Model Based Black-Box Testing of SDN Applications

Jiangyuan Yao*†, Zhiliang Wang¶†, Xia Yin*†, Xingang Shi¶†, Jianping Wu*† and Yahui Li§
*Department of Computer Science and Technology, Tsinghua University
¶Institute for Network Sciences and Cyberspace, Tsinghua University
†Tsinghua National Laboratory for Information Science and Technology (TNLIST)
§College of Software, Jilin University
{yaojy,wzl,yxia,jianping,liyahui@csnet1.cs.tsinghua.edu.cn}

ABSTRACT

The quality of control plane applications determines reliability of the Software-Defined Networking (SDN). The risk of bugs and challenges for testing actually have been increased due to the programmability of SDN. In this paper, we propose a model based black-box testing method for SDN applications. We use a group of components to describe the data structure stored in the applications and the system behaviors. Based on our models, we present our test generation approach and work-in-progress test tool. With partial model composition, it can systematically generate data plane test sequences and alleviate state explosion.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol verification; D.2.5 [**Testing and Debugging**]: Testing tools

Keywords

Software-Defined Networking (SDN); Model-Based Testing (MBT); SDN application testing; Test generation;

1. INTRODUCTION

The programmability of Software-Defined Networking (SDN)[6] ease the difficulty of new network functions' development and deployment. Unfortunately, it also brings new challenges for testing. Some researchers have worked on testing of SDN applications. NICE[2] employed model checking for testing OpenFlow applications. It uses the assignments of the variables in the source codes as states of the models. It is hard to discover implementation faults by white-box method. Furthermore, the source code based models may become time-consuming or even lead to space explosion due to details. Similarly, Vericon[1] and FlowLog[4] extract model from source codes for verification. They are both white-box methods. STS[7] proposed a black-box fuzz tester for SDN applications. The network events were generated based on probability. Without formal models, it can avoid state explosion problem. As the price, STS cannot ensure the test coverage for the specific applications.

In this paper we propose a model based black-box testing method for SDN applications. First, We use a group of parallel component models to describe the system behaviors and data structure stored in the applications. An application can be specified with several component models and the data can be passed through these components. Then, we propose a test generation approach which can systematically generate data plane test sequences and alleviate state explosion. We calculate model composition of a few related components. Then we can generate test sequences from the composition instead of exploring the global state space of the whole system. Finally, we plan to implement our test tool based on our model and test generation. This on-going tool uses the generated data plane sequences for SDN network simulation traffic. It can expose both the design flaws and the implementation bugs.

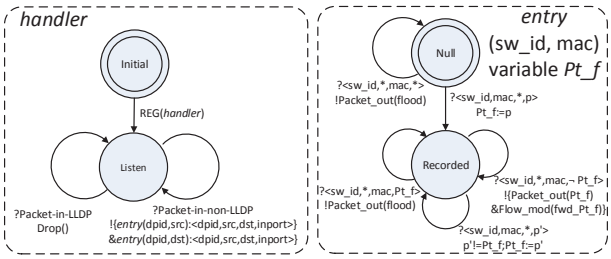

Figure 1: Component Models of MAC-learning

2. FORMAL MODEL

According to SDN architecture, the applications need to get global view of the networks. To achieve this target, the applications usually need to store some information which may be collected by both positive or passive ways.

In our model, the *handler* component extracts information from the Packet-In messages and then saves it into *data table*. We use a group of parallel *entries* components to specify the *data table*. They deal with specific data passing from *handler* and output Packet-Out Messages.

For example, Fig.1 shows the component models of MAC-learning[5] application. This application keeps a MAC table for forwarding. Firstly, The *handler* filters all LLDP messages. Then, it extracts data $< dpid, src, dst, inport >$ from other Packet-in messages and sends it to corresponding *entries* of the MAC table for processing. The *entry* is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT Student Workshop'14, December 2, 2014, Sydney, Australia.
Copyright 2014 ACM 978-1-4503-3282-8/14/12 ...$15.00.
http://dx.doi.org/10.1145/2680821.2680828.

identified by *(sw_id,mac)*. The *handler* sends the data to $entry(dpid, src)$ for MAC-learning and $entry(dpid, dst)$ for forwarding. The *Null* state of *entry* means no this MAC in the MAC table. The variable Pt_f stores the forwarding port. When this MAC and ingress port are learned, the *entry* moves into *recorded* state. Different outputs will be generated according to different states of $entry(dpid, dst)$ and value of *inport*. Moreover, the ingress port can be updated in *recorded* state.

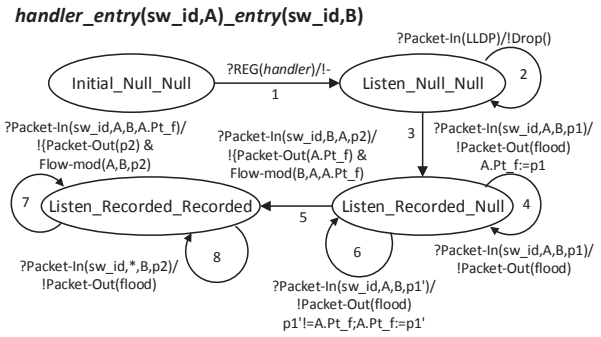

Figure 2: Partial Composition of MAC-Learning

3. TEST GENERATION

There may be a lot of data stored in the SDN applications. As a result, if we compose all components together, its global state space will go explosion inevitably. However, we notice that each *entry* component behaves in the same way. We can systematically cover the application's behaviors with partial model composition. For illustration, as shown in Fig.2, we make the partial composition of MAC-Learning from its three components, *handler*, *entry* (sw_id,A) and *entry* (sw_id,B). A and B are abstractions of MAC addresses of the data plane packets which trigger the Packet-In messages. Because of the similarities between *entry* (sw_id,A) and *entry* (sw_id,B), we can only keep the transitions of *entry* (sw_id,A) which are *learning* and the ones of *entry* (sw_id,B) which are *forwarding*. As a result, the composition can be further reduced.

No	Path	Data Plane Sequences
1	1.2	$Packet(LLDP)$
2	1.3.4	$Packet(A \rightarrow B)@p1$ $Packet(A \rightarrow B)@p1$
3	1.3.6.5	$Packet(A \rightarrow B)@p1'$ $Packet(B \rightarrow A)@p2$
4	1.3.5.7	$Packet(A \rightarrow B)@p1$ $Packet(B \rightarrow A)@p2$ $Packet(A \rightarrow B)@p1$
5	1.3.5.8	$Packet(A \rightarrow B)@p1$ $Packet(B \rightarrow A)@p2$ $Packet(* \rightarrow B)@p2$

Figure 3: Test Sequences of MAC-Learning

Fig.3 shows the result of test generation covering all transitions of the composition. Firstly, we choose a transition and find the shortest preamble (i.e. the path from initial state to the start state of the transition). Then we connect the preamble and the transition to get a path. When we derive all paths, we can remove the paths included in the others and get the final $Paths$. According to the information of Packet-In messages, we can get the corresponding data plane sequences. For example, $Packet(A \rightarrow B)@p1$ is the data plane packet injected at port $p1$ with source address A and destination address B.

After test generation, we can use the generated data plane sequences as traffic template. We replace the MAC addresses and ports with different concrete host addresses and interfaces in the topology. Finally we can achieve systematic coverage of the topology.

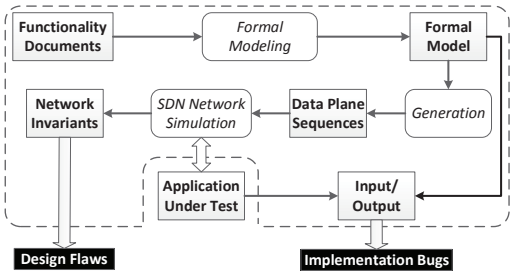

Figure 4: Overview of Our Test Tool

4. TESTING TOOL

Based on our formal model and corresponding test generation, we plan to implement our test tool for black-box testing of SDN applications. Fig.4 shows overview of our on-going testing tool. First, we model the application under test and generate data plane sequences from the model as stated in Section 3. Then, we build a simulation of the typical SDN network which suits the application under test and inject the data plane traffic following the guidance of generated sequences. Finally, we can check the network invariants for design flaws and observe the difference between applications and models for implementation bugs. The simulation and invariant checker are developed based on the methods of STS[7] and HSA[3].

5. CONCLUSION

In this paper, we propose a model based black-box testing method for SDN applications. We specify the applications with our model which describes the data structure into a group of parallel components. Based on our new model, we present a test generation approach to derive data plane sequences with partial model composition. It can cover the applications behaviors systematically and alleviate the sate explosion. For testing practice, we plan to develop our test tool using our test generation and SDN network simulation. It can expose both design flaws and implementation bugs.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61202357), the Project for 2012 Next Generation Internet technology research and development, industrialization, and large scale commercial application of China (No. 2012 1763). We also thank the support of HPL IRP (HP Labs Innovation Research Program).

6. REFERENCES

[1] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 282–293. ACM, 2014.

[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140, San Jose, CA, 2012. USENIX.

[3] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, USA, 2012. USENIX.

[4] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, Seattle, WA, 2014. USENIX.

[5] NOX. Pyswitch application. https://github.com/noxrepo/nox-classic/blob/destiny/src/nox/coreapps/examples/pyswitch.py.

[6] ONF. White paper (software-defined networking: The new norm for networks). https://www.opennetworking.org/sdn-resources/sdn-library/whitepapers.

[7] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox sdn control software with minimal causal sequences. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 395–406. ACM, 2014.