification of both sequence of interactions input data in order to reproduce the same faulty behaviour.

Our application of PBT to SDN policies validation demonstrates two complementary approaches, namely:

Given a fixed set of policies, an SDN deployment can be automatically evolved (for example, by adding and removing links or changing bandwidth) to detect violations.

The policies in the fixed set act as invariants, and the PBT stateful model features SDN changes as interactions. Sequences of SDN changes are then randomly generated as test cases, with invariants being checked both after each change and after the whole sequence. Upon invariant violation detection, SDN change sequence is reduced to show a minimum counterexample, so a simplified network is presented, where all components which are not affected by the issue are removed. For our example in Sect. 1, test model invariant (cf. 3a) and counterexample (cf. 3b) would look as shown in Fig. 3. Actually, the first counterexample found by our model (cf. Fig. 3c) was directly linking s_2 and s_5, but since that sort of "configuration error" will likely be immediately spotted, we modified the *test data generator* `tree_link` so that a combination of at least two links were needed to hit the error. Due to the random component of PBT, sometimes test run would yield a different counterexample: linking s_2 to s_3 and s_3 to s_5 (and any of the two in the reverse order).

Given a fixed SDN deployment, policies can be generated and applied in random order, to automatically detect conflicts.

In this case, the interactions generated by the PBT model correspond to new SDN policies. After a new policy is generated and added to the SDN, all accumulated policies are checked. Upon conflict detection, the shrinking process is able to eliminate from the test case the policies that do not take part in the detected issue, effectively trimming the list of conflicting policies to the incompatible ones.

```
@invariant()
def fixed_policies(self):
    print '***  INVARIANT ***'
    host1 = self.net.get('h1')
    host16 = self.net.get('h16')
    value = self.net.ping([host1,
                           host16])
    assert (value == 100) # reachability measured
                          # in % of dropped pckgs
```

(a) Invariant for SDN reachability

```
Falsifying example:
+++ Created network
***  INVARIANT ***
h1 -> X
h16 -> X
*** Results: 100% dropped (0/2 received)
+++ Linking s2 and s4
***  INVARIANT ***
h1 -> X
h16 -> X
*** Results: 100% dropped (0/2 received)
+++ Linking s4 and s5
***  INVARIANT ***
h1 -> h16
h16 -> h1
*** Results: 0% dropped (2/2 received)
----------------------------------------
Ran tests in 2868.531s

FAILED (failures=1)
****************************************
```

(b) Counterexample found automatically for invariant violation

```
class MininetStatemFixedPolicies(RuleBasedStateMachine):
    def __init__(self):
        super(MininetStatemFixedPolicies, self).__init__()
        self.net = TreeNet(depth=2,fanout=4,autoStaticArp=True)
        self.net.start()
        self.net.configLinkStatus('s1', 's2', "down")
        self.net.configLinkStatus('s1', 's5', "down")
        print '+++ Created network: {0}'.format(self.net.keys())

    @rule(link = mns.tree_link())
    def connect_components(self, link):
        print '+++ Linking {0} and {1}'.format(link[0], link[1])
        self.net.configLinkStatus(link[0], link[1], "up")

    @rule(link = mns.tree_link())
    def disconnect_components(self, link):
        print '+++ Unlinking {0} and {1}'.format(link[0], link[1])
        self.net.configLinkStatus(link[0], link[1], "down")

    def teardown(self):
        self.net.stop()
```

(c) PBT model

Fig. 3: Example of PBT for SDN validation with fixed policies

For the second situation in Sect. 1, test model, test invariant, and counterexample would look like Fig. 4. For simplicity, the *policies generator* `policy` we have defined produces only two kinds of basic rules: `(reachable, source, destination)` and `(unreachable, source, destination)`, that are all in all enough to demonstrate our approach.

In both cases, PBT models are very small (55 and 54 SLOC respectively[1], with a shared module where utility functions and data generators are defined, amounting an additional 48 SLOC), and yet reusable in a large amount, if we consider that only the network definition (that is to say, the class constructor) would need to be modified.

Also in both cases, potential issues are reported back the administrator in a human-readable form (cf. Fig. 3b and 4b). Although the simplification process that counterexamples are subjected to using PBT are general and not SDN-specific, they are somewhat equivalent to sub-graph exploration or determination of subnet(s) affected by change that other tools and methods perform.

[1] The models in Fig. 3c and 3c do not contain import sentences and the like, for the sake of simplicity.

```
@invariant()
def fixed_deployment(self):
    print '***  INVARIANT ***'
    result = [True]
    for (p, (source, destination)) in self.policies:
        value = self.net.ping([self.net.get(source),
                               self.net.get(destination)])
        result.append({
            'reachable': (lambda x: (x == 0)),
            'unreachable': (lambda x: (x == 100))
        }[p](value))
    for r in result: assert r
```

(a) Invariant for SDN coherence

```
Falsifying example:
+++ Created network
***  INVARIANT [] ***
+++ Applying new policy: ('reachable', ('h1', 'h12'))
***  INVARIANT [('reachable', ('h1', 'h12'))]***
h1 -> h12
h12 -> h1
*** Results: 0% dropped (2/2 received)
*** ASSERT 0.0 pckg drop between h1-h12
+++ Applying new policy: ('unreachable', ('h1', 'h14'))
***  INVARIANT [('reachable', ('h1', 'h12')),
                ('unreachable', ('h1', 'h14'))]***
h1 -> h12
h12 -> h1
*** Results: 0% dropped (2/2 received)
*** ASSERT 0.0 pckg drop between h1-h12
h1 -> h14
h14 -> h1
*** Results: 0% dropped (2/2 received)
*** ASSERT 100.0 pckg drop between h1-h14 FAILED
+++ Dropping network
----------------------------------------
Ran tests in 25.311s

FAILED (failures=1)
****************************************
```

(b) Counterexample found automatically for unfeasible policy set

```
class MininetStatemFixedSDN(RuleBasedStateMachine):
def __init__(self):
    super(MininetStatemFixedSDN, self).__init__()
    self.net = TreeNet(depth=2,fanout=4,autoStaticArp=True)
    self.net.addLink('s2','s4')
    self.net.addLink('s4','s5')
    self.net.start()
    self.policies = []
    print '+++ Created network: {0}'.format(self.net.keys())

policies = Bundle('policies')

@rule(target = policies, p = mns.policy())
def update_policies(self, p):
    print '+++ Applying new policy: {0}'.format(p)
    self.policies.append(p)

def teardown(self):
    self.net.stop()
```

(c) PBT model

Fig. 4: Example of PBT for SDN validation with fixed deployment

3. Discussion

Our preliminary experiments, reported here, show that modelling SDN network behaviour fits the PBT abstractions nicely. The models we have designed are admittedly simple, as are the examples we have chosen, but represent the first step into more complete, realistic ones.

Randomly generating policies, as we have seen in Fig. 4c, is of course not a very effective nor efficient way of introducing conflict. We do plan on using real cases with industrial deployments and operational sets of policies to further evaluate our approach. In doing so, policies would be instead randomly sampled from such existing sets built or configured by human experts. Similarly, reachability invariants (cf. Fig. 3a) will likely become more complex in larger and more sophisticated SDN topologies, and possibly dynamic as well. However, we are confident that they can be defined in a way that their logic is interpreted from configuration parameters or inferred from PBT model status, maintaining their reusability.

Another aspect to extend our experiments to is the different parametrisations supported by MiniNET. So far, we have relied on the default controller (namely OpenFlow reference controller), but there are several alternatives that can

be configured (i.e. Ryu, NOX, Open vSwitch. . .), and also the option to use custom, externally-run controllers. Given that the choice of controller can significantly impact the behaviour of the SDN [12], this is very relevant. The same applies to topological options: while we have chosen a tree net as basis, there are others (linear, torus. . .), alongside with the possibility of full customisation [11]. Last but not least, switches, hosts and other network components are also fine-tunable, in aspects like link direction, link bandwidth, RTT, etc. An exhaustive review of all use cases and configuration options is unfeasible, considering that SDN deployments vary in size and complexity from small-scale test labs to globally distributed network overlays connecting multiple data centres [4]. However, our approach can be benchmarked against existing solutions on a common data set, sufficiently representative of the real-world complexity.

The examples we have discussed were executed in a 4-core machine (i7 running at 2.4GHz each) with 16GB of RAM, where MiniNET version 2.2.2 was running inside a VirtualBox (the recommended setting) on Ubuntu 19.04. Figs. 3b and 4b report two considerably different test running times: over 45 minutes vs. less than half a minute. PBT tools typically run 100 test sequences in each execution, but given that these are failing runs (that is to say, runs in which failures are found and shrinking is performed), some test sequences will pass but some others will not, and the latter are typically shorter. However, it is hard to further elaborate on this point, because Hypothesis, the PBT tool we have applied, does not excel for its introspection tools. Empirically, we can partially explain the rather noticeable two-orders of magnitude difference between the fixed policies vs. fixed topology experiments due to the timeouts used by MiniNET to determine that a link is down when pinging from one host to another. Since the fixed policies model features setting links up and down as part of the test sequence operations, those timeouts are much more frequently hit than in the case of the fixed policies, where links are stable and it is the policies checking for reachability that are dynamically added to the tests. All in all, a more exhaustive evaluation of test performance with regard to, at least, network size and number of policies, is a required next step to evaluate the validity of this work, especially with regard to its applicability to large scale SDN testing.

Finally, even if we consider the test feedback shown in Figs. 3b and 4b readable, less PBT-experienced developers and practitioners might have a different opinion. Using a tool compatible with MiniNET, such as the one we used to automatically generate the diagrams in Figs. 1 and 2 [9], could be a friendlier alternative and thus have a positive impact on the adoption of this approach. A similarly significant change in this regard would be replacing Hypothesis by a more powerful PBT tool, such as Quviq QuickCheck [10]. Quviq QuickCheck is arguably the most powerful PBT tool at the moment, and features more self-explanatory feedback, in our opinion. Even if this means adding a communication indirection between Erlang (the language of Quviq QuickCheck) and Python (the language of MiniNET), which will have an impact in test performance, the level of documentation, support, and more importantly, of model expressiveness, could very well bring in enough benefits to compensate for the additional running time. Another solution in this case is to use a network simulation tool invokable from Erlang, such as OMNET++ implemented in C++ [5]. Ultimately, for large SDN deployments the choice of the simulation tool will depend on the available support for the most relevant network properties.

While incorporating all these aspects into our approach is adamant to the final applicability and impact of this work, we consider this proof-of-concept illustrative of the potential of PBT in this area.

4. Conclusions

In this paper, we outline the possibility of using PBT as validation strategy for SDN policies. Although the field of work in SDN verification is rather extensive, we believe there are gaps to be filled, especially when it comes to solutions usable in practice.

Our experiments demonstrate that PBT, in particular stateful PBT, can be used to effectively model and test SDN policies from two complementary perspectives: testing enforcement of a fixed set of policies on an evolving SDN, and conversely, testing enforcement of evolving set of policies on a fixed SDN.

We intend to keep exploring the potential of PBT in the context of SDN policies validation. In particular, we plan to extend our work to massive SDNs and very large sets of policies, as well as to benchmark the performance of PBT models testing on a large scale.

Acknowledgements

This work was financially supported in part by the Swedish Foundation for Strategic Research, grant RIT17-0035.

References

[1] Derrick, J., Walkinshaw, N., Arts, T., Benac Earle, C., Cesarini, F., Fredlund, L.A., Gulias, V., Hughes, J., Thompson, S., 2010. Property-Based Testing - The ProTest Project, in: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (Eds.), Formal Methods for Components and Objects, Springer Berlin Heidelberg. pp. 250–271.

[2] Fink, G., Bishop, M., 1997. Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes 22, 74–80.

[3] Hughes, J., 2007. Quickcheck testing for fun and profit, in: Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages, Springer-Verlag, Berlin, Heidelberg. pp. 1–32.

[4] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., et al., 2013. Experience with a globally-deployed software defined WAN, in: ACM SIGCOMM Computer Communication Review, ACM. pp. 3–14.

[5] Klein, D., Jarschel, M., 2013. An openflow extension for the omnet++ inet framework, in: Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium. pp. 322–329. URL: http://dl.acm.org/citation.cfm?id=2512734.2512780.

[6] Lantz, B., O'Connor, B., 2015. A Mininet-based Virtual Testbed for Distributed SDN Development. SIGCOMM Computer Communication Review 45, 365–366.

[7] MacIver, D.R., 2019. Hypothesis. URL: https://hypothesis.readthedocs.io.

[8] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., 2008. Openflow: Enabling innovation in campus networks. SIGCOMM Computer Communication Review 38, 69–74.

[9] Narmox Spear, 2019. Mininet topology visualizer. URL: http://demo.spear.narmox.com/app/?apiurl=demo#!/mininet.

[10] QuviQ AB, 2019. Quviq quickcheck. URL: http://www.quviq.com.

[11] Scott, L., 2019. Custom Mininet Topologies and Introducing Atom. URL: https://inside-openflow.com/2016/06/29/custom-mininet-topologies-and-introducing-atom/.

[12] Wang, S., Chiu, H.W., Chou, C.L., 2015. Comparisons of SDN OpenFlow Controllers over EstiNet: Ryu vs. NOX, in: Fourteenth International Conference on Networks, pp. 244–249.

[13] Wang, Y., 2017. TenantGuard: Scalable Runtime Verification of Cloud-Wide VM-Level Network Isolation. Master's thesis. Concordia University. URL: https://spectrum.library.concordia.ca/982513/.

[14] Yousaf, F.Z., Bredel, M., Schaller, S., Schneider, F., 2017. NFV and SDN—Key Technology Enablers for 5G Networks. IEEE Journal on Selected Areas in Communications 35, 2468–2478.