A Behavior-Driven Approach to Intent Specification for Software-Defined Infrastructure Management

Flavio Esposito*, Jiayi Wang*, Chiara Contoli†, Gianluca Davoli†, Walter Cerroni†, Franco Callegati†
*Saint Louis University, USA †University of Bologna, Italy
{flavio.esposito,holly.wang}@slu.edu {chiara.contoli,gianluca.davoli,walter.cerroni,franco.callegati}@unibo.it

***Abstract*—One of the goals of Software-Defined Networking (SDN) is to allow users to specify high-level policies into lower level network rules. Managing a network and decide what policy set is appropriate requires, however, expertise and low level know-how. An emerging SDN paradigm is to allow higher-level network level decisions wishes in the form of "intents". Despite its importance in simplifying network management, intent specification is not yet standardized. In this work, we propose a northbound interface (NBI) for intent declaration, based on Behavior-Driven Development. In our approach, intents are specified in plain English and translated by our system into pre-compiled network policies, that are in turn, converted into low-level rules by the software-defined infrastructure $e.g.$ an SDN controller. We demonstrated our behavior-driven approach with two practical use cases: service function chaining deployed on OpenStack, supported by both ONOS and Ryu controllers, and dynamic firewall programming. We also measured the overhead and response time of our NBI. We believe that our approach is far more general and paves the way for a more expressive and simplified northbound interface for intent-driven networking.**

I. Introduction

The Network Function Virtualization (NFV) paradigm advocates moving middlebox functionality — Network Functions (NFs) — from dedicated hardware devices to software applications that run in virtual environments on top of shared hardware [1]. One of the key paradigms to effectively support the deployment of NFV is Software-Defined Networking (SDN), which enables network programmability by taking advantage of (mostly) vendor-agnostic open application programming interfaces (APIs) [2]. Sharing similar goals with researchers in programming languages, the focus of network programmability has typically been on (i) making (network) programming easier and more accessible, or (ii) to enable safer (network) programming, *e.g.*, via formal methods and verification techniques. Several successful attempts were made to make network programming safer [3]–[5], or easier to debug and test [6]–[8]. Most of these approaches rely on OpenFlow [9], which defines the open southbound API towards network equipment. Despite the successful control-plane decoupling between programming directives and vendor-specific data forwarding mechanisms, OpenFlow — the de-facto standard SDN control plane protocol — is still dependent on low-level technical details for correct data forwarding, such as switch IDs, port numbers, MAC addresses, etc.

With the aim of generalizing network programmability operations, researchers today are seeking new ways to manage Software-Defined Infrastructures (SDIs) — including SDN and NFV environments, cloud computing platforms, and any other form of communication infrastructure controlled by software. The main goal is to define a simple and usable northbound interface (NBI) that provides a high level of abstraction for programming SDI controllers and easily deploying new services. Such NBI could be in many cases helpful to make network programming easier to application developers and network administrators running DevOps, but also to allow personnel with less technical skills, *e.g.* business managers or technical group leaders, to specify a desired set of operations or services by merely knowing some (but not many) low-level aspects.

To inspire the design of such NBI, the Open Networking Foundation has loosely defined the notion of *intent* as a form of network service abstraction at a higher level than typical policy instantiation or composition in SDIs [10]. In literature the words "policies" and "intents" have often been used interchangeably, see for instance [11]. Differently, in this paper by intent we mean a high-level predicate or keyword that can be used to program a mechanism directly, or via instantiation of a policy set. The idea of having an intent-based network interface is then to declaratively allow "what should be achieved," either loosely or tightly, with a high-level description rather than with a detailed specification of "how it should be achieved."

A key challenge that has yet to be addressed, which we plan to tackle in this paper, is how to provide an intuitive intent programming northbound abstraction for SDIs that, even though not being rigorous as a programming language, is simple and expressive enough to deploy service policies and mechanisms on top of any SDI, without requiring neither network operator expertise nor heavy programming experience.

Recent work attempted at defining domain-specific languages for network programmability, raising the level of abstraction [11]–[16]. Although those solutions focus on high-level policies expression, they still require knowledge of different low-level details, *e.g.*, the declarative or functional syntax of the specification language. Such languages or abstractions for northbound interfaces: (i) are not simple to use, (ii) focus on policy specification, not intent, and (iii) still require underlying mechanisms expertise, merely shifting the entry barrier for network programmability without lowering it.

Two more recent solutions share at the high level our same design goals [17], [18]. They both attempt to use human semantics of a text in English to abstract out the low-level

details of a network; these solutions require to either use a Natural Language Processing [17] or to solve a complex optimization problem to interpret a set of intents (despite not calling them intents) [18].

Our Contributions. In this paper, we design and implement a NBI abstraction layer for SDI management where intents can be specified in plain English (or even Mandarin Chinese).[1] Our design principle is based on *Behavior-Driven Development* (BDD) [19], an agile software development technique, corollary of the Test-Driven Development paradigm [20]. A Behavior-Driven Development framework provides the ability of expressing (network level) wishes in a simple way, *i.e.*, using natural language. The expressed wishes, in our case intents, are then translated into (SDI) policies and implemented through relevant network mechanisms by means of appropriate interpreter functionalities. Our focus in this paper is to demonstrate how it is feasible to allow network programmability without having to be familiar with the lower level details of any policy of the underlying SDI.

To this end, we prototyped our approach by applying the behavior-driven, intent-based SDI management to two practical use cases: (i) service function chaining on a NFV environment, deployed on the OpenStack cloud platform,[2] integrating our BDD layer with an interpreter that manages two different SDN controllers — ONOS[3] and Ryu;[4] (ii) dynamic firewall programming with `iptables`, the native Linux Kernel packet filtering software tool. While the former use case refers to the management of a relatively complex SDI, facilitated by the use of off-the-shelf cloud and SDN controllers, we decided to include the latter use case to demonstrate that our approach is not limited to an OpenFlow-based SDN environment to translate high level intents into policies [21].

How are we different? We are different from the aforementioned relevant work [17], [18] for two main reasons. The first is that we not only focus on defining an intent-based NBI taking advantage of natural language features, but we also consider the support of previously compiled underlying network *behaviors*. To interpret (or compile) any general English (or Mandarin) sentence and translate it into an action would be infeasible [22]. Instead, by restricting the scope of our framework to pre-defined feasible behaviors, we enable SDI management to expressed and verified in the form of intents following the Behavior-Driven Development philosophy. The second main difference with existing approaches is that we prove our concept on several application scenarios based on off-the-shelf SDI software tools demonstrating the flexibility of our solution.

The rest of the paper is organized as follows: In Section II we describe our behavior-driven intent NBI design. Then in Section III we describe the workflow of an intent specification, while in Section IV we discuss some implementation details of the considered use cases. In Section V we detail the evaluation of our approach, and we conclude in Section VI.

[1]Our approach can easily support different languages.
[2]http://openstack.org
[3]https://onosproject.org
[4]https://osrg.github.io/ryu

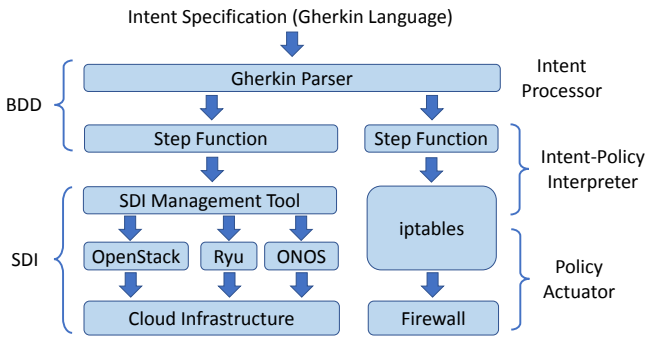

Fig. 1. Behavior-driven intent specification architecture for SDI management. Intents specified in English using the Gherkin language are interpreted by the Gherkin parser and translated by step functions into policies in the form of instructions for the underlying SDI management tools. The picture shows the specialized architecture for the case of a SDI based on NFV and SDN components, as well as the case of a firewall programmed with iptables.

II. Behavior-driven Intent Specification Design

In this section we highlight the main components of our behavior-driven approach and its design principles. Our intent specification layer is composed by (i) a language definition framework based on Gherkin, (ii) an intent-policy interpreter, the core of our northbound interface, and (iii) a set of plugins that act as a middleware between the intent declaration and the underlying SDI management layer. The system architecture, specialized for the use cases presented in this paper, is represented in Figure 1.

Intent definition in plain English with Gherkin. Behavior-driven development (or BDD) is an agile software development technique that was designed to encourage collaboration between developers, Quality Assurance (QA) engineers and non-technical or business participants in a software project [19]. We extended the BDD notion to *behavior-driven SDI management*, by focusing its general notion of declarative requirement specification to network programmability via intent specification. We define our network intents with Gherkin,[5] a language used by non-computer scientists to define requirements in plain English. The following listing shows an example of Gherkin-generated intent that is possible to specify with our framework:

```
Feature: load balancing via NFV
Scenario: modify NF chain after high load
  detection
    Given traffic is flowing on NF1
    And traffic is flowing on switch SW1
    When response time of NF1 is too high
    Then start a new network function NF2
    And redirect half of the traffic to NF2
```

Gherkin was designed so that business (not network) managers would be allowed to express application or service requirements with little but not null technical expertise. For

[5]https://docs.cucumber.io/gherkin

example, a network manager may want to subsequently author such feature with the aim of expressing another policy to perform load balancing when the end-to-end delay becomes higher than a given threshold. The keywords `Given`, `When`, `Then`, `And`, are sufficient to compose complex intent predicates, and the language interpreter can be easily extended.

Having shown a concrete example of how simple our approach can be to specify network intents, we continue this section describing its three design principles: (i) usability, (ii) being verification-agnostic, and (iii) being controller-agnostic.

Usability. Users, managers, applications and network programmers have to have the ability to quickly start, stop or modify basic or complex service behaviors on the SDI, with minimal coding and network management expertise. It is then responsibility of the interpreter to convert those English sentences into a policy set that can be used to program the underlying SDI via northbound API. Note how existing SDN controllers, such as ONOS, already have an intent specification NBI, but programming intents requires low-level expertise.

Verification-agnostic. When we program in a dynamically-typed language such as Python or JavaScript, we do not have any proof that the program will be safe. It is (arguably) clear that writing Python code is easier for newbies. Developers do however, write unit or integration tests to verify correctness of their code. Similarly, our intent management layer does not use any formal verification techniques, but can be used to verify software-defined network behaviors. For example, we could specify intents that ensure that a given packet header is generated after another one is received by a given NF.

Controller-agnostic. Existing intent specification frameworks are controller-specific. Successful abstractions are by definition agnostic from the underlying software-defined infrastructure. In the rest of the paper we show how our approach can adapt to different controllers and to networks that do not support a centralized controller.

III. Intent Interpretation Workflow

In this section, we describe the general workflow of an intent specification when applied to the use case of a service function chain deployment. To apply an intent, the workflow first has to input the specification of the intent in plain (English) text, and then our intent-policy interpreter binds the policy to the underlying SDI management tool, *e.g.*, the Ryu and/or ONOS controllers.

A. General Workflow

Our intents are specified using the Gherkin language, which enables the description of the desired (virtual) network chain or other service behavior. Business and network managers may leverage Gherkin to describe the *feature* they desire. Features can represent higher level goals, as well as lower level protocol policies ($i.e.$, NFV constraints on the software-defined infrastructure). A feature file may or may not start with a title, used as tag to group set of intents, and it is followed by a few lines that provide context and describe the benefits or the feature itself. A Gherkin specification envisages a *scenario* and (possibly) multiple *steps*; the details of the intent specifications start with a Gherkin keyword that has the purpose of providing some context or preconditions, and define what managers should expect as outcome. Outcomes can be tests, protocol messages, logs, network service deployment or termination.

With our approach, not only we are able to express higher-level intents using Gherkin keywords, but also terms that are relevant to specific use cases scenarios. Examples of such terms are those that describe service flows crossing function chain components, or firewall-ing states, *e.g.*, incoming/outgoing/malicious/dangerous traffic, or blacklist. All those terms are seen as keywords for each relevant scenario. Such terms or keywords constitute the information set adopted to achieve infrastructure-independence in the intent specification, as well as to implement mappings among layer-specific terms. Once the intent specification is complete, it is interpreted by our engine, capable of matching previously defined regular expressions, just as in a standard programming language compiler.

When there is a matching, our interpreter translates the intents into policies by calling the appropriate callback function that in turn may call an underlying mechanism. Our approach may be supported by any SDN controller exposing a programmable NBI — although our current implementation[6] is limited to ONOS and Ryu only — as well as by any other form of network programmability.

B. Chain Intent Interpretation Workflow

Let us assume that a user, *e.g.*, a startup CTO, a technical leader or a network manager, wishes to enforce an intent defined by the deployment of a service function chain on their infrastructure. The ordered set of network functions will be implemented on an SDI by means of an OpenFlow-based controller. The controller, responsible for translating the service function chain policy into lower-level infrastructure rules, will then deploy the required service.

With our system, the user, through a user interface such as a keyboard (or a script), expresses such intent with Gherkin syntax, defining a *feature* file. Together with this file, to support the intent specification, a set of rules called *step functions* need to be implemented by the system interpreter. These steps functions are interpreter functionalities and act as callbacks; their signature has a regular expression that needs to be matched against the text subsequent to a defined Gherkin keyword in the feature (intent) file. For instance, in our use case the system interpreter generates from the intent/feature file a JSON policy configuration file that is then sent to the SDI management tool through a REST API call [23]. The management tool runs an application that exposes the service function chaining service as a REST API endpoint. Once such call is received, it is responsibility of the SDI management tool to translate this policy into lower level rules that will be applied to the underlying infrastructure through the ONOS/Ryu controller NBIs. We implemented and tested this example, as presented in the following sections.

[6]https://github.com/flavioesposito/BeA

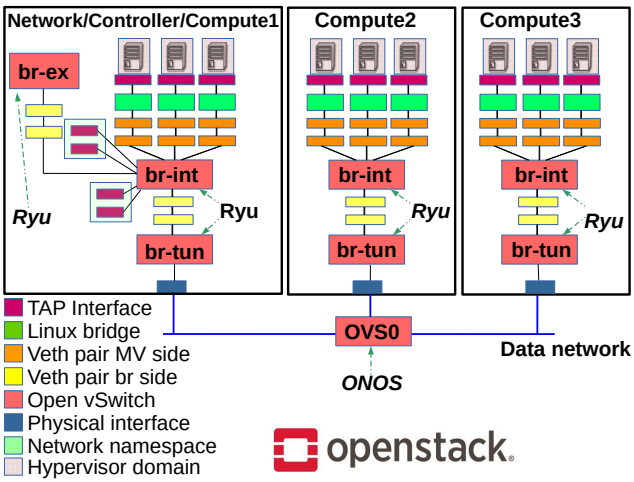

Fig. 2. OpenStack cluster used as a SDI to demonstrate our behavior-driven intent specification for service function chaining. The figure shows the internals of the compute nodes, with virtual bridges natively controlled by Ryu instances, as well as the physical data network interconnection through an OpenFlow-enabled switch controlled by ONOS.

IV. Prototype Implementation and Testbed

We prototyped our intent specification approach leveraging the BDD framework[7] and implementing our own interpreter. We tested our interpreter over an SDI management tool, controlling an OpenStack cloud platform used to execute virtual network functions (VNFs) and two OpenFlow-based SDN controllers to properly steer traffic flows. Moreover, to demonstrate the flexibility of our approach, we also implemented support for intent translation within an infrastructure without controllers. In particular, we demonstrated a use case in which intents are translated to `iptables` (firewall and forwarding) rules. Both the OpenFlow and the non-OpenFlow use cases are integrated with the framework and interact with the underlying SDI via the interpreter [8].

Deploying Service Function Chains on OpenStack. As part of our evaluation, we developed the SDI management tool as a standalone Python plugin that interacts with the cloud operating system (OS). We choose OpenStack, but our code can be ported on any other cloud OS. Our plugin gathers information to monitor the deployed VNFs used to build a service chain. We use multiple SDN controllers to install customized forwarding paths between the VNFs, as dictated by an intent-based service chain specification. The plugin uses the REST interface that the cloud OS and the SDN controllers provide and expose, and the intent-based northbound interface (also REST) to communicate with any logically higher-level component, or to users themselves.

A typical OpenStack cluster includes, among other elements, multiple compute nodes, each of which can host virtual machine or container instances. The OpenStack test bed we used in our experiments is shown in Figure 2 and is composed by a network node co-located with a controller and a compute node, plus two additional compute nodes. Each node includes several virtual elements that form the internal network infrastructure. A set of Open vSwitch (OvS) virtual bridges provide a programmable, distributed virtual networking environment. In recent releases of OpenStack, each compute node also runs an instance of the Ryu SDN controller, used for controlling the internal virtual network components. With a minor intervention on the OpenStack configuration files, it is possible to expose the Ryu REST interface, allowing direct interaction with the controller and facilitating intra-node network programmability in a native way. In our test bed, the connectivity between OpenStack nodes is provided by a flat data network interconnected via an OvS bridge running on a dedicated server and controlled by an ONOS instance. Therefore, in our deployment the interaction with both Ryu and ONOS controllers allows management of intra-node and inter-node traffic steering, respectively.

V. Evaluation Results

In this section, we present some experimental results with the aim of demonstrating the feasibility of our Behavior-driven approach. First, we provide some basic performance analysis of our framework, to demonstrate the NBI overhead: to support from 100 to 1000 intents the processing time increases from less than 1 second to about 7 seconds, respectively (Figure 3). We obtained this result by measuring the intent processor compile time independently from the underlying SDI.

Intent-based firewall programming with iptables. To demonstrate that our approach can be used even in absence of an SDN management platform, we injected firewall intents via our NBI that are interpreted by `iptables` and enforced by the netfilter framework.[9] In particular, our intents have a goal of adding and/or removing IP addresses from built-in whitelist and blacklist to update a firewall dynamically. We use the `iperf` utility to generate and send traffic to all hosts. Figure 4 shows the effect of dynamic firewall programming. As soon as an intent updates any of these lists, packets will start to be delivered (or stopped to be delivered in case of a new blacklisted IP) to the target host. For each of the three scenarios (whitelisted, blacklisted and blacklisted/whitelisted), our test lasted 10 seconds, while each intent-policy translation change occur around time equal to 5 seconds.

OpenStack-based Service Function Chaining. To evaluate this use case, we launched a set of intents that define an ordered list of VNFs composing a given service chain. We first show how the service traffic is successfully steered across the specified VNF chain. Then we measure the response time of the different layers involved in the intent specification and

[7]https://github.com/behave/behave

[8]Our code is available at https://github.com/flavioesposito/BeA

[9]https://www.netfilter.org

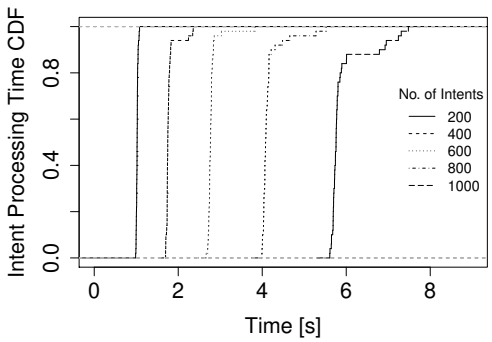

Fig. 3. *Intent Processor* Cumulative Distribution Function of overhead: our system takes on average less than 1s to process 100 intents and about 6s 80% of the time to process 1000 intents, with peaks never reaching 8 seconds.

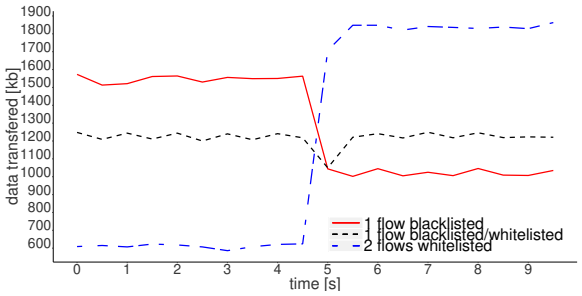

Fig. 4. Effect of dynamic firewall programming via behavior-driven intents. The solid red curve shows a blacklisted flow and the corresponding reduce amount of data transferred. The dotted-dashed curve (blue) shows the case of two whitelisted flows and its related data increase. The dashed black curve shows the effected of a flow being blacklisted and a flow being whitelisted, simultaneously.

interpretation, as well as in the resulting policy enforcement in the underlying SDI.

In our experiment we submit to the Gherkin parser the following intent, which considers the case of changing the service chain between two endpoints when network congestion is detected.

```
Feature: Adaptive Service Function Chaining
  To provide adaptive SFC, deploy service chains
  and monitor the network

@firstdeployment
Scenario: First SFC deployment
        Given  I want to deploy a service chain
  from source NODE-A to destination NOCE-C that
  includes, exactly in this order, a network
  function VF-1, traversed upstream only
        And a network function VF-3, traversed
  both upstream and downstream
        Then deploy service chain

@congestion
Scenario: Congestion
        Given that network monitoring service
  detects congestion, I want to deploy a service
  chain from source NODE-A to destination NOCE-C
  that includes, exactly in this order, a traffic
  shaping network function VF-T, upstream only
        And a network function VF-1, both upstream
   and downstream
        And a network function VF-3, both upstream
   and downstream
        Then update service chain
```

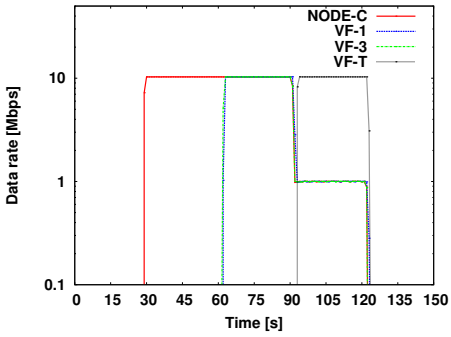

Fig. 5. Input traffic flow bitrate measured on relevant points of a service chain deployed on the OpenStack-based SDI.

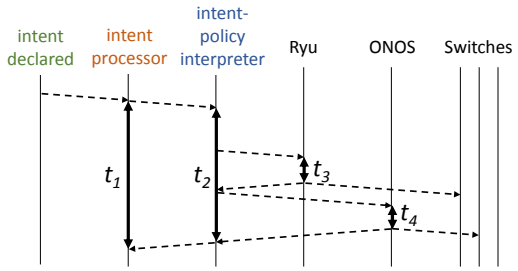

Fig. 6. Workflow of the interactions at different layers of the behavior-driven, intent-based service chain specification.

We start at time $t = 30s$ by generating a 10 Mbits/s `iperf` traffic flow between `NODE-A` and `NODE-C` (hosted on `Compute3` and `Compute2`, respectively). Then we send the *Adaptive Service Function Chaining* intent specification listed above to the parser. As a result, we expect the following chain to be deployed: `NODE-A` / `VF-1` / `VF-3` / `NODE-C`, where `VF-1` and `VF-3` are generic virtual functions hosted on `Compute3` and `Compute2`, respectively. After the intent has been correctly interpreted and the relevant policies have been enforced in the SDI, we verify that the service chain has been actually deployed. This is shown in Figure 5, which reports the input traffic flow bitrate measured at relevant VNF ports. We verify that traffic flow starts crossing `VF-1` and `VF-3` at around $t = 60s$. When network congestion is detected, at around $t = 90s$, the *congestion* scenario is executed (see listing above), resulting in the update of the chain between the source and destination, as follows: `NODE-A` / `VF-T` / `VF-1` / `VF-3` / `NODE-C`. `VF-T` is a traffic shaper hosted on `Compute3`, which is added to the chain in order to limit the bitrate to 1 Mbits/s. The traffic flow finally stops at around $t = 120s$. Chaining is hence properly accomplished for both scenarios described in the intent specification.

Table I shows the response times of the different software layers involved in our system, with reference to the workflow illustrated in Figure 6. The overall time t_1 needed for the

TABLE I
BEHAVIOR-DRIVEN INTENT SPECIFICATION RESPONSE TIME

	Mean [ms]	Conf. Int. 95% [ms]
t_1	2868.999	2632.375 - 3105.622
t_2	2725.459	2488.884 - 2962.033
t_3	4.861	4.724 - 4.998
t_4	4.997	4.543 - 5.451

complete enforcement of an intent in the system is, on average, smaller than 3 seconds, which is a quite reasonable value as a service chain setup time. Most of it, *i.e.* t_2, is spent by the intent-policy interpreter for the computation of the technology-specific instructions to be submitted to the SDI so as to obtain the requested service function chain. Times t_3 and t_4 represent the system calls to the REST interfaces of the SDN controllers used to steer traffic inside and outside OpenStack nodes. For each of them, the interpreter awaits for a positive response, and in turn it returns a positive response to its caller, the *intent processor*.

VI. CONCLUSION

In this paper we presented a northbound interface solution for network intent specification based on Behavior-Driven Development. Our approach allows intent expressiveness in English, Mandarin, or any other natural language, by leveraging the Gherkin programming language. Our prototype includes an intent processor, an intent-policy interpreter, and several Software-defined infrastructure specific policy actuators.

To demonstrate the feasibility, practicality and portability of our approach, we prototyped it over two practical use cases: service function chaining deployed on OpenStack, supported by both ONOS and Ryu controllers, and dynamic firewall programming. We have shown our approach at work with intents that launched dynamic chaining of network functions and dynamic access control rules. We also found that the overhead and response time of our NBI scales reasonably well with up to 1000 intents. We believe that the expressiveness of our behavior-driven northbound intent specification could be in many cases helpful to make network programming easier to application developers and network administrators running DevOps, but also to allow personnel with less technical skills, *e.g.* business managers or technical group leaders, to specify a desired set of operations or services by merely knowing some (but not many) low-level aspects.

ACKNOWLEDGMENT

This work has been funded by NSF Award CNS-1647084, and by Italian PRIN 2015 project "GAUChO - A Green Adaptive Fog Computing and Networking Architecture." We would like to thank also T. Merod for his initial work on this project.

REFERENCES

[1] "Network Functions Virtualisation (NFV); Architectural Framework," European Telecommunications Standards Institute, Standard ETSI GS NFV 002, 2014.

[2] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[3] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. of 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 15–27.

[4] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. of 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 99–112.

[5] S. Ghorbani and B. Godfrey, "Towards correct network virtualization," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 109–114, Aug. 2014.

[6] G. N. Nde and R. Khondoker, "SDN testing and debugging tools: A survey," in *Proc. of 5th International Conference on Informatics, Electronics and Vision (ICIEV)*, 2016, pp. 631–635.

[7] B.Heller et al., "Leveraging SDN layering to systematically troubleshoot networks," in *Proc. of 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013, pp. 37–42.

[8] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker, "An assertion language for debugging SDN applications," in *Proc. of 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014, pp. 91–96.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[10] "Intent NBI - Definition and Principles," Open Networking Foundation (ONF), Tech. Rep. TR-523, October 2016.

[11] Y. Yuan, D. Lin, R. Alur, and B. T. Loo, "Scenario-based programming for SDN policies," in *Proc. of 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2015, pp. 34:1–34:13.

[12] B.L. Loo et al., "Declarative networking," *Commun. ACM*, vol. 52, no. 11, pp. 87–95, Nov. 2009.

[13] R. Soulé et al., "Merlin: A language for provisioning network resources," in *Proc. of 10th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2014, pp. 213–226.

[14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proc. of 16th ACM SIGPLAN International Conference on Functional Programming*, 2011, pp. 279–291.

[15] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. of 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013, pp. 1–13.

[16] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi, "Tierless programming and reasoning for software-defined networks," in *Proc. of 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 519–531.

[17] A. Alsudais and E. Keller, "Hey network, can you understand me?" in *Proc. of 2017 IEEE Conference on Computer Communications Workshops (INFOCOM)*, 2017, pp. 193–198.

[18] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Net2Text: Query-guided summarization of network forwarding behaviors," in *Proc. of 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 609–623.

[19] D. North. (2003) Behavior-driven development. [Online]. Available: https://dannorth.net/introducing-bdd

[20] K. Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[21] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[22] Z. Wang, X. Li, and J. Zhou, "Small-footprint keyword spotting using deep neural network and connectionist temporal classifier," *CoRR*, vol. abs/1709.03665, 2017. [Online]. Available: http://arxiv.org/abs/1709.03665

[23] F. Callegati, W. Cerroni, C. Contoli, and F. Foresta, "Performance of intent-based virtualized network infrastructure management," in *Proc. of 2017 IEEE International Conference on Communications (ICC)*, 2017, pp. 1–6.