Formal Foundations For Software Defined Networks

Arjun Guha Mark Reitblatt Nate Foster

Department of Computer Science
Cornell University

Introduction. Software-defined networking (SDN) makes it possible to control an entire network in software, by writing programs that tailor network behavior to suit specific applications and environments. Unfortunately, developing *correct* SDN programs is easier said than done. SDN programmers today must deal with several complications:

- *Two-tiered architecture:* An SDN "program" has two distinct components: the controller program itself and the packet-processing rules installed on switches. These pieces have intricate dependencies that make reasoning difficult—*e.g.*, installing or removing a rule can prevent the controller from receiving future network events. Hence, a programmer must reason about the behavior of the controller program, the rules on switches, and the interactions between the two via asynchronous messages.
- *Low-level operations:* SDN platforms such as OpenFlow force programmers to use a low-level API to express high-level intentions, which makes reasoning about SDN unnecessarily hard. Recent revisions of OpenFlow expose even more hardware details, such as multiple typed tables, port groups, and vendor-specific features, which makes the problem worse.
- *Event reordering:* Hardware switches employ a number of techniques to maximize performance, including reordering controller messages. This makes the semantics of SDN programs highly non-deterministic, further complicating reasoning. For example, in the absence of barriers, a switch may process messages from the controller in any order.

Together, these complications make it difficult to reason rigorously about SDN programs. Even establishing basic reachability properties—*e.g.*, the program provides connectivity, correctly enforces access control policies, or is free of loops—involves intricate reasoning that exceeds the capabilities of most programmers.

Our goal is to provide a mathematical foundation for software-defined networking that can be used to build and verify high-level SDN tools. A programmer who uses these tools will be assured that certain specified formal guarantees will not be violated. To this end, we have developed a low-level model of SDN, called *Featherweight OpenFlow*. This model is based on the informal OpenFlow specification, but has a precise mathematical definition that makes it suitable for formal reasoning. We have implemented Featherweight OpenFlow in the Coq theorem prover as an executable artifact that can be used to build practical, high-level tools. Using this executable model, we have built a verified controller for the NetCore policy language [3] that addresses several serious bugs that were present in our original unverified controller. We have also built an automatic property-checking tool for NetCore policies, and have developed libraries of theorems that capture key correctness properties involving controllers and flow tables. We expect these libraries will be useful in other developments.

This talk will present an overview of our approach, our main results, and our verified controller. A forthcoming paper in the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) describes our system in further detail [2].

Vision. In other fields of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to build reliable systems. For example, in processor development, automated theorem proving routinely uncovers deep bugs in designs before they become costly errors in silicon; avionics developers use program analysis to verify critical safety properties of the embedded software running on airplanes; and operating system vendors have successfully used model checking to eliminate whole classes of bugs in device drivers. Yet, until recently, networks have largely resisted analysis using formal techniques.

Our vision is a mathematical foundation for SDNs that enables and facilitates formal network reasoning. Recent advances in formal methods have made it possible to precisely model systems of realistic size. In particular, operational semantics have been used to model the behavior of complex systems such as the C programming language, x86 processors, and even whole operating systems. We seek to develop detailed models of SDNs that support reasoning about essential network functionality such as forwarding, as well as complex features such as bandwidth, queues, controller resources, and failures. With these models, researchers can communicate their ideas concisely and unambiguously; developers of SDN controller platforms and tools can verify that their features are implemented correctly; and users and network operators can be assured that critical safety properties are correctly enforced.

Contributions. *Featherweight OpenFlow* models key features of OpenFlow networks, including the semantics of switches, flow tables, and links, and all essential sources of asynchrony (control message and packet reordering). The model is implemented in Coq and is thus an executable artifact. Using Featherweight OpenFlow as a foundation, we have built a verified software stack (depicted in the figure). The stack implements a compiler from NetCore to flow tables, a flow table optimizer, and a controller. All these components are verified to be correct: the only unverified component is the low-level serialization of OpenFlow messages (which could also be verified if desired).

We have established two fundamental correctness results for our system:

- *Simple Controller Correctness Principles:* Proving from scratch that a given controller correctly implements a given packet-processing function is a formidable task. Doing so requires reasoning about intricate details such as asynchrony in the network and the possibility of message reordering. We have developed a generic reasoning technique that dramatically simplifies the proof task. To verify a controller, it is only necessary to prove two natural properties: (i) the controller program must implement the packet-processing function, and (ii) each switch must approximate the packet-processing function and otherwise send packets to the controller. For most controllers, proving these properties is straightforward.

  This result encapsulates a large amount of intricate reasoning about OpenFlow programs and packages it up into a generic controller-correctness theorem. This is a powerful result: to establish correctness for a new controller, we do not have to start from scratch; we only have to prove two simple properties. Thus, controllers that use our technique can safely provide high-level abstractions to SDN applications.

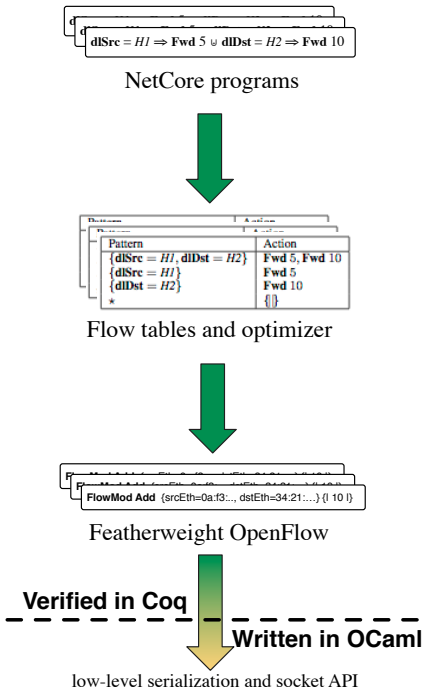

Figure 1: Verified controller stack.

- *Compiler Correctness and Flow Table Manipulation Library:* Using our theorem , we rapidly developed (and formally verified in Coq!) several different controllers that use various schemes for managing the rules installed on switches. In particular, we built a compiler from a high-level SDN policy language, NetCore [3], to flow tables. NetCore policies have a simple, high-level semantics that is also more expressive than flow-tables, since NetCore supports many different ways to compose policies. We verified that the compiler and run-time system are semantics-preserving: the compiler translates NetCore policies into equivalent flow-tables, and the run-time system correctly installs these rules on switches. In the course of developing our verified implementation, we uncovered several bugs in our previous work on NetCore as well as other SDN policy languages.

  The key tool we use is a library of flow table transformation functions and associated theorems, none of which are specific to NetCore. This library precisely states how flow tables match packets (including subtle dependencies in patterns); it includes operations to compose and transform flow tables in several ways; and it proves several useful properties about these operations. Using these theorems, we built and verified a generic flow table optimizer that is part of our stack.

Future work. We hope that our SDN model will serve as a useful foundation for building other tools. For example, the model could be used as a test-oracle for OpenFlow switches, or as an engine for an OpenFlow software model-checker, in the style of NICE [1]. The model could also be used to develop property-checking tools for high-level abstractions. We have built such a tool for NetCore based an encoding in first-order logic extended with fixed points.

The model can also serve as a foundation for future extensions to OpenFlow and complementary APIs. We designed Featherweight OpenFlow to be extensible by modeling individual devices as independent processes. The first version of our model elides communication errors and failures, but it would be easy to account for them by adding rules that drop or modify packets non-deterministically. It is also easy to add new kinds of devices such as switches with additional capabilities (e.g., queues).

Acknowledgments. This work was supported in part by the NSF under grant CNS-1111698 and TRUST, the ONR under award N00014-12-1-0757, and by a Google Research Award.

References

[1] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. Automating the testing of OpenFlow applications. In *NSDI*, 2012.

[2] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In *PLDI*, 2013.

[3] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.