Towards Test-Driven Software Defined Networking

David Lebrun, Stefano Vissicchio, Olivier Bonaventure
Université catholique de Louvain, ICTEAM
Email: firstname.secondname@uclouvain.be

***Abstract*—To configure, troubleshoot and operate their networks, operators often have no alternatives than relying on error-prone manual procedures. The emerging Software Defined Networking paradigm opens new possibilities for more structured networking methodologies. We argue that provably-effective practices can be borrowed from more developed engineering fields, especially software engineering.**

In this paper, we propose an adaptation of test-driven software development methodologies to software defined networks (SDNs). To support our methodological guidelines, we propose an expressive requirement formalization language. Further, we describe a prototype tool able to check the compliance of an SDN controller with requirements expressed in the proposed language. Our evaluation of the prototype shows promising results on the practical viability of our approach.

I. Introduction

Despite the criticality of computer networks, the tools used by operators to design, operate and manage their networks are still fairly basic. Mostly, operators rely on low-level configuration files held by network devices, and they tend to miss high-level network requirements and objectives. Often, the network requirements themselves are imprecisely defined [1]. Indeed, it is not rare that "specifications" produced by network operators consists in a limited set of PowerPoint slides.

With the growing development of Software Defined Networking, computer networks are currently on the verge of an epochal paradigm shift. If this shift occurred, networks would be completely managed and operated by logically centralized software controllers. The promise is to deploy a more flexible interface to program the network behavior, and to unlock vendor dependencies. Beyond providing those powerful management knobs, we argue that Software Defined Networking offers the possibility of adapting software engineering principles and practices to improve network design, configuration and test. This would enrich the networking practices with solid and effective methodologies like those defined by software engineers to develop mission critical systems.

In this paper, we make a first step towards a test-driven methodology for Software Defined Networks (SDNs). Our methodology is inspired by software deployment best practices, that are based on the formal definition of requirements and identify iterative testing practices as a guide [2], [3]. In those approaches, well-defined test suites are indeed used to identify bugs as quickly as possible during the software development and verify that code modifications do not disrupt previously accommodated requirements. Similarly, in our methodology, a central role is played by automated network testing of formalized requirements. The network testing activity is prescribed before applying any change to the (controller of the) network. To enable frequent and safe network tests, we rely on the reproducibility of SDNs to build a testing SDN analogous to software test environments.

Our current proposal is tailored to data path requirements, i.e., requirements on the shape of forwarding paths followed by user traffic flows. Our choice is motivated by the primary importance of this kind of requirements, which resemble functional requirements in computer networks (e.g., by permitting and denying communication between end hosts). Moreover, they can be powerful enablers for network function virtualization [4] and can reflect a wide range of important network objectives, like fine-grained traffic engineering, security and service chaining via traversal of middle-boxes [5], [6], [7].

After presenting an overview the test-driven methodology (Section II), this paper develops the following contributions.

Firstly, to allow automation and reproducibility of network tests, we define a high-level formal language that allows operators to express complex data path requirements as it would be needed for SDNs (Section III). Our proposed language, which we called Data Path Requirement Language (DPRL), extends the previously proposed FML language [8] to support arbitrary constraints on data paths. Such a support is provided by new constructs and by the definition of a regular grammar allowing the specification of network devices and links that have to be traversed either in sequence or in alternative.

Secondly, we describe and evaluate a prototype requirement checker for automated testing of data path requirements (Section IV). To ensure independence from hardware implementations, our prototype currently uses a Mininet emulated network [9] as testing SDN. Starting from DPRL statements, the checker generates the corresponding test packets, injects them into the testing SDN, tracks the data path followed by each test packet, and checks whether all the specified network requirements are satisfied. In particular, to check requirement accommodation, the prototype transforms DRPL statements into automata, and verifies that the string representing the data paths followed by each test packet is accepted by the corresponding automaton. We perform an experimental evaluation of our prototype. The prototype currently has limited testing abilities, and our experiments were not targeted to a complete evaluation of the tool itself. However, our evaluation highlights functional correctness and good scalability properties of our prototype, and shows the practical viability of automated pre-deployment tests for SDNs.

Finally, we describe how our proposals complement research efforts both in traditional networks and in SDNs (Section V), and we draw the conclusions (Section VI).

Stefano Vissicchio is a postdoctoral researcher of the Belgian fund for scientific research (F.R.S.-FNRS).

II. A Methodology for Testing Data Path Requirements in Software Defined Networks

One basic function of computer networks is to allow (or deny) data exchanges between remote end points, e.g., hosts and servers. By configuring network devices, operators can decide paths on which data are forwarded. Different data paths correspond to different performance and resource utilizations.

In the following, we refer with *data path requirements* to constraints on the shape of data paths that have to be enforced between pairs of sources and destinations in a network. An example of data path requirement is reported in the bottom part of Fig. 1, while the rest of the figure depicts the topology of a network to which the requirement applies. The requirement imposes the traversal of a sequence of middle-boxes, consisting of one network monitor and one firewall. Note that some middle-boxes (i.e., the two firewalls) can alternatively be present in the data path.

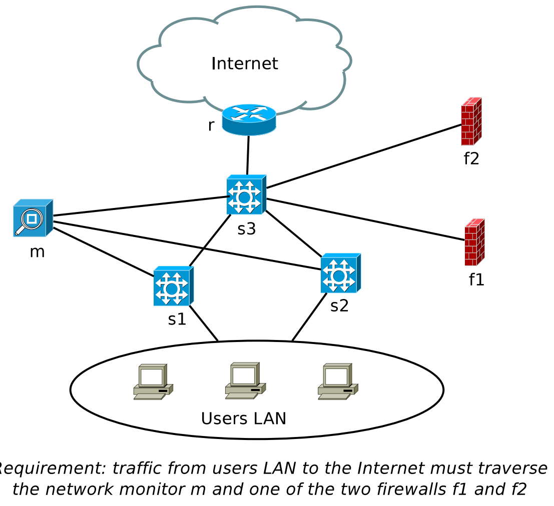
Internet
r
f2
s3
m
f1
s1
s2
Users LAN
Requirement: traffic from users LAN to the Internet must traverse the network monitor m and one of the two firewalls f1 and f2

Fig. 1. Example of data path requirement.

Data path requirements are crucial in many networks. A basic data path requirement is represented by binary connectivity between end points. Binary connectivity corresponds to services like VLANs in switched Ethernet networks or VPNs in service provider networks. Moreover, this constraint must be enforced in many enterprise networks, often for security reasons [5]. Nevertheless, network operators' objectives go far beyond binary connectivity. For example, when an operator configures IGP weights, she indicates her preference for some paths over others, typically for traffic engineering purposes. In enterprise networks, operators often need to force traffic to pass through specific middle-boxes [7] or avoid some parts of the network [10]. For example, traffic to a server may always need to traverse a firewall. In some networks, the traffic must be load-balanced between several firewalls for performance reasons. In banks, security policies might require access to an e-banking server to pass through two consecutive firewalls from different vendors. In other enterprises, Web access must go through one of the enterprise proxies while IPSec traffic can exit the network without constraint and other services must pass through a `socks` proxy. This kind of policy steering deserves more and more attention and promises to be a key point in favor of SDN adoption [6].

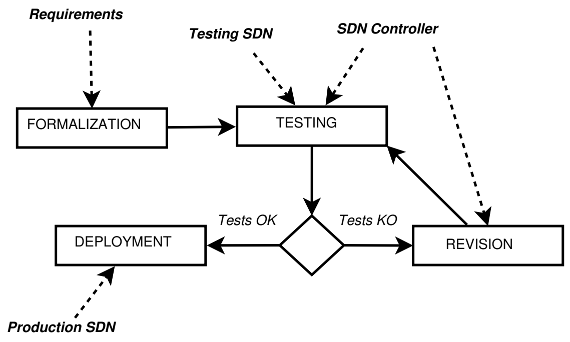

Fig. 2. Block diagram of our methodology.

Software Defined Networking promises to enable more fine-grained control of traffic flows through network logic centralization. In SDNs, centralized software controllers are responsible for computing data paths for each incoming traffic flow, and for configuring network devices accordingly. Logic centralization effectively translates to the possibility to accommodate more complex requirements, e.g., by exploiting Network Function Virtualization [4]. However, implementing and deploying an SDN controller that satisfies specific requirements are still open problems.

We propose to fill this gap for SDNs by relying on a test-driven approach. In particular, we propose to rely on the reproducibility of SDNs to check the compliance of SDN controllers with given data path requirements. Indeed, SDN logic centralization eases the construction of a testing network which is a one-to-one copy of a given production network, and to test the production software on such a testing network (see, e.g., [11]). Testing networks represents controlled environments, similar to test software deployments, in which unverified controllers do not disrupt user traffic. Moreover, testing SDN do not need production-level equipment as they can be built on emulation platforms like Mininet [9].

Fig. 2 shows an overview of our approach, as a block diagram of the activities to be performed each time a change occurs to network requirements or to SDN controllers. Activities are identified by rectangles, and precedence constraints between activities are represented by solid arrows. The input to each activity is specified using text in bold and dashed arrows. The first step in our approach is to formalize the data path requirements. The formalization has a documentation function and enables automatic verification of requirements on the testing network. Tests are prescribed to be run on the testing network *before the deployment* of any change in the network. A revision activity, to tweak the intended controller modification, has to be performed if some tests failed.

By enabling verification of one requirement at a time, this methodology supports an equivalent of software unit tests, targeted to check the correctness of single software functions. Moreover, our approach supports regression testing, i.e., checking that the accommodation of new or modified requirements does not negatively affect the satisfaction of all the unmodified requirements. Indeed, the requirement specification can be evolved over time following the evolution of requirements, and comprehensive sets of tests can be run on the full requirement specification each time a requirement is added or changed.

In order to show the viability of the proposed approach, we define a formal language to express data path requirements, and we developed a prototype software supporting the automatic testing phase. We provide more details on those contributions in the following sections.

III. A FORMAL LANGUAGE FOR DATA PATH REQUIREMENTS

The first component for any test-driven methodology is a model. According to [3], a model should be compact, predictive, semantically meaningful and sufficiently general. Network operators rarely use such models. They typically use graphs with different types of attributes to represent their network topology. However, the main complexity of the network lies in the configuration of all the network devices. Such configuration does not meet the requirements for a good model. In the following, we propose a better model based on a tailored language to express network requirements.

A. FML

The starting point of our proposal is the FML language proposed in [8]. The basic building blocks in the FML language are predicates and rules. A predicate is a symbol, or keyword, associated with a fixed number of arguments. An FML rule is defined as follows.

$$h \Leftarrow [\neg]b_1 \wedge \cdots \wedge [\neg]b_n$$

The left part of the rule, i.e., h, is called head and is a predicate. The right part is the body of the rule. Each b_i, with $i = 1, \ldots, n$ is a boolean expression matching the value of a header field in incoming packets. Intuitively, each rule represents an if-then statement, in which the body defines the condition under which the rule applies (the *if*) and the head defines the type of action to be performed (the *then*).

A context of predicates is associated to each rule. A rule is well-formed if all the variables in the body appear in the head, and if h does not contain any predicate $g \in G$, where G is its associated context. Moreover, the first argument of the head must be $\overline{F}$, to indicate that the predicates apply to all the flows $\overline{F}$ matching the conditions expressed in the body of the rule. More precisely, $\overline{F}$ is defined as a vector containing several values that can be used in the boolean expressions in the body of any rule. Indeed, a rule applies to all and only the flows whose values match the conditions expressed in the body of the rule. In the following examples, we focus on the source of the flow H_s, the destination of the flow H_t and the protocol $Prot$.

An FML specification consists of the context definition and of a list of rules. Network devices are identified by unique names. In general, a flow can match the conditions expressed by several rules. In such a case, there is a conflict between rules to be applied to the flow. In the original FML specification, a precedence between rules has to be defined on a per-application basis. Then, for each flow, the conflict resolution mechanism keeps only the constraints with the highest priority. In the following, we always assume that a sequential precedence between rules, i.e., the rules specified before have higher preference.

B. DPRL

We extend the syntax of the FML language in several ways, in order to support intuitive and high-level definition of non-trivial data path requirements. We called the resulting language Data Path Requirement Language (DPRL). In the following, we discuss our extensions one at the time.

1) Expression of arbitrary data paths: FML defines predicates to allow or deny binary connectivity, and to indicate the need for a flow to traverse a single waypoint. However, many current networks and future SDNs would need support for more complex data path requirements, like traversal of multiple middle-boxes in sequence or in alternative. To this end, DPRL extends FML for capturing different kinds of data path requirements, including (i) binary connectivity, (ii) strict source routing across a fixed list of network nodes, (iii) loose source routing in which traversal of specific nodes has to be guaranteed (e.g., for traffic steering through middle-boxes), and (iv) path length restriction (e.g., for performance purposes).

To express this wide range of data path requirements, we add the $path$ keyword to the $allow$ and $deny$ predicates defined in FML for binary connectivity. The $path$ predicate takes one argument p in addition to the specification of the matching flows $\overline{F}$. Hence,

$$path(\overline{F}, p) \Leftarrow [\neg]b_1 \wedge \cdots \wedge [\neg]b_n$$

The argument p is a string representing the valid paths for the flows matching the b_i boolean expressions. More precisely, p is a string belonging to a tailored regular language which we describe in the following.

Our regular language is defined on the alphabet Σ of the language is constituted by the set of all the network device identifiers, plus a special symbol . (dot), representing any (unspecified) device. The language includes three operators.

- The concatenation operator "," that applies to two device identifiers and is equivalent to a logical conjunction. In particular, given two device identifier s and t, s, t indicates that t must follow s on the path.
- The union operator "|" that applies to two identifiers and is equivalent to a logical disjunction. Given two device identifier s and t, $s|t$ translates to having either s or t in the path.
- The Kleene star operator "*" that matches the preceding symbol zero or more times. For example, it can be used in combination with the dot symbol to match "any node, any times".

Normally, the Kleene star operator has to be applied before the union operator, which in turn has precedence over the concatenation one. Parentheses are used to indicate custom precedences between operators. For example, $(a|b)^*$ postpones the application of * after the evaluation of $a|b$.

Observe that strings of this regular language can be analyzed in linear time. This is the main reason why we decided not to rely on standard regular expressions like those supported by PCRE [12]. Indeed, the language generated by regular expressions cannot be expressed by a Type-3 grammar in the Chomsky hierarchy, hence making the analysis of regular expressions computationally inefficient.

Also, note that, by definition of our regular language, the $path$ keyword generalizes the $waypoint$ predicate defined in FML to impose the traversal of a single device. Indeed, $waypoint(\overline{F}, s) \Leftarrow \ldots$ can be expressed by a DPRL rule having $path(\overline{F}, \text{'}.*, s, .*\text{'})$ as head. However, the $path$ keyword can express more complex data path requirements like traversal of multiple network nodes, in sequence or in parallel. For example, the requirement in Fig. 1 is impossible to express in FML, while it is straightforward to express with a DPRL rule whose head is $path(\overline{F}, \text{'}.*, m, .*, f1|f2, .*, r\text{'})$.

Other examples of data path requirements for the network in Fig. 3 follow. Those examples are meant to illustrate the expressive power of our regular language for practical networking goals. For simplicity, all the requirements are on the path to be followed from X to Y.

- Source routing, e.g., the flow must traverse all and only $s1, s3, s4$ and $s6$
  $path(\overline{F}, \text{'}s1, s3, s4, s6\text{'}) \Leftarrow H_s = X \wedge H_t = Y$
- Middleboxing, e.g., the flow must cross $s3$
  $path(\overline{F}, \text{'}.*, s3, .*\text{'}) \Leftarrow H_s = X \wedge H_t = Y$
- Middleboxing with multiple waypoints, e.g., the flow must pass through $s3$ and $s4$, in this order
  $path(\overline{F}, \text{'}.*, s3, .*, s4, .*\text{'}) \Leftarrow H_s = X \wedge H_t = Y$
- Link traversal, e.g., the flow must traverse link $(s3, s4)$
  $path(\overline{F}, \text{'}.*, s3, s4, .*\text{'}) \Leftarrow H_s = X \wedge H_t = Y$
- Endpoint restriction, e.g., the data path must start at $s1$ and end at $s6$
  $path(\overline{F}, \text{'}s1, .*, s6\text{'}) \Leftarrow H_s = X \wedge H_t = Y$
- Node avoidance, e.g., the flow must not cross $s2$
  $path(\overline{F}, \text{'}(s1|s3|s4|s5|s6)*\text{'}) \Leftarrow H_s = X \wedge H_t = Y$
- Path length restriction, e.g., the data path must have exactly 3 hops
  $path(\overline{F}, \text{'}., ., .\text{'}) \Leftarrow H_s = X \wedge H_t = Y$

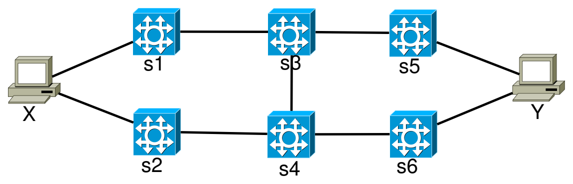

Fig. 3. Example network for path constraints

Observe that only the loose source routing with a single waypoint can be expressed in FML, even if they could be useful in realistic network settings.

2) Expression of alternative requirements: FML directly supports the specification of one requirement per flow. Indeed, if multiple requirements concern the same flow, they are solved with a conflict resolution mechanism which keeps only one of the matching requirements. However, in real-world cases, data path requirements may be enforced in alternative. For instance, this case occurs when requirements have to be specified on backup paths, e.g., to be used in case of critical network load or in case of a network failure, for the same flow. Moreover, alternative requirements can apply to different flows. For example, a network operator may want that a gateway g_1 is traversed either by all the flows from a source LAN L_1 or by all the flows from a source LAN L_2, in order not to mix flows from L_1 and L_2. Similarly, given a source LAN L, it may be desirable that either all the flows from L traverse a single gateway g_1, or that half of the flows from L traverse gateway g_2 and the other half are routed through g_3. This may be due to performance or capacity restrictions of g_2 and g_3. To support requirements on alternative data paths and alternative requirements, we introduce prioritized sets of rules in DPRL.

Informally, a *set* is a group of rules that have to be satisfied in a coordinated way. More precisely, a set is a group of rules, such that each rule is assigned an integer value denoting the *subset* it belongs to. A subset S' of a set S is satisfied if all the rules belonging to S' are satisfied. A set is satisfied if one and only one subset is satisfied. Each rule in a set is defined as $: set : subset : rule$, where set is the name of the set and $subset$ is the name of the subset of the rule. Intuitively, subsets correspond to alternative groups of requirements that have to be satisfied at the same time. Consider, for example, the set $grptest$ defined by the following statements.

$$\begin{aligned} &: grptest : 1 : h \Leftarrow ...(a) \\ &: grptest : 1 : h \Leftarrow ...(b) \\ &: grptest : 2 : h \Leftarrow ...(c) \\ &: grptest : 3 : h \Leftarrow ...(d) \end{aligned}$$

Subset 1 is satisfied if either rules a and b are both satisfied. The subsets 2 and 3 are satisfied if rules c and d, respectively, are satisfied. The entire set $grptest$ is satisfied if one and only one level is satisfied.

3) Syntactic sugar: Finally, in order for high-level specifications to be expressed more easily in DRPL, we add support for custom aliases. An *alias* is a syntactic shortcut for a boolean expression without variables. Formally, we denote $a : b$ an alias in which a is a shortcut for the boolean expression b. Aliases can be used inside the body of DPRL rules, with the purpose of making rules clearer and closer to requirement definition. When a rule will be evaluated, however, aliases have to be replaced using their respective definitions. The easiest aliases substitute device identifiers As an example, an alias $firewalls : f_1 \vee f_2$ can be used to simplify the DPRL rule corresponding to the requirement in Fig. 1. Protocols can also be defined as aliases. For example, we can define $HTTP$ as $HTTP : proto = tcp \wedge port = 80$. Finally, boolean expressions on paths can be grouped together using aliases. Such path aliases can be passed to the $path$ predicate as arguments. This is often useful for readability and conciseness if the same group of paths has to be imposed to several flows, as in the following example.

$$\begin{aligned} &paths1 = \text{'}s1, s2, .*\text{'} \vee \text{'}s3, s4, .*\text{'} \\ &path(\overline{F}, paths1) \Leftarrow H_s = X \wedge H_t = Y \\ &path(\overline{F}, paths1) \Leftarrow H_s = X_1 \wedge H_t = Y_1 \\ &path(\overline{F}, paths1) \Leftarrow H_s = Y \wedge H_t = Y_1 \end{aligned}$$

Path aliases are replaced by convenient sets at compilation time. In the example above, for instance, the alias $path1$ has to be replaced by a set with two subset containing $\text{'}s1, s2, .*\text{'}$ and $\text{'}s3, s4, .*\text{'}$ respectively.

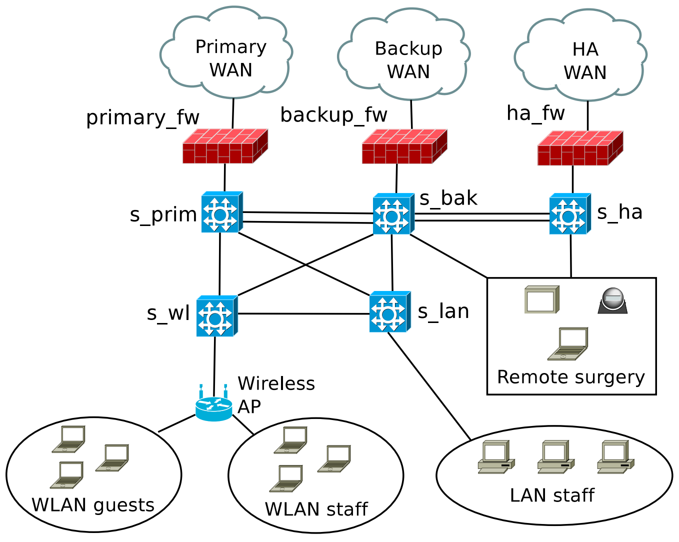

Fig. 4. A hospital network.

C. DPRL Example

To illustrate the expressiveness of DPRL, we now provide a formalization of realistic requirements for a large hospital network. The network is depicted in Fig. 4. The hospital has three WAN connections, i.e., a primary, a backup, and a special high availability (HA) WAN used for remotely controlled surgeries. Each WAN is accessed via a different router (not shown in the figure). The internal LANs comprise guests, wireless staff, wired staff and operating rooms. The main requirements are: (1) guests and staff connected through a WLAN can only access the Internet, (2) hosts in the LAN staff can access any other subnetwork but the remote surgery one, (3) the remote surgery room cannot be accessed from the outside, (4) the remote surgery can exchange only TCP packets with destinations reachable through the WAN, (5) the flows from and to the operating room must go through the HA WAN connection, i.e., they can be forwarded through the primary or backup connections if and only if the HA WAN cannot be used to reach the flow destinations.

All these requirements can be simply expressed in DPRL as in Fig. 5. Lines starting with # identify comments. Moreover, $prim\_wan\_id\_range, bak\_wan\_id\_range$ and $ha\_wan\_range\_id$ indicate a range of host identifiers (e.g., IP prefixes) reachable through the primary, the backup and the HA WAN respectively. Note that expressing the same requirements in FML would have been impossible because of the absence of aliases, sets and the *path* keyword.

# Define aliases for hosts
$wlan : wlan\_ap\_id$
$staff : h_1 \vee \cdots \vee h_m$
$surgery : i_1 \vee \cdots \vee i_n$
$ha\_wan : ha\_wan\_id\_range$
$wan : prim\_wan\_id\_range \vee bak\_wan\_id\_range \vee ha\_wan$
# WLAN access restrictions
$allow(\overline{F}) \Leftarrow H_s = wlan \wedge H_t \in wan$
$allow(\overline{F}) \Leftarrow H_s \in wan \wedge H_t = wlan$
$deny(\overline{F}) \Leftarrow H_s = wlan$
# Isolate operating room
$allow(\overline{F}) \Leftarrow H_s = surgery \wedge H_t \in wan \wedge Proto = TCP$
$allow(\overline{F}) \Leftarrow H_s \in wan \wedge H_t = surgery \wedge Proto = TCP$
$\quad \wedge\, Tcp\_syn = 0$
$deny(\overline{F}) \Leftarrow H_t = surgery$
# operating room flows go via HA connection if possible
$: out : 1 : path(\overline{F}, {}'s\_ha{}') \Leftarrow H_s = surgery \wedge H_t \in ha\_wan$
$: out : 2 : path(\overline{F}, {}'.*, s\_prim{}') \Leftarrow H_s = surgery \wedge H_t \in wan$
$\quad \wedge\, H_t \notin ha\_wan$
$: out : 3 : path(\overline{F}, {}'.*, s\_bak{}') \Leftarrow H_s = surgery \wedge H_t \in wan$
$\quad \wedge\, H_t \notin ha\_wan$
$: in : 1 : path(\overline{F}, {}'s\_ha{}') \Leftarrow H_t = surgery \wedge H_s \in ha\_wan$
$: in : 2 : path(\overline{F}, {}'.*, s\_prim{}') \Leftarrow H_t = surgery \wedge H_s \in wan$
$\quad \wedge\, H_s \notin ha\_wan$
$: in : 3 : path(\overline{F}, {}'.*, s\_bak{}') \Leftarrow H_t = surgery \wedge H_s \in wan$
$\quad \wedge\, H_s \notin ha\_wan$

Fig. 5. Requirements for the hospital network in Fig. 4.

IV. From Requirements to Tests

Section III describes a language to formalize network requirements. We now describe how to automatically translate formalized requirements in pre-deployment tests.

A. Design and Implementation

To evaluate the feasibility of a test-driven methodology in SDNs, we developed a prototype checker, which we called TASTE (Toolkit for Automated Sdn TEsting) [13]. The goal of TASTE is to test compliance of an input SDN controller with given formalized requirements. To this end, our prototype generates test packets from DPRL rules, injects those packets in an emulated network, and collects and analyzes test packet traces. The tool is designed by assuming that it is logically separated from the tested controller and that it can connect to all the switches in the testing SDN.

Our prototype runs on top of the Mininet platform [9]. The software has been implemented from scratch and is composed of about 2,500 lines of python code. The source code of our prototype is available at [13]. Our implementation includes the following components. Two basic components of the tool are devoted to the support of languages and protocols. The first of those two components implements a subset of the OpenFlow protocol to interact with OpenVSwitch. The second one is a parser of network requirements expressed in DPRL. It supports all the constructs described in Section III and uses an input file to map DPRL device identifiers to MAC addresses of OpenFlow switches. However, it currently supports comparison on a subset of flow values, namely the source of the flow H_s, the destination of the flow H_t and the protocol $Prot$. While this would be a major limitation for a real tool, we considered it as tolerable since we focused on showing the feasibility of our proposed test-driven approach.

The core components of the prototype are the ones responsible for generating and running the network tests. Fig. 6 shows

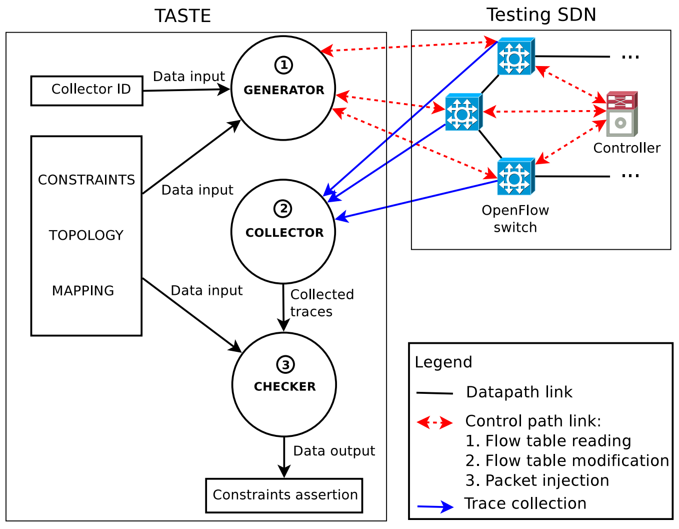

Fig. 6. TASTE architecture.

an overview of those core components and of their interactions.

To generate test packets, our prototype includes a test *generator* module. The test generator maps each DRPL rule to test packets that match the conditions expressed by the body of rule. Moreover, it is responsible for limiting the number of generated test packets. In our prototype, test packets are UDP packets destined to the port of the protocol specified in the considered rule. If no protocol is specified, then we use port 64242, i.e., an arbitrary port which is not a well-known one. Observe that this approach is sufficient to check whether a single protocol (or basic communication) is allowed or denied between a given pair of hosts. However, in some cases, more test packets per rule may be needed, e.g., to guarantee that all possible protocols are denied between a certain pair of hosts. As a first step to limit the duration of the testing phase, the generator currently includes a simple test reduction technique that uses only one test packet for checking all the DRPL rules with the same body. We plan to improve our prototype in future work, by investigating efficiency and scalability issues related to the number of test packets, and adding support for more refined network test reduction techniques (e.g., [14], [15], [16]).

Another important task of the generator is to modify the flow tables of the switches in testing SDN so that the path followed by test packets can be tracked. In particular, the generator configures the switches in the testing SDN to send a truncated copy of transmitted packets to another component of the tool, i.e., the collector module. Observe that the packet tracking mechanism we implemented is similar to the NDB tool described in [17]. However, we generate the packets corresponding to the specified requirements while NDB processes only the regular packets transmitted inside the network. Moreover, NDB directly outputs the collected data to the user (e.g., packet backtrace), while our approach is to use them as an input for automatic requirement compliance checks.

The *collector* module has two main responsibilities. First, it injects test packets in the testing SDN, once the test packets are generated and the switches are properly configured to support packet tracking. Second, it collects all the truncated copies of all the injected packets.

Finally, the *checker* component is responsible to check compliance of collected traces with the specified requirements. From the traces, it reconstructs the path taken by each test packet and checks that they do not violate any requirement. Moreover, our implementation supports the test of any data path requirement defined in Section III. In particular, binary connectivity constraints are verified by checking that the first switch in the path is connected to the source host and the last switch is connected to the destination host. If this is the case, then the *allow* constraint is verified. Conversely, if the edge switches are not connected to the source or destination hosts, the *deny* constraint is satisfied. Indeed, a switch sends a packet copy to the controller if and only if the original packet is actually forwarded by the switch. To check generic path constraints, we exploit the regularity of our constraint language. Namely, the checker component builds a deterministic finite automaton corresponding to each path constraint. The automaton is able to accept or reject any path in linear time [18]. The construction of the each automaton is performed in three steps. First, the expression is transformed into a NFA (Nondeterministic Finite Automaton) by using the Thompson algorithm [19]. Second, this NFA is transformed into a DFA (Deterministic Finite Automaton). Finally, the DFA is optimized by removing unreachable states or merging indistinguishable states.

By design, our prototype is suitable for checking whether a given data path requirements is accommodated by a set of forwarding rules (e.g., the ones configured on production network devices), or by a stateless SDN controller whose logic does not depend on previous traffic received by the SDN. Moreover, it supports testing of advanced SDN functions like arbitrary header field re-writing. Indeed, if tests are performed sequentially, then each test packet can always be tracked in the testing SDN, independently of the modifications it undergoes. Finally, it can be used to check distributed SDN controllers, i.e., it does not rely on any assumption on the number of controllers installing rules in the same network.

On the flip side, our current prototype does not immediately fit the case of stateful controllers which behave differently depending on incoming traffic flows. In this case, additional modules are needed, i.e., to extract (for example, through symbolic execution) the conditions (for instance, the number of incoming flows) under which the given stateful controller installs different entries in the switches, and to recreate those conditions in the testing network. Similarly, we expect that the current architecture can be extended to test other requirements. For example, a measurement module can be added to check dynamic load balancing requirements, e.g., traffic split over a set of middleboxes depending on the number of incoming flows. Since in this paper we aim at showing the feasibility of an SDN test-driven approach for data path requirements, a full investigation of such architectural extensions are left for future work.

B. Evaluation

We evaluated our prototype by performing functional, performance and scalability analyses. We stress that the goal of our evaluation is to show the viability of a test-driven approach for networking, and not to fully evaluate the performance of our prototype software. All the tests were performed with Mininet in a virtual machine with 1.5GB of RAM and a CPU i7 620M @ 2.67GHz (two cores, four threads with HyperThreading, all four were allocated to the virtual machine as virtual CPUs). Our prototype used Python 2.7.1. For the sake of brevity, we now report a part of the evaluation we performed on our prototype. Other results can be found in [13]. They show the ability of our prototype to pinpoint mismatches between DPRL rules and SDN controller implementation and apply to different topologies.

We performed *functional and performance analyses* of our tool with respect to specific network topologies. In all our tests, we used rather simple topologies, since effectiveness and scalability of our prototype do not depend on topology design nor on the SDN implementation, that are both treated as black box in our approach. Consider the topology shown in Figure 7, on which the following constraints must hold.

1) The path between `s1` to `s4` must go through `s2` and the reverse path via `s3`.
2) The path from `s1` to `s5` must go through `s3` and the reverse path via `s2`
3) The path from `s2` or `s3` to `s6` must go through `s4` or `s5`.
4) All the paths must contain the minimum number of hops.

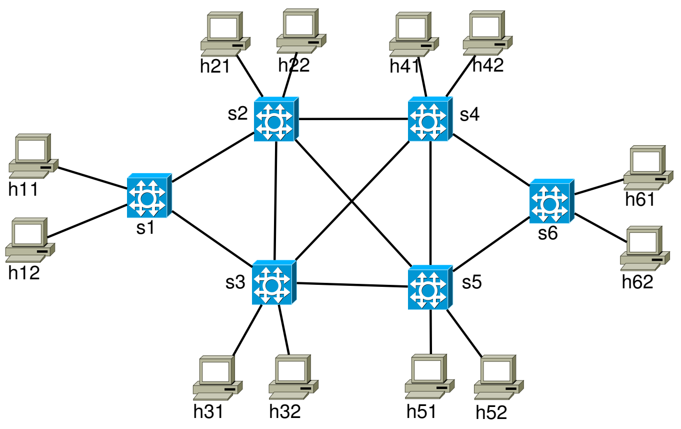

Fig. 7. Test network for path constraints

We coded an OpenFlow controller that modifies the configuration of the switches upon the arrival of new flows. Then, we verified the correctness of such an OpenFlow controller by running our prototype checker. Table I reports the time required (in milliseconds) to generate, inject, collect and verify the test packets.

Phase	**Elapsed time (in ms)**
Test Generation	69.27
Packet Injection	53.37
Packet Collection	38.72
Checking	4.73

TABLE I. BREAKDOWN OF THE TIME TAKEN BY OUR PROTOTYPE FOR REQUIREMENT VERIFICATION ON THE NETWORK IN FIG. 7.

We also performed *scalability analyses*. Observe that the test generation and checking phases are the most critical ones from a scalability point of view. The test packet generation depends on the number and type of requirements and on the size of the testing SDN. However, generation of many tests can be factorized over multiple runs of the tool. Indeed, test packets associated to each requirement (i.e., DPRL rule) can be stored after the first time they are generated, so that they do not have to be re-generated for each single change of a requirement or of the network topology.

The checking phase depends both on the requirements and on the size of the network. In turn, translating requirements into automata is a critical operation of the checker from a performance viewpoint. With a naive implementation, the dot character would be replaced by a union of all possible switches. As such, the complexity of the checking phase would rapidly grow with the number of devices in the network. For example, with this naive approach, the building time of an automaton containing one dot character over a alphabet of 512 symbols (e.g., switches) takes about 360 seconds. We optimized the construction of the automaton by considering the dot character as any other symbol, i.e., not expanding it. Moreover, a dot character overrides any other transition from a given state. With our optimization, the construction time for an automaton with one dot character over an alphabet of 512 symbols then drops to less than one millisecond.

Requirements	**Checking time (in sec)**
13,699	2.82s
109,600	12.74s
986,409	103.4s

TABLE II. CHECKING TIME ON LARGE NUMBERS OF DATA PATH REQUIREMENTS (EACH REQUIREMENT IMPOSES A GIVEN DATA PATH).

We performed further scalability tests of the checking phase by generating a large number of paths and defining one requirement per path. Table II reports the results of those scalability experiments. The table shows reasonably low checking times even for huge numbers of requirements, i.e., around 100 seconds for almost $1,000,000$ requirements. We stress that those results suggest that the total time taken by our prototype for the checking phase seems to increase linearly with respect to the number of requirements.

V. RELATED WORK

Our main contributions span a networking methodology, a formal language to express network requirements, and a testing framework, and link them together. Previous contributions typically focus on one of those three areas, hence we analyze them separately in the following.

Design and deployment methodologies are rarely the main target of research efforts. Moreover, the few papers in this area often restrict to limited and specific use cases (see, e.g., [20]). As such, methodologies available to operators are often the ones proposed by vendors on the basis of their consultancy activity. Despite those methodologies retain a great experience value, they often resemble old software methodologies and cost-efficient testing techniques (see, e.g., [21], [22]).

In particular, current network methodologies overlook requirement formalization. Recently, several research efforts aimed at progressively filling this gap by proposing high-level languages tailored to SDNs [8], [23], [24]. Some of those efforts also focuses on some data path requirements [5], [6], [25]. This paper is mainly complementary to those works. On one hand, our DPRL language can express more complex data path requirements, like alternative requirements on backup paths, than previously proposed languages. On the other hand, the automated generation of SDN controllers, which is the main target of that works, can be easily encompassed in our test-driven methodology and would be strengthened by our testing framework.

The current lack of methodologies and requirement formalization is also reflected in the limitation of current network testing practices. There are no well-established theory and tools that enable to efficiently design and run network tests. Most operators rely on lab tests that are often described as a sequence of manual operations [22]. Moreover, the network tests performed by operators are often limited to run commands as simple as `ping` and `traceroute`, to manually double-check device configurations or to debug which routes are received by which devices. Unsurprisingly, configuration errors are often the cause of network misbehavior and downtime [26].

Shortcomings in current testing practices are attracting more and more attention from both the industrial and the research community. In the last years, various testing techniques have been proposed. Some of them (e.g., [27], [28], [29]) analyze the configuration of traditional networks and are hard to extend to SDN controller verification, e.g., because bound to specific network protocols. Others propose testing frameworks targeted to SDN (e.g., [30], [16]). However, they are typically tailored to efficient reactive testing, i.e., to timely detect SDN policy violations. This approach is comparable to admitting latent errors in software, and may not be suitable for some network objectives, e.g., security or strict performance constraints. We argue that well-established and effective proactive testing is also needed, both to ensure accommodation of crucial requirements and to improve network design and management practices. Finally, techniques based on model checking and symbolic execution have been proposed to assess the correctness of SDN controller software (e.g., [31], [32]). Our framework includes a more expressive and high-level formal language to express correctness properties (stated as snippets of Python code in [31]), and our approach is more flexible, e.g., it directly applies to distributed SDN controllers. Nevertheless, we plan to explore the integration of previous testing techniques in our framework in future work, e.g., to improve efficiency and scalability of our prototype.

VI. Conclusions

Software Defined Networks (SDNs) are expected to change the way networks are configured and managed. This opens new possibilities for more formal and structured methodologies to be introduced in network design and operation.

In this paper, we explored an adaptation of test-driven techniques commonly used in software engineering. Because of their primary importance, our methodology is tailored to data path requirements. To realize this methodology, we identified two main missing building blocks, and we presented proposals to fill this gap. First, we defined a formal language which is powerful enough to express the wide variety of data path requirements. Second, we designed, implemented and evaluated a testing framework to check the accommodation of requirements expressed in the proposed language. As such, our framework complements recent efforts for network configuration generation and efficient post-deployment testing.

While we focused on data path requirements, our vision is broader. Indeed, our long-term goal is to deploy methodological and practical support for a complete test-driven methodology that enables network operators to formalize and proactively test all their requirements, including performance and resource optimization ones. We argue that this would improve network design capabilities and significantly help preventing human errors that are the most common source of network mis-behaviors and downtime [26].

Acknowledgement

This work has been supported by the ARC grant 13/18-054 from Communauté française de Belgique.

References

[1] J. D. McCabe, *Network analysis, architecture, and design*. Morgan Kaufmann, 2010.

[2] A. van Lamsweerde, "Requirements engineering: from craft to discipline," in *SIGSOFT*, 2008.

[3] M. Pezze and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[4] M. Chiosi et al., "Network functions virtualisation - introductory white paper," in SDN and OpenFlow World Congress, 2012.

[5] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: a slice abstraction for software-defined networks," in *HotSDN*, 2012.

[6] Z. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," in *SIGCOMM*, 2013.

[7] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," in *SIGCOMM*, 2012.

[8] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Practical declarative network management," in *WREN*, 2009.

[9] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *CoNEXT*, 2012.

[10] X. Sun, S. G. Rao, and G. G. Xie, "Modeling complexity of enterprise routing design," in *CoNEXT*, 2012.

[11] S. Jain et al., "B4: experience with a globally-deployed software defined wan," in *SIGCOMM*, 2013.

[12] P. Hazel, "PCRE - Perl-compatible regular expressions," Available at http://www.pcre.org/pcre.txt.

[13] D. Lebrun, "TASTE - a Toolkit for Automated Sdn TEsting," available at http://inl.info.ucl.ac.be/softwares/taste.

[14] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000.

[15] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *CoNEXT*, 2012.

[16] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.

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

[18] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2001.

[19] A. V. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[20] Y.-W. Sung, X. Sun, S. Rao, G. G. Xie, and D. Maltz, "Towards systematic design of enterprise networks," *IEEE/ACM Trans. Netw.*, vol. 19, no. 3, pp. 695–708, Jun. 2011.

[21] P. Oppenheimer, *Top-Down Network Design*, ser. Networking Technology. Pearson Education, 2010.

[22] A. Sholomon and T. Kunath, *Enterprise Network Testing: Testing Throughout the Network Lifecycle to Maximize Availability and Performance*, 1st ed. Cisco Press, 2011.

[23] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011.

[24] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," in *NSDI*, 2013.

[25] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: declarative fault tolerance for software-defined networks," in *HotSDN*, 2013.

[26] Juniper Networks Inc., "What's Behind Network Downtime?" 2008.

[27] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, "Network configuration in a box: towards end-to-end verification of network reachability and security," in *ICNP*, 2009.

[28] L. Cittadini, M. Rimondini, S. Vissicchio, M. Corea, and G. Di Battista, "From theory to practice: Efficiently checking BGP configurations for guaranteed convergence," *IEEE Trans. Netw. and Serv. Man.*, vol. 8, no. 4, pp. 387–400, 2011.

[29] A. X. Liu and A. R. Khakpour, "Quantifying and verifying reachability for access controlled networks," *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 551–565, 2013.

[30] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *HotSDN*, 2012.

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

[32] M. Kuzniar, M. Canini, and D. Kostic, "OFTEN Testing OpenFlow Networks," *EWSDN*, 2012.