Is Your OpenFlow Application Correct?

Peter Perešíni, Marco Canini
EPFL, Switzerland
{peter.peresini,marco.canini}@epfl.ch

ABSTRACT

Nowadays OpenFlow enables controller programs to dynamically reconfigure the network. Controller programs can install, modify and delete packet processing rules as well as collect statistics from individual OpenFlow-enabled switches.

But how can we know if such programs are correct? While the abstraction of a logically-centralized network controller can ease their development, this abstraction does not remove the complexity of the underlying distributed system. For instance, small differences in packet header fields or packet timing can "tickle" subtle bugs [1].

We argue that there is a need for thorough automatic testing of OpenFlow applications. In this paper, we describe our preliminary experiences with taking two state-of-the-art model checkers (SPIN and Java PathFinder) and applying them "as is" for checking an example of OpenFlow program: a MAC-learning switch application. Overall, the preliminary results we report suggest that these tools taken out-of-the-box will have difficulties to cope with the large state-space explosion of OpenFlow networks.

1. MODEL CHECKING OPENFLOW

Model checking is an automatic approach for verifying the correctness of a system. Traditionally, model checking operates with a model that describes an abstraction of the system and discovers whether correctness properties asserted by the user are valid on the model. The main idea behind this approach is to systematically explore the space of all global states reachable from an initial state. Depending on the size of the system, the number of reachable states can become too high for current computing resources. Fortunately, a number of general techniques exist that help to mitigate this problem. For instance, partial order reduction (POR) is a technique that avoids exploring the sequences of transitions when the relative ordering of independent events is irrelevant to determine the final state; so, it is worth exploring with just one sequence.

Modern model checking uses the actual implementation as the model. This has the advantages that the user does not need to port the real application code to the modeling language and that the verification is more realistic. However, the sizes of the systems that can be

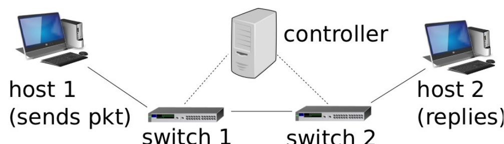


Figure 1: The network model used for model checking.

verified are limited because an actual program has many more states than its abstract model representation.

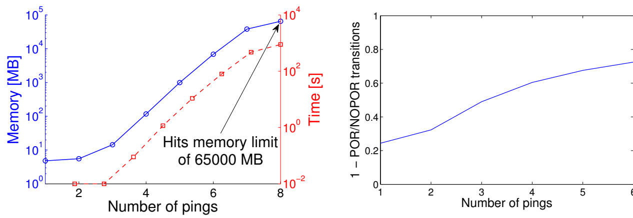
We believe model checking is a good fit for testing OpenFlow applications because we want to check safety properties (*e.g.*, absence of forwarding loops) that entail the global state of the network, *i.e.*, states of the controller and switches. Moreover, we expect these violations to be caused by unexpected interleavings of events and corner-case conditions, which are a convenient domain for applying model checking. Therefore, we want to quantify the state-space explosion in this verification problem and investigate the limitations of existing model-checking tools to constrain that explosion.

We consider a small network consisting of two OpenFlow switches driven by a single controller that mimics the functions of a simple MAC-learning switch[1]. As testing requires a closed system, we model two end hosts that behave in this simple way: host 1 sends a packet to host 2; host 2 replies with another packet. Figure 1 shows how these components are inter-connected.

1.1 Traditional Model-Checking: SPIN

Model. We model the system in PROMELA, the modeling language supported by SPIN[2], a very efficient model checker. This language exposes non-determinism as a first-class concept, making it easy to model the distributed behavior of our simple system. But we found the language to be lacking in expressiveness (*e.g.*, there is no support for procedures and common data structures). Finally, we need to cautiously capture the system concurrency at the right level of granularity. In our case, this means that any event at a single component (e.g., processing a packet on a switch) has to be modeled as a single atomic computation that cannot be

[1]Derived from NOX's `pyswitch` component.
[2]`http://spinroot.com`



(a) Memory usage and elapsed time (log y-scales). **(b) Efficiency of POR.**

Figure 2: SPIN: Exponential increase in computational resources partially mitigated by POR.

interleaved to any other event.

Experiments. To understand the scalability challenges in model-checking for OpenFlow, we perform an exhaustive search of the state space and we report on these metrics: memory usage, elapsed time, and number of transitions. We assign exclusive rights to the processes involved in each communication channel so to allow SPIN's implementation of POR to be most effective. We instruct SPIN to check for a simple correctness property: the absence of deadlocks. The final valid state of our system is when host 1 receives a reply to each packet it sends.

Results. Figure 2a shows the memory usage and elapsed time[3] for the exhaustive search with POR as we increase the number of packets sent by host 1. As expected, we observe an exponential increase in computational resources until SPIN reaches the memory limit when checking the model with 8 pings (*i.e.*, 16 packets).

To see how effective POR is, we compare in Figure 2b the number of transitions explored with POR vs. without POR (NOPOR) while we vary the number of pings. In relative terms, POR's efficiency increases, although with diminishing returns, from 24% to 73% as we inject more packets that are identical to each other. The benefits due to POR on elapsed time follow a similar trend and POR can finish 6 pings in 28% of time used by NOPOR. However, NOPOR hits the memory limit at 7 pings, so POR only adds one extra ping.

Finally, we test if POR can reduce the search space by taking advantage of one simple rule of independence for the networking domain: *i.e.*, packets involving disjoint pairs of source and destination addresses are completely independent. Unfortunately, we observe that there is no reduction when we inject two packets with distinct address pairs compared to the case with identical packets. This is because SPIN uses the accesses to communication channels to derive the independence of events.

1.2 Modern Model-Checking: Java PathFinder

Model. Using Java, we follow two approaches to write two models of the system (based on porting the original Python code) for Java PathFinder (JPF)[4], an explicit state model checker. In the first approach, we naively use threads to capture non-determinism. However, in our case, the built-in POR is not very efficient in removing unnecessary network event interleavings because thread interleaving happens at finer granularity than event interleavings. To solve this problem, we tuned this model by using the `beginAtomic()` and `endAtomic()` JPF functions. As this still produces too many interleavings, we further introduced a global lock.

In a second approach to further refine the model, we capture non-determinism via JPF's choice generator: `Verify.getInt()`. This gives a huge improvement over threads (shown in the results later), mainly because we are able to specify precisely the granularity of interleavings. However, there are several caveats in this case too. For example, explicit choice values should not be saved on the stack as the choice value may became a part of the global state, thus preventing reduction. The vector of possible transitions must also be sorted[5].

Experiments. We perform an exhaustive search with the default JPF settings and report on the following metrics: elapsed time, number of unique states, number of distinct end states, and memory usage.

Results. Table 1 illustrates the very fast exponential explosion when using the thread-based model. Unfortunately, as show in Table 2, the choice-based model improves only by 1 ping the size of the model that we can explore within a comparable time period (≈ 4 hours).

pings	time [s]	unique states	end states	mem [MB]
1	0	55	2	17
2	9	20638	134	140
3	13689	25470986	2094	1021

Table 1: JPF: Exhaustive search on thread-based model.

pings	time [s]	unique states	end states	mem [MB]
1	0	1	1	17
2	1	691	194	33
3	16	29930	6066	108
4	11867	16392965	295756	576

Table 2: JPF: Exhaustive search on choice-based model.

2. CONCLUSION AND FUTURE WORK

We presented our experiences with model checking for OpenFlow. While SPIN is fast, the main difficulty lies in writing the model. It took several person-days to implement the model. The modern model-checking approach of JPF solves the complexity of model specification but this comes at the cost of significant performance slowdown. To cope with the large state-space explosion of OpenFlow networks, our next step will be to supply the model checker with domain-specific knowledge (*e.g.*, independence based on packet header fields) that allows to reduce non-interesting interleavings.

3. REFERENCES

[1] M. Canini *et al.*. Automating the Testing of OpenFlow Applications. In *WRiPE*, 2011.

[3] The machine where we run the experiments has 64 GB of RAM and a clock speed of 2.6 GHz.

[4] `http://babelfish.arc.nasa.gov/trac/jpf`

[5] We order events by their states' hash values.