A (Not) NICE Way to Verify the OpenFlow Switch Specification

Formal Modelling of the OpenFlow Switch Using Alloy

Natali Ruchansky, Davide Proserpio
Boston University
{natalir, dproserp}@cs.bu.edu

Categories and Subject Descriptors: C.2.2[**Network Protocols Subjects**]: Protocol verification
Keywords: OpenFlow Switch; Alloy.

1. INTRODUCTION

The introduction of Software Defined Networks (SDNs) is completely changing the way in which networks are built and managed. SDNs decouple data from control plane access, which makes introduction of new network functionalities significantly simpler. The philosophy of OpenFlow is a move towards centralization, where a single controller program manages the logic of switches. While centralized systems are often easier to coordinate, the likelihood of bugs is still high. Despite the existence of an OpenFlow Specification [3], it may still be possible observe unexpected behavior while adhering to this Specification. This can be due to various reasons, such as underspecification of some aspect of the protocol or a contrived sequence of events.

One of the emerging techniques to verify (prove that a system satisfies its specification) standards and protocols is formal modeling. Created at some chosen level of abstraction, the purpose of a formal model is to enable precise understanding, specification, and analysis of the system. The modeling language Alloy has been noted as a tool that lends itself to modeling complex networks. In fact, it has been used in many applications [1], including the analysis of Chord [6, 7] which led to a counterexample proving the incorrectness of the protocol.

The main contribution of this paper is to apply the principles of formal modeling to OpenFlow. Concretely we use model enumeration (Alloy and Alloy Analyzer [5]) to model an OpenFlow-capable switch. The aim of this project is twofold: (1) to provide a proof of correctness (or not) of the OpenFlow Switch Specification Version 1.1.0 and (2) provide researchers with a complete OpenFlow Switch module that can be used as a foundation to verify various applications or types of networks (more detail in Section 4 and our site [2]).

The remainder of the paper is organized as follows. In Section 2 we briefly introduce Alloy. In Section 3 we describe exactly what we model, and the assumptions and abstractions that go along with it. Then, we present some of the properties we defined. Finally, in Section 4 we summarize our results and future work.

SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
ACM 978-1-4503-2056-6/13/08.

2. ALLOY

We start by drawing a distinction between Alloy models and implemented systems. An Alloy model is not a working system that can be deployed as software in a real-world Switch, rather it is a proof of the logic behind the Switch's function. It is a model enumeration tool that is similar to model checking (as used in [4]), where the language is a mix of first-order predicate logic and relational algebra. A model specified using Alloy can be thought of as consisting of statements that take a boolean value (true or false) and together form a proof. This model is paired with the Alloy Analyzer [5] to verify consistency and desired properties.

To better understand Alloy, it can be helpful to imagine observing a working network with packets wizing around and all sorts of actions being taken; after some time the *pause* button is pressed and a snapshot of the system is taken. The (oversimplified) role of Alloy Analyzer is to evaluate possible next-steps from this snapshot. In a sense, the tool assigns a truth value to every possible event and finds logical inconsistencies using a SAT solver.

Consider expressing the rule of *No self-loops*. In Alloy, this rule can easily be enforced for all nodes with the code:

```
fact{no n:Node,p:n.ports | p.connection in n.ports}
```

The line of code requires that at any point in the model, there is no node that is connected to its own port. While Alloy allows for simple expression of complex requirements, it does have drawbacks. Since it relies on a user-specified model size, the Alloy Analyzer is sound, but not complete. In other words, if no sample model is found then nothing can be inferred. For example, checking a model for three nodes does not guarantee that it will be correct for four. On the other hand, when Alloy does not ask for the size, it generates all possible models and the space explodes in size. However, this problem is inherent in all state-of-the-art model checkers and enumerators.

3. MODEL

It has been conjectured that the OpenFlow Specification is underspecified, and that implementation as-is could result in undesired behavior. For example, it may be that certain

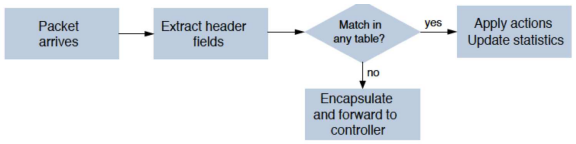

Figure 1: Packet travel through an OpenFlow Switch.

message encapsulation leads to an unexpected forwarding loop because the packet is not recognized. These situations are often caused by a very specific (perhaps rare) chain of events, which makes them hard to fabricate and more readily found by model-checking tools. From the construction and verification we have done so far, no inconsistencies have been encountered. Nevertheless, many of the underspecification conjectures focus on group tables and broadcast functionality, that are not yet included in our model. In this section we describe what our model currently contains.

3.1 Model Overview and Assumptions

The goal of our model is to explore the various configurations and behaviors of an OpenFlow Switch. We model the internals of a `Switch`, covering all parts of the Specification – managing tables, entry-matching (Figure 1), messaging, and more. Outside of the `Switch`, we also define a `Network` and its component `Node`s (`Controller`s, `Switch`es, and `Endhost`s), but only the functionalities needed for their interaction with the `Switch`. One advantage of doing so is the possibility for a user to give a specific network topology as input (discussed further in Section 4).

Though our network is generic and without specific structure, we incorporate some intuitive properties. For example, we require that the *network is connected* – every `Node` is connect to at least one `Node` and no `Node` is connected to itself.

An important part of Openflow is the secure and reliable channel between the `Switch` and `Controller`. Given the nature of our model we assume that this connection has already been established and exhibits all desired properties. We also note that our model focuses on *'OpenFlow-only'* Switches (for more detail reference [3]).

3.2 Abstractions

Our focus is solely on the behavior of an OpenFlow Switch. Therefore there are various aspects of the Specification that are abstracted out – these include counters, field-wise entry matching, GoTo functionality, and fully formed messages. We do not care about whether the counter value is 2 or 3, or the IP address is 192.168.0.1 or 192.168.19.22 – such things are important in a real-world system, but not in a formal model. What matters to us are abstract logical expressions. Does the IP match? Is the counter updated? Regardless of the precise value. The GoTo is unnecessary because assuming everything can be replicated in the next table is sufficient; however, future models may make this explicit.

3.3 Properties

With our model assumptions and abstractions in hand, we now present some properties that are inherent in our model.

NoForwardingLoops. This is ensured by checking that a packet entering a switch has not previously entered the switch. In Alloy this is written as:

```
pred noForwardingLoop [s:Switch, p:Packet]
   {no port:s.ports | port in (p.seen)}
```

The statement above (called a *predicate*) evaluates for a given `Switch` and `Packet` whether there is a forwarding loop. It returns *true* or *false* and can be used with *if...else...* statements to create larger constraints.

NoBlackHoles. No packet disappears from the system. This property is implicitly defined in our model since if the `Switch` receives a packet, the appropriate actions are taken. To understand this better, recall the example in Section 2 with the packets wizing around, the *pause* button, and the snapshot. Now lets look at this snapshot. If the snapshot depicts a `Packet` inside the `Switch`, then necessarily there is a next state where the `Switch` will process the `Packet`. It cannot be that this `Packet` will never be acted on.

CorrectInstall Upon receiving a new flow rule, installation respects the rules for existing entries. This property is enforced with statements that require the `Switch` to check for overlapping and identical entries, as described in [3].

We have also implemented a few other properties such as *FIFOprocessing*, *EchoAwareness*, *InstantOFResponse*, and *NoForgottenPackets*. More detail in [2].

4. CONCLUSION

In this work we begin to formally model the OpenFlow Switch using Alloy. We started by following the OpenFlow Specification 1.1.0 and modelling the properties stated. Although our model is in the early stages, it is already able to cover a great part of the properties of an OpenFlow Switch. We are currently working on the introduction of the concept of time using a specific *Ordering* module. This will make our model richer and more flexible, by allowing us to express properties that are history-dependent. For example, we can very nicely re-write the *NoForwardingLoop* property:

```
fact NoForwardingLoops
   {all t:Tick, m:Msg | no (t.visible.m & t.read.m)}
```

The property must hold for any node at any point in time, and states that there is no message that has both been read in the past and is available to read.

The final goal is to have a complete model of the OpenFlow Switch. Ideally this model would provide a useful tool for researchers to verify arbitrary protocols or application. Our model could be imported as a black box into a more complex model, without worrying about the component correctness. Further, the fact that a specific network topology can be passed as an input will allow researchers to verify particular network structures or applications.

5. REFERENCES

[1] http://alloy.mit.edu/alloy/citations/case-studies.html.
[2] http://cs-people.bu.edu/natalir/ofswitch/.
[3] Openflow switch specification version 1.1.0, 2011.
[4] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A nice way to test openflow applications. *NSDI, Apr*, 2012.
[5] D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.
[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peertopeer lookup service for internet applications. *ACM SIGCOMM 2001*, 2001.
[7] P. Zave. Lightweight verification of network protocols: The case of chord. *Unpublished, http://www2. research. att. com/˜ pamela/chord. pdf*, 158, 2009.