Transitioning to Software-Defined Networks

Anubhavnidhi Abhashkumar
Department of Computer Sciences
University of Wisconsin-Madison
anubhavnidhi@cs.wisc.edu

Roney Michael
Department of Computer Sciences
University of Wisconsin-Madison
rmichael@cs.wisc.edu

Abstract

With the ever increasing difficulty in configuring networks, network operators have greater impetus today than ever before to transition to Software Define Networks. This however is not an easy task given the convoluted nature of today's poorly defined network configurations and various constraints such as providing service level guarantees both during transition & after, difficulties in composing SDN applications in conflict-free manners, etc.

Central to these requirements is the need for a well-designed, capable language for programming SDN controllers. Though many options are available today to this end, they showcase significantly diverse designs. In this report we will be making an empirical comparison of two such state-of-the-art languages, Pyretic and FlowLog and provide results which we expect to help further exploration in this enterprise.

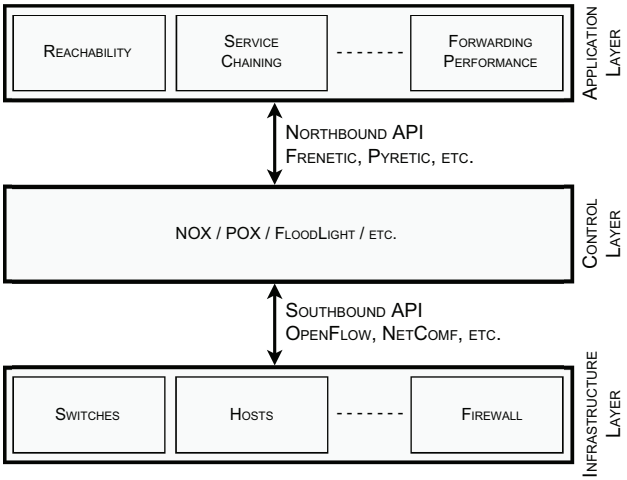

Figure 1: A model of the SDN architecture illustrating the application, control, and infrastructure layers of the network.

1 Introduction

Network configuration is today a highly skilled occupation requiring the operator to have accurate knowledge of a large number of heterogeneous network elements. Such skill is very often hard to come by and human errors may end up costing enterprises considerable time and resources. According to a 2002 report by the Yankee Group [11], these errors accounted for 62% of the network downtime experienced by multi-vendor networks even though on an average nearly 80% of an IT budget would be spent on network management and operation. All of this indicates a present and immediate need for serious rethinking of the way we manage and configure networks today.

Software Defined Networking takes root in this idea of simplifying this management task by constructing logical abstraction over the physical network with the abstracting layers taking care of the fine-tuned metrics which often were the hurdle for human operators. It separates the control aspect of the network from its physical manifestation and provides a platform where network operators may easily and effectively manage computer networks from a central controller. In theory, this would allow for optimal management decisions to be made as the controller would have an all-encompassing view of the topology. The basic design of SDN architecture is illustrated in Figure 1 as described in [19].

As seemingly attractive as SDNs are, there is considerable stasis in their adoption. Current-day networks are vastly different from SDN architectures and making the move to this new technology is a considerable task in itself. Enterprise networks may seem to be the ideal starting point for this deployment but many are loathe to change because of the difficulty in transferring existing network functionality unchanged. ISPs on the other hand are just as unwilling due to the high perceived magnitude

of the required changes. Briefly, the major challenges we have identified are:

- **Programmability without compromising on performance.** SDN switches require the capability to alter their functional behavior based on control information received from the central controller. Achieving this in practice often introduces greater constraints on processing power and delays in packet processing speed which is unacceptable for some high-throughput and/or low-latency applications.

- **A scalable global view of the network.** In order for the controller to make globally optimal decision following the basic idea of SDN, it has to have a real-time complete view of the network topology. Though achievable with only a reasonable level of overhead in smaller networks, latency becomes a considerable roadblock even for mid-size enterprise networks.

- **Incremental transitioning from existing infrastructure.** Though it would be straightforward to deploy SDN infrastructure staring with a clean slate, such an approach would be practically unacceptable due to the magnitude of capital investment that would be required. For real-word use to be viable, there needs to be an incremental bottom-up way to deploy it with SDNs coexisting with legacy infrastructure.

- **Maintaining service level guarantees during the transition.** Network operators should be able to guarantee that nodes are serviced in the same manner and provided with comparable (or better) levels of performance when transitioning to SDN. The problem of rewriting legacy configurations in a manner understandable and modifiable by an SDN controller is non-trivial, especially when the actual intention of some aspects of the configuration are obscured by years of incremental modifications.

In this report we will explore FlowLog and Pyretic, two state-of-the-art languages for programming SDN controllers which attempt to tackle some of these challenges. § 2 will go over the key ideas involved. § 3 will provide a quick overview of the related work and literature in this domain. FlowLog and Pyretic will be briefly described in § 4 and § 5 respectively; we will go over our empirical evaluation in § 6 and the details of implementation of a simple SDN app will be reviewed in § 7. Ideas for future work and pointers to the project's source are provided in § 8 and we will close with our conclusions in § 9.

2 Key Ideas

2.1 Intents

Network programming has traditionally been carried out in an incremental fashion. Computer networks have shown tremendous growth in the recent past and with this comes a great deal of infrastructural changes. This high degree of demand on these networks in the face of requirements for minimal downtime and maintenance of service level guarantees has meant that existing configurations are constantly modified to "get things working" rather than optimally rewriting these to meet the needs. Over the years this has led to the basic ideas behind the configurations being obscured.

The essential idea behind *intents* in networking is that of simplification of network management. For example, when one hires a valet to park their car, they should not have to specify how or where it should be parked; once the keys are handed over, it is the responsibility of the valet to figure out these specifics of carrying out the task.

2.2 Intent-Driven Networking

As networks scale and evolve, the difficulty in maintaining and managing them is getting harder with each passing day. Proper configuration is a difficult task even in highly controlled enterprise network environments and is only exacerbated as many of these subnets converge. True intent-based networking if achieved, would operate on a level opaque to these intricacies and should thus be manageable even by a client from a non-technical background. [2]

The following characteristics of intent-driven networking are what makes it such a potential big-ticket idea:

- **Scalable:** The same intents may be applied to a network with a thousand nodes as to one with a million. The size of the network is inconsequential in defining many intents such as fault tolerance and minimal downtime. The same intent should be applicable to a small private network as to a large enterprise undertaking.

- **Composable:** True intent-driven networking would automatically entail that different intents be easily composable. Say all traffic flowing through a particular node was required to be served by a firewall as well as an Intrusion Prevention System; clients would be able to express these requirements as high-level abstractions and the underlying system should be able to take care of the details of implementation.

- **Constant:** An intent would work at a high-layer abstraction and would not be concerned with the lower layer implementation details. The same intent should be usable across different physical implementation, being agnostic to hardware changes, protocol changes and even unto handling hardware failures. The intent would operate on whatever it has been provided to work on and the network operator should not have to deal with the nitty gritties.
- **Understandable:** OpenFlow rules which dictate how and where each packet is to be forwarded can be overwhelming to analyze. This difficulty in making sense of which rule corresponds to what idea can very easily lead to network misconfiguration and consequently errors or downtime. By specifying high-level intents with the OpenFlow implementation relegated to a trusted controller implementation, conflicts can be found quickly and efficiently.

3 Related Work

OpenDaylight [3] is a collaborative, open source project which is working towards the advancement of Software-Defined Networking. For the purpose of fast adoption of SDN, they have proposed a new project called Network Intent Composition (NIC) [4]. NIC is suppose to use "Intents" representing network behaviors and network policies to control network services and resources. The current SDN interface describes how to provide different services, but Intents can provide the requirements of infrastructure in a descriptive manner. They are trying to use intent as an intermediate language to solve the problem of composition of multiple Intent-Driven network policy into "a consistent set of programmed network actions, monitors, and responses".

One of the important step in transitioning to SDN is to support different vendors. Each of the vendors have their own configuration file and to get Intent from the underlying network we need a vendor independent data structure. To handle we can use Batfish [17] (We use the first two stages of Batfish). The first stage generates the logical model (in logic QL) of control plane from network configuration and topology information which represents the computation done to produce data plane and the second stage generates the data plane by running the logic model on a LogicBlox engine.

Deployment of SDN is one of the primary challenges faced by SDN adoption. Full deployment is hard and may take a long time, to counter that [12] analyzed the benefits of doing incremental deployment by using a design called Panopticon. Its main insight was that in a transitional network(network with both SDN and legacy switch), the benefits of SDN on the network (like enterprise network) could be realized for all the source-to-destination path that included at least one SDN switch. [12] also showed that this could be done without violating reasonable resource constraints.

After getting the intent we then have to deal with the problem of composing Software Defined Networks. Many approaches have been proposed to handle them. [14] talked about the importance of modularity in handling this complex problem. Its main aim was to build a single application out of multiple, independent, reusable network policies acting on the same traffic. It used a programming language called Pyretic [5] that looks at network policy at a high level of abstraction, composes them (sequentially or in parallel) and then executes them on a abstract network topology. More information on Pyretic is given in § 5

One problem which was not considered earlier was the fact that different modules would be competing for the resources or would have conflicting requirements. To overcome this problem [13] proposed a new design called Corybantic in which each module would give information about their local objective i.e. each module proposes how network resources are used and at the same time each module would also evaluate other proposals. After that the Corybantic system uses all these proposal and evaluation to select the configuration that maximizes the system wide objectives while satisfying all policy constraints.

Networks today include middleboxes, directing traffic to middleboxes and ensuring that the sequence in which multiple middleboxes are traversed by a packet are in order requires manual effort. [18] avoids the need of such middlebox policy enforcement and proposes SIMPLE (Software defIned Middlebox PoLicy Enforcement), a SDN based policy enforcement layer which translates logical routing policy to forwarding rules in actual topology. It handles the problems of composition, resource constraints and dynamic traffic/packet transformation without making any changes to middleboxes or SDN API.

[9] uses graphs to express network policies. It proposes a high-level graph based policy abstraction called Policy Graph Abstraction (PGA) in which each subdomain expresses networking policy on endpoints. It then looks at the problem of Policy composition, service chaining as "composing multiple policy graphs". It does it in a proactive, automated manner and also makes use of Veriflow to provide runtime verification(flow rules are compliant with PGA policies).

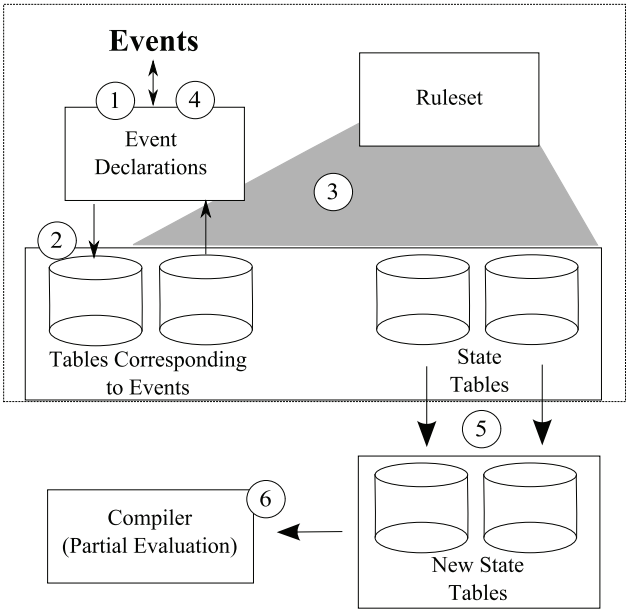

Figure 2: FlowLog system diagram

4 FlowLog

For this project we worked with a programming language called FlowLog [16], a tierless language for programming SDN controllers [15]. SDN has multiple tiers : system flow rules on switches, a controller program, and a data-store for controller state. Expressing such multi-tier application will require multiple languages: databases to manage controller state, remote-procedure call (RPC) for event-handling and policy generation code to create rules on switches. In FlowLog, they all have the same abstraction. This makes it easier to write application in FlowLog and this is the main reason why we chose to use FlowLog to compose different intents. FlowLog is a unified program which describes both control and data plane support.

FlowLog includes SQL like relation state and it allows interaction with external code via either asynchronous events or synchronous remote tables. FlowLog is implemented in Ocaml and it uses OpenFlow & frenetic for packet handling, Prolog engine for evaluation, Thrift RPC for orchestrating events & remote state and Alloy to provide built-in verification.

To give a brief overview of how FlowLog controller works, the FlowLog system diagram is shown in Figure2. The controller's workflow is as follows: (1) when an event arrives, the controller converts it into a tuple, (2) it places this tuple in the input table using XSB's assert command, (3) for each outgoing and state modification table, the controller queries XSB to obtain a set of outgoing tuples, (4) these tuples are converted to events, (5) each state-modification tuple is asserted or retracted to result in the new state, (6) proactive compilation is performed on the new state, producing a NetCore policy.

FlowLog uses both local and remote table. Local table is managed by the controller (via assert and retract statements to XSB), and remote table over callouts to external code. Like events, callouts use Thrift RPC to interact with external code.

The FlowLog program contains a declarative ruleset to govern the controller behavior and a set of declarations for the programs state tables and incoming/outgoing interface. The rules in the ruleset describe how each packet is handled (syntax is reminiscent of SQL and relation semantics). The ruleset contains set of ON blocks of rules. Each rule indicates an action to be taken when the ON-specified trigger is seen. The states in the program will be updated in reaction to packets and other external events. These events can be arbitrary, but their interfaces must be explicitly declared or built in.

FlowLog rules are more powerful than OpenFlow 1.0 flow rules and the proactive ruleset compiler of FlowLog uses the following 3 steps to convert FlowLog to openflow rules: (1) it simplifies each rule and identifies the compilable forwarding rules. The non-forwarding rules (state updates, emission of new packets etc) and non-compilable forwarding rules require switches to send packets to the controller. These notifications can be extensively filtered based on the programs structure. (2) it partially evaluates the rule-set at the current controller state to produce a new ruleset with no references to state tables. The original ruleset defined a function that accepted a state & an incoming tuple and returned a set of outgoing tuples, so the resulting ruleset depends only on the incoming tuple. (3) it compiles the new ruleset automatically to flow table rules in two steps (i) it converts to NetCore (a stateless forwarding policy language for OpenFlow) and then (ii) it applies NetCores compiler to produce flow table rules.

External tables are managed by arbitrary external programs; for e.g. [15] used OCaml programs running on the controller machine.

5 Pyretic

Pyretic [5] is an SDN programming language and is part of the Frenetic family of SDN languages. Pyretic allows network programmers to write/program SDN policy. Pyretic is considered to be both a programmer-friendly domain-specific language embedded in python and the runtime system that implements programs written in the Pyretic language on network switches. The Pyretic policy is a function that takes a packet as input and returns

a set of packets (by matching a packet's metadata/field-name with specific values, it also uses boolean predicates to match multiple header values). This function describes what the network switches should do with incoming packets. Pyretic has built-in support for parallel and sequential composition (which makes it easier to do service chaining). It also has support for dynamic policy which can be updated using callbacks. These are some of the main advantages that Pyretic provides over FlowLog. FlowLog is topology independent and hence does not give us adequate control over network traffic. Pyretic also has the concept of virtual packet using which you can add metadata, tags, add/modify packets etc. In the evaluation§ 6 part of the paper we talk about the reasons behind why we think Pyretic will be a better option for solving the problem of transitioning to SDN compared to FlowLog.

6 Evaluation

We looked into 4 simple intents which we think will be important to a network; reachability, forwarding performance/routing choice, service chaining and end point/policy resolution. For the purpose of evaluation, we tried to answer the following questions: does FlowLog have the capability to satisfy these intents on its own? If not why can FlowLog not handle them and can they be satisfied using Pyretic? We found that reachability and service chaining could be handled by both FlowLog and Pyretic but FlowLog on its own could not handle end point/policy resolution and forwarding performance.

6.1 Reachability (Allow / Deny)

This intent is used to specify whether two end-points should be allowed to communicate with each other or not; the option will be to either allow or deny it. Both Pyretic and FlowLog can satisfy this intent. Pyretic has a variable called policy belonging to the dynamic policy object which can be updated to represent the intent in network and FlowLog can do the same thing using remote/local tables.

6.2 Forwarding Performance

An example of forwarding performance is the routing information used in the network e.g. Shortest Path, Any Path, Min Load Path, etc. FlowLog will not be able to take care of this intent on its own. One of the main reasons behind this is that FlowLog is topology independent. Routing may be taken care of by another program and then FlowLog can invoke it via Thrift RPC making use of remote tables, but there is no particular advantage to doing that. On the other hand, related research [10] has shown that it is possible to support routing in Pyretic.

6.3 Service Chaining

Service chaining is used to specify the order in which a packet must serviced for example, a packet must go through a firewall followed by a load balancer and not the other way around because the firewall has to operate on real packet headers and not on virtual headers (which would have happened if the packet traversed the load balancer first). This can also be taken care by both Pyretic and FlowLog. Pyretic has in-built support for parallel and sequential composition and this can handle the service chaining problem easily. FlowLog on the other hand can solve this problem by making changes to the state table.

6.4 End Point / Policy Resolution (CDN-esque)

FlowLog fails to support this intent by the same argument made for the case of forwarding performance; it is topology independent. It cannot identify closest/best server by itself and hence cannot make the decisions of which CDN server to use on its own. FlowLog can make use of Thrift RPC and remote tables, to handle this but this would defeat the purpose of having a tierless programming language in the first place. Pyretic on the other hand, using virtual headers and dynamic policy, can keep track of information like weight/congestion in links, latency in links, etc. and make appropriate decision to satisfy this intent.

The main reason why FlowLog is not able to handle some of the intents is because it is topology independent and works at too high a level of abstraction which does not allow for fine-grained control; Pyretic appears to be better suited to support the idea of transitioning to SDN. [9] has also demonstrated that Pyretic is well suited to address the problem of composition of SDN apps.

7 Implementation

We present here the implementation and results regarding the intent of reachability.

The reachability app was implemented using both FlowLog and Pyretic, but with some important differences due to the difference in the capabilities provided by the two languages. When launched initially, the switches would naively forward any received packets to the controller. Based on the policies provided to it, the controller would then craft OpenFlow rules and install them on the appropriate switches automatically. Any following

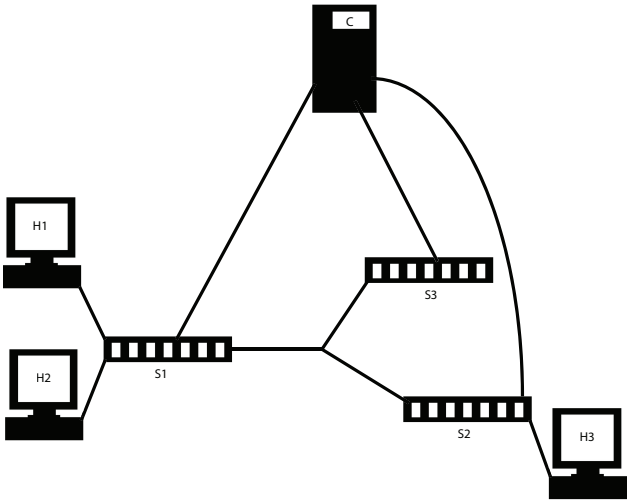

Figure 3: A simple reachability example: H1, H2 & H3 are hosts; S1, S2 & S3 are switches; C is the SDN controller.

packets meeting the criterion of the aforementioned rules would then be forwarded directly at the switch without consulting the controller.

7.1 Reachability

As mentioned before, *reachability* in the SDN context, refers to an intersection of both the control-plane and the data-plane reachability metrics. Data-plane reachability is mandated by access-control lists and the like, whereas control plane reachability is decided by diverse factors such as the state of a particular network node, system downtime, and so on.

Consider the simple SDN example shown in figure 3. H1 & H2 are hosts connected to the network through switch S1 and H3 is connected through switch S2. S1, S2 and S3 are connected to different ports of the controller C, which is tasked with directing flows as per specified network policies by installing OpenFlow rules at the switches. The notion of reachability comes into play when any one of the three hosts attempts to communicate with another. For instance if H1 attempts to send data to H3, the switches on the path between the two hosts, namely S1 & S2, will have to play a part in forwarding the data along on the appropriate path.

We define a *policy group* as a particular set of end-hosts which are to be governed by the same set of high-level rules. A policy group is defined in the networking context by a tuple of acceptable host identifiers. A number of different identifiers maybe used in this determination; for the purposes of our evaluation we used a combination of IP prefixes, port ranges and protocol ranges for group assignment. In regards to reachability these rules are simply the set of other policy groups a host in one group may communicate with. It is to be noted here that different policy groups may not necessarily be mutually exclusive. The OpenFlow rules which operate upon the traffic are prioritized and the rule with the highest priority matching a particular pattern is the one that will be effected.

7.2 FlowLog Implementation

FlowLog does not natively support input to be provided on the fly. This means that policies once put in place may not be modified natively without stopping and restarting the controller. FlowLog however is provisioned to use *remote tables* in addition to local ones. With this in place external programs possibly in other languages or even on remote machines could make RPCs (Remote Procedural Calls) over Thrift [1] to add to and delete from these remote tables. This workaround allowed us to make dynamic changes to the enforced policy. This however expectedly entails a performance trade-off and at the moment it is not entirely clear how this would affect the scale of operation.

Another shortcoming of FlowLogs's current release was its inherent inability to add OpenFlow rules efficiently when ranges of values were involved. In such cases, separate data entries would need to be made for each discrete value in the range; the controller would need to be consulted by the switch each time any packet in the policy group's ranges are encountered, leading to considerable performance degradation. This however is merely a shortcoming of the implementation and not of the design itself and may be expected to be resolved in future releases.

7.3 Pyretic Implementation

Pyretic seemed to offer much wider functionality and much greater flexibility as compared to FlowLog due to the inherent support from Python, a Turing-complete language. This enables input to be read on-the-fly and mathematical computation and comparison to be carried out with ease. We compared our FlowLog implementation of the reachability intent with that of the default *simple_ui_firewall* example [6].

Pyretic was able to support all of FlowLog's capabilities for the reachability app natively and additionally supported efficient dynamic policy assignment. Accounting for ranges in policy groups was also fully supported by design.

8 Future Work and Source

8.1 Future Work

Python's Turing-complete nature is currently the most compelling voice in favor of Pyretic, but it is to be noted that neither one of these languages may yet be in their final form. With changes in design to **take topological information into consideration**, and **support for dynamic reconfiguration of network policies**, FlowLog could offer stiff competition to Pyretic in days to come.

8.2 Source

- LaTeX source of the project report. [7]
- Source code relating to the project report. [8]

9 Conclusion

In this report we have set out an empirical comparison of FlowLog and Pyretic, two of today's state-of-the art languages for programming SDN controllers. The pros and cons of the two languages were compared from both a design as well as an implementation perspective and we have concluded that as it currently stands, Pyretic offers greater flexibility and functionality.

References

[1] Apache thrift. `https://thrift.apache.org/`.

[2] Intent: Dont tell me what to do! (tell me what you want). `https://www.sdxcentral.com/articles/contributed/network-intent-summit-perspective-david-lenrow/2015/02/`.

[3] Opendaylight. `http://www.opendaylight.org`.

[4] Opendaylight: Network intent composition. `https://wiki.opendaylight.org/view/Project_Proposals:Network_Intent_Composition`.

[5] Pyretic. `http://www.frenetic-lang.org/pyretic/`.

[6] pyretic/simple_ui_firewall.py. `https://github.com/frenetic-lang/pyretic/blob/master/pyretic/examples/simple_ui_firewall.py`.

[7] Transitioning to sdn: Latex source. `http://pages.cs.wisc.edu/~rmichael/CS740_Spring15/transitioning-sdn.tar.gz`.

[8] Transitioning to sdn: Source code. `http://pages.cs.wisc.edu/~rmichael/CS740_Spring15/transitioning-sdn-code.tar.gz`.

[9] CHAITHAN PRAKASH, JEONGKEUN LEE, YOSHIO TURNER, JOON-MYUNG KANG, ADITYA AKELLA, SUJATA BANERJEE, CHARLES CLARK, YADI MA, PUNEET SHARMA, YING ZHANG. Pga: Using graphs to express and automatically reconcile network policies. ACM.

[10] JIANG, J.-R., HUANG, H.-W., LIAO, J.-H., AND CHEN, S.-Y. Extending dijkstra's shortest path algorithm for software defined networking. In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific* (2014), IEEE, pp. 1–4.

[11] KERRAVALA, Z. Configuration management delivers business resiliency. *The Yankee Group* (2002).

[12] LEVIN, D., CANINI, M., SCHMID, S., SCHAFFERT, F., FELDMANN, A., ET AL. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *USENIX ATC* (2014).

[13] MOGUL, J. C., AUYOUNG, A., BANERJEE, S., POPA, L., LEE, J., MUDIGONDA, J., SHARMA, P., AND TURNER, Y. Corybantic: Towards the modular composition of sdn control programs. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013), ACM, p. 1.

[14] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., WALKER, D., ET AL. Composing software defined networks. In *NSDI* (2013), pp. 1–13.

[15] NELSON, T., FERGUSON, A. D., SCHEER, M., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. *NSDI, Apr* (2014).

[16] NELSON, T., GUHA, A., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. A balance of power: Expressive, analyzable controller programming. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM, pp. 79–84.

[17] PEDROSA, A. F. S. F. L., WALRAED-SULLIVAN, M., AND MILLSTEIN, R. G. R. M. T. A general approach to network configuration analysis.

[18] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 27–38.

[19] SEZER, S., SCOTT-HAYWARD, S., CHOUHAN, P.-K., FRASER, B., LAKE, D., FINNEGAN, J., VILJOEN, N., MILLER, M., AND RAO, N. Are we ready for sdn? implementation challenges for software-defined networks. *Communications Magazine, IEEE 51*, 7 (2013), 36–43.