Proof-based Verification of Software Defined Networks*

Chen Chen[◇] Limin Jia[†] Wenchao Zhou[‡] Boon Thau Loo[◇]
[◇] *University of Pennsylvania* [†] *Carnegie Mellon University* [‡] *Georgetown University*

1 Introduction

Software defined network (SDN) eases the task of programming and managing computer networks. The conceptually centralized nature of the control plane provides a holistic view of the network, thereby making it feasible to verify SDN's functionalities. Verification of SDN is gaining attention in the last few years [1, 5]. There are two main challenges of SDN: (1) SDNs are often programmed in general-purpose programming languages (e.g. Java, Python), which makes it tedious and error-prone to apply formal methods over controller applications; (2) the sheer scale of modern networks makes state explosion problem an insurmountable challenge for model checking. Model checking techniques combined with limiting the expressiveness of the programming language have demonstrated as an effective approach to verifying basic properties. However, due to the highly dynamic nature of SDN, verification of more complex security properties is still challenging.

To address the above challenges, we propose a unified framework for programming and verification of SDNs. Our framework relies on the use of a declarative language, *Network Datalog* (NDLog) [4], which provides compact encoding of SDN functionalities and serves as a basis for formal analysis. As a preliminary step, we demonstrate that NDLog can encode basic openflow applications succinctly, and preserve well-formed logical structure. Based on the semantics of NDLog, we develop a sound program logic for verifying invariant properties of NDLog program. The approach of static analysis avoids the state explosion problem. Also, properties of the system can be verified in a compositional manner by dividing them into smaller invariants of different components. Compared to existing proposals such as Frenetic [3], NDLog has a tighter connection to first-order logic and therefore makes the verification tasks easier.

2 Declarative Programming

A NDLog program consists of several rules of the form $h :- b_1, ..., b_n$, where h and b_i are tuples representing controller (or switch) state. Each tuple is associated with a location specifier (indicated by an "@" sign) that indicates the location of the tuple. Informally, the rule head (h) is derived when all tuples in the rule body (b_i) hold. The distributed evaluation of NDLog programs computes all derivable tuples from given base tuples. During this process, when necessary, tuples are sent over (retrieved from) the network.

Network Datalog. To illustrate NDLog's syntax and how it can be used to model SDN controllers, we present an example controller program for Ethernet MAC learning. (For ease of exposition, the program is based on a simplified openflow specification. But it is not difficult to modify it to capture the exact packet format).

rc1 $\mathsf{flowMod}(@swc, smac, iport) :- \mathsf{ofConn}(@ctl, swc), \mathsf{ofPacket}(@ctl, swc, iport, smac, dmac).$
rc2 $\mathsf{broadcast}(@swc, smac, dmac) :- \mathsf{ofConn}(@ctl, swc), \mathsf{ofPacket}(@ctl, swc, iport, smac, dmac).$

The program models the scenario where an Ethernet packet with source MAC address $smac$ is received at switch swc from the $iport$ port, and cannot match any rule on the switch. The packet is then forwarded to the controller (i.e., the ofPacket tuple), and the controller, on receiving ofPacket, performs two actions: 1) it instructs the switch to install a rule that forwards all packets destined to $smac$ through port $iport$ (rule *rc1*), and 2) it instructs the switch to broadcast the unmatched Ethernet packet (rule *rc2*).

Openflow Switch. To capture a uniform view of a complete SDN system for ease of verification, we also model openflow switches—both its software (openflow interface) and hardware (packet forwarding) functionalities using NDLog. Due to space limitation, we omit the detailed NDLog encoding. It is worth noting that, unlike the controller, the encoding of openflow switch is fixed given a specific openflow version, thus can be included as an embedded component in our proposed framework, transparent to the users.

3 Verification

We intend to use an extension of a program logic for proving invariant properties of NDLog programs [2] to verify SDN configurations.

Network Abstraction. We model a network as a set of connected nodes, each of which runs an NDLog program. The behavior of the network is then modeled as execution traces, each of which is a sequence of actions, generated

by controllers and switches in the network. A set of transition rules based on NDLog's operational semantics dictates how these traces are generated. Such a network abstraction goes beyond modeling flows of packets; it also captures events occurring at switches and controllers, which are key to verifying complex properties.

Properties. The properties of interest include the usual reachability type of properties (e.g., can packets matching specific descriptions flow from A to B?) and stateful properties (e.g., can packets matching specific descriptions flow from A to B after a specific event?). More concretely, we are interested in properties expressible as linear temporal logic formulas (LTL). Many correctness and security properties of the network can be specified in LTL. Below is an example property of our example program:
Reactive flow installation: the controller adds a flow entry to a switch only after the controller receives a corresponding openflow packet from that switch.

Program Logic. We use first-order logic as the specification language for properties. Atomic predicates represent actions, as well as derivable NDLog tuples. For instance, $\mathsf{ofConn}(ctl, swc, ctl, t)$ means that a tuple $\mathsf{ofConn}(@ctl, swc)$ is derivable at the controller ctl at time t.

Our prior work has developed a Hoare-style program logic to reason about the properties of NDLog programs [2]. We can prove invariant properties (properties that are true throughout the execution of a program) using a combination of manual annotation, automated lemma generation and interactive theorem proving. The main idea is that the global property of the entire network is manually decomposed into local properties of each individual node and the local properties are verified against the NDLog program on that node. The verification steps include (1) specifying the global and local properties, (2) generating supporting lemmas for proving that local properties are satisfied given the NDLog program, (3) proving these lemmas using a theorem prover and (4) proving that the composition of local properties implies the global property. Our initial results show that this methodology can be extended to apply to SDN verification.

4 Challenges

At a high level, the switches and controllers communicate to each other via asynchronous messages. The number of possible permutations of these messages can be huge, which is a main cause of problems in model-checking based techniques. This is also a problem for our proof-based verification technique, as our model over-approximates the states to ensure soundness. Naïve application of our technique is likely to leave many properties unprovable, even when they are true. We are interested in refining the model based on failed proofs. Failed proofs can also be used to identify potential problems. For example, a failed attempt in proving "non-redundant flow installation" of our example program reveals that two identical openflow packets sent to the controller before the controller makes any response would cause the same flow entry to be installed twice on the same switch.

Automation is key to the adoption of our techniques. We will investigate (1) automated local property specification and (2) automated first-order logic proofs. Our prior work relies on manual annotations and an interactive theorem prover. From our experience, only a small fragment of the proofs requires manual efforts (e.g., how to carry out inductive proofs). Sub-goals of these proofs tend to be plain first-order logic formulas, and therefore, it is feasible to maximize automated portions of the verification.

References

[1] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[2] C. Chen, L. Jia, H. Xu, C. Luo, W. Zhou, and B. T. Loo. A formal framework for secure routing protocols. Technical Report MS-CIS-13-03, CIS Dept. University of Pennsylvania, April 2013.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *ICFP*, pages 279–291, 2011.

[4] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.

[5] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. A balance of power: expressive, analyzable controller programming. In *HotSDN*, pages 79–84. ACM, 2013.

*This work is funded by NSF CNS-1218066, NSF CNS-1117052, NSF CNS-0845552, NSF ITR-1138996, NSF CNS-1115706, and AFOSR Young Investigator award FA9550-12-1-0327.