SDN-Actors: Modeling and Verification of SDN Programs
Technical Report (including proofs)

Elvira Albert[1], Miguel Gómez-Zamalloa[1], Albert Rubio[2], Matteo Sammartino[3], and Alexandra Silva[3]

[1] Complutense University of Madrid, Spain
[2] Universitat Politècnica de Catalunya, Spain
[3] University College London, UK

Abstract. Software-Defined Networking (SDN) is a recent networking paradigm that has become increasingly popular in the last decade. It gives unprecedented control over the global behavior of the network and provides a new opportunity for formal methods. Much work has appeared in the last few years on providing bridges between SDN and verification. This paper advances this research line and provides a link between SDN and traditional work on formal methods for verification of distributed software—actor-based modelling. We show how SDN programs can be seamlessly modelled using *actors*, and thus existing advanced model checking techniques developed for actors can be directly applied to verify a range of properties of SDN networks, including consistency of flow tables, violation of safety policies, and forwarding loops.

1 Introduction

SDN is a novel networking architecture which is now widely used in industry, with many companies –such as Google and Facebook– using SDN to control their backbone networks and datacenters. The core principle in SDN is the separation of the control and data planes –there is a centralized *controller* which operates a collection of distributed interconnected switches. The controller can dynamically update switches' policies depending on the observed flow of packets, which is a simple but powerful way to react to unexpected events in the network. Network verification has become increasingly popular since SDN was introduced, because in this new paradigm the amount of detailed information available about network events is rich enough and can be centrally gathered to check for properties, both statically and dynamically, of the network behavior. Moreover, the controller itself is a program which can be analyzed. The distributed and concurrent nature of network behavior makes the verification tasks challenging and has inspired much research in the verification and formal methods communities.

This paper provides a new bridge between SDN and a strand of formal methods –actor-based modeling [2], which is a framework that was developed to analyze concurrent systems. Actors form the basic unit of computation in such

framework, are equipped with a private memory, and can interact with others through *asynchronous* messages. This setup enables reasoning about local properties without knowledge of the whole program, which gives rise to more compositional and thus scalable methods. Actors provide the foundations for the concurrency model of languages used in industry, e.g., *Erlang* and *Scala*, and libraries used in mainstream languages, e.g., *Akka*.

Contributions. The main contributions of this paper are:

1. SDN-Actors: An encoding of all components of an SDN network into the actor-based language ABS [13]. One of the most challenging aspects to encode were the OpenFlow *barrier* messages, special instructions that the controller can use to force switches to execute all their queued tasks.
2. A soundness proof of the encoding (and implementation) of *barriers* (Th. 2).
3. Application of (context-sensitive) dynamic partial-order reduction (DPOR) techniques to model check SDN programs. We have implemented this model checker on top of the SYCO tool [4] for actors.
4. Several case studies of SDN and properties to illustrate the versatility and potential of the approach. We were able to find bugs related to programming errors in the controller, forwarding loops, and violation of safety policies.

Though we did not explore it in this paper, the encoding we provide opens the door to apply a range of techniques other than model checking. For instance, static analysis, runtime monitoring or simulation of network behavior can be done now using the ABS toolsuite [1]. Other tools and methods for verification of message-passing and concurrent-object systems could be also easily adapted [6, 8, 15, 16]. In addition, because the encoding is not very far from the original flow tables, both model extraction from existing network code and code generation from an actor model should be achievable with a small extension of the tool.

2 Overview

This section provides an overview of the contents of the paper through an extended example, that we also use to introduce some basic concepts and notations.

2.1 Concurrency errors in SDN networks

SDN is a networking architecture where a central software *controller* can dynamically change how network switches forward packets by monitoring the traffic. Switches can be connected to hosts and to other switches via bidirectional channels that may reorder packets. Each switch has a *flow table*, that is a collection of guarded forwarding rules to determine the route of incoming packets. Whenever a switch receives a packet, it checks if one of the flow table rules applies. If no rule applies, the switch sends a message to the controller via a dedicated link, and the packet is buffered until instructions arrive. Depending on its policy, the controller instructs the switch, and possibly other switches in the network, on

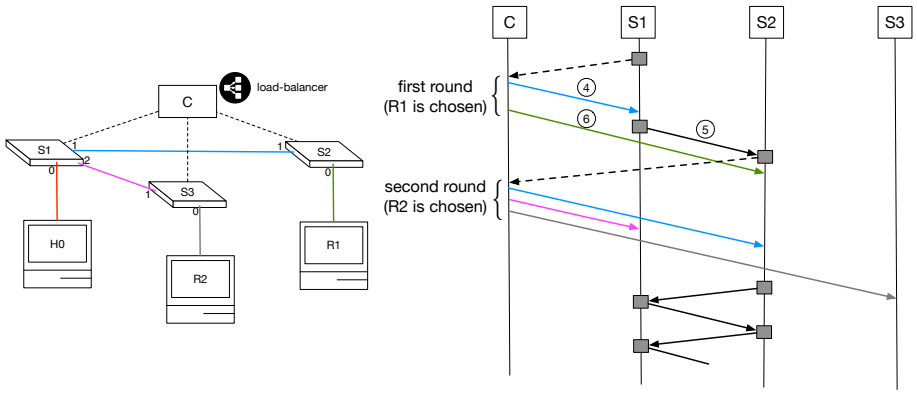


Fig. 1. Example SDN load-balancer. On the left: structure of the SDN. On the right: messages exchanged in a possible execution of a naive controller program. Coloured arrows stand for control messages to switches, indicating which flow rule to install (colours specify the link to be used for the forwarding). Grey boxes and arrows among them represent packet forwardings. Dashed arrows indicate messages to the controller.

how to update their flow tables. Such control messages between the controller and the switches can be processed in arbitrary order.

We now show how a simple load-balancer can be implemented in SDN (example taken from [11]) and how potential bugs can easily arise due to the concurrent behavior and asynchrony of message passing. Suppose we want to balance the traffic to a server by using two replicas R1 and R2 to which the controller alternates the traffic in a round-robin fashion. The structure of the SDN is shown in Fig. 1, on the left: H0 is any host that wants to communicate with the server and S1, S2 and S3 are switches (numbers on endpoints stand for port numbers).

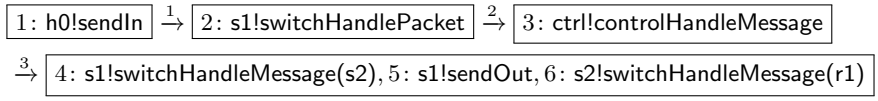
Even in this simple network, an incorrect implementation of the controller can lead to serious problems. In Fig. 1, on the right, we show an execution of a naive controller, which simply instructs switches to forward packets along the shortest path to the chosen replica. This implementation ignores the potential concurrency in actions taken by switches and controller, leading to a forwarding loop between S1 and S2. In the first round, when S1 queries the controller, R1 is chosen. The figure shows S1 forwarding the packet to S2 before the end of the first round, i.e., before a rule is installed on S2 (green arrow). This causes S2 to query the controller, which triggers the second round in which the controller chooses R2. Thus, it sends instructions to install rules on S2, S1 and S3 to forward the packet to S1, S3 and R2, resp. When the controller rules arrive at S1, it will have two contradictory instructions, telling to forward the packet either to S2 or to S3. In the former, the loop at the bottom of the figure occurs. This issue can be avoided if the implementation uses barriers –the controller will then guarantee that S2 receives and processes control messages before taking any other action.

2.2 Actor-based modeling of SDN networks

We now explain how we can automatically detect the above problem using actors and model checking. We use the object-oriented actor language ABS [1, 13], where each actor type is specified as a class, consisting of a set of fields and methods. Actors are instances of actor classes. For instance, the instructions: Controller ctrl = new Controller(); Switch s1 = new Switch("S1",ctrl); Host h0 = new Host("H0",s1,0); create three actors: a controller ctrl; a switch s1 with name "S1" and a reference to ctrl; a host h0, with name "H0", connected to the switch s1 via the port 0. The SDN in Fig. 1 can be modeled using one actor per component (additional data structures for network links will be shown later).

The execution model of actors is *asynchronous*. Each actor can be thought of as a processor, with a queue of pending tasks and a local memory. Actors are executed in parallel and, at each actor, one task is *nondeterministically* selected among all the pending ones and executed. The syntax Fut<type> f=a!m($\overline{x}$) spawns an asynchronous task m($\overline{x}$), that is added to the queue of pending tasks of a, type is the type of the data returned by m or Unit if no data is returned. This task consists in executing the method m of a with arguments $\overline{x}$. The variable f is a *future variable* [9] that will allow us to check if such task has been completed. *Synchronous* calls are written a.m($\overline{x}$), we omit a if the target actor is this.

A partial trace of execution of our SDN actor model computed by the model checker is (the code that the tasks below execute will be given in Sec. 3):



Intuitively, a packet sending (sendIn) is executed on h0 (label 1), which causes the packet to be forwarded to the switch s1 (2), then s1 sends a control message to the controller (3). Finally, the controller spawns the three tasks in the last state (parameters tell where to forward the packet). When executed, these tasks will produce the messages in Fig. 1 with the same numbers. Their execution order is arbitrary: if it is the one shown in Fig. 1, the execution trace may lead to a state exhibiting a forwarding cycle between s1 and s2. As we will show later, this situation can be easily detected by our model checker via an exploration of a *reduced* execution tree, which avoids equivalent executions (Sec. 4).

The ABS language provides a convenient await primitive that will be used to model barriers and to rule out the behavior described above. The instruction await f? can be used to synchronize with the termination of the task associated to the future variable f, by releasing the processor (so that another task can be scheduled) if the task is not finished. Once the awaited task is finished, the suspended task can resume. The await can be used also with boolean conditions await b? to suspend the execution of the current active task until condition b holds. The formal semantics of the language can be found in App. A.

3 SDN-Actors: an actor based encoding of SDN programs

We present the concept of *SDN-Actor* in four steps: Sec. 3.1 describes the creation and initialization of the actors according to the topology. Sec. 3.2 provides the encoding of the operations and communication for Switch and Host actors. Sec. 3.3 proposes the encoding of the controller, and Sec. 3.4 the extension to implement barriers. Altogether, our encoding provides an actor-based semantics foundation of SDN networks that follow the OpenFlow specification [18].

3.1 Network topology

The topology can be given as a relation with two types of links:

- *SHlink(s,h,o)*: switch s is connected to host h through the port o
- *SSlink(s_1,i_1,s_2,i_2)*: switch s_1 is connected via port i_1 to port i_2 of switch s_2

from which we automatically generate the initial configuration as follows.

Definition 1 (initial configuration). *Let S and H be, respectively, the set of different switch and host identifiers available in the* link *relations that define the network topology. The initial configuration (method* main*) is defined as:*

- *We create a controller actor* Controller ctrl=**new** Controller()
- *For each* sid$\in S$*, we create an actor* Switch s=**new** Switch(sid,ctrl)
- *For each* hid$\in H$*, we create an actor* Host h=**new** Host(hid,s,o) *where* s *is the reference to the switch actor,* o *the port identifier, that* hid *is connected to.*
- *The data structures* srefs *and* hrefs *store, resp., the relations between identifier in the topology and reference in the program, for all switches in S and hosts in H.*
- *The data structure* ntw *contains the link relations in the network topology.*
- *The synchronous call* ctrl.addConfig(srefs,hrefs,ntw) *initializes in the controller the topology relations and the references to switches and hosts s.t. the controller can send control messages to redirect the traffic to the involved links.*

Example 1. By applying Def. 1 to the topology in Fig. 1, given as the relation: $SHlink(S1, H0, 0)$, $SHlink(S2, R1, 0)$, $SHlink(S3, R2, 0)$, $SSlink(S1, 1, S2, 1)$, $SSlink(S1, 2, S3, 1)$, we obtain the following initial configuration which constitutes the main method from which the execution starts:

```
1 main() { Controller ctrl = new Controller(); Switch s1 = new Switch("S1",ctrl);
2          Switch s2 = new Switch("S2",ctrl); Switch s3 = new Switch("S3",ctrl);
3          Host h0 = new Host("H0",s1,0); Host r1 = new Host("R1",s2,0);
4          Host r2 = new Host("R2",s3,0);
5          Map<SwitchId,Switch> srefs = {"S1":s1, "S2":s2, "S3":s3};
6          Map<HostId,Host> hrefs = {"H0":h0, "R1":r1, "R2":r2};
7          List<Link> ntw = [SHLink("S1","H0",0), SSLink("S1",1,"S2",1),..];
8          ctrl.addConfig(srefs,hrefs,ntw); }
```

The data structures srefs and hrefs are implemented using maps, and the network ntw as a heterogeneous list. The use of data structures is nevertheless orthogonal to the encoding as actors. We just assume standard functions to create, initialize, access them (like getters, put, lookup, etc.) that will appear in italics in the code.

```
9  type SwitchId=... type HostId=... type PortId=... type PacketId=...
10 type PacketH=... type Packet=... type Action=... type Link=...
11 type MatchF=(PacketH,PortId);

12 class Host(HostId hid, Switch s, PortId o) {
13    Unit sendIn(Packet p){ s!switchHandlePacket(p,o);}
14    Unit hostHandlePacket(Packet p){ /*output packet*/}
15 }
16 class Switch(SwitchId sid, Controller ctrl) {
17    Map<MatchF,Action> flowT={};
18    Map<PacketId,(Packet,PortId)> buffer={};
19  Unit switchHandlePacket(Packet p, PortId o){
20     Action l=lookup(flowT,(getHeader(p),o));
21     if (isSwitch(l)) getSwitch(l)!switchHandlePacket(p,getPort(l));
22     else if (isHost(l)) getHost(l)!hostHandlePacket(p);
23     else { buffer=put(buffer,getId(p),(p,o));
24            ctrl!controlHandleMessage(sid,o,getId(p),getHeader(p)); }}
25  Unit sendOut(PacketId pi){
26     Packet p; PortId o; (p,o)=lookup(buffer,pi);
27     Action l=lookup(flowT,(getHeader(p),o));
28     if (isSwitch(l)) getSwitch(l)!switchHandlePacket(p,getPort(l));
29     else if (isHost(l)) getHost(l)!hostHandlePacket(p);
30     /*else packet is dropped*/}
31  Unit switchHandleMessage(MatchF m, Action a){ flowT=put(flowT,m,a);}
32 }
```

Fig. 2. Type declarations (top) and actor-based host and switch classes (bottom)

3.2 The switch and host classes

Fig. 2 presents the actor-based Switch and Host classes. We include at the top some **type** declarations that are assumed and must be implemented (such as identifiers, packets and their headers, etc.). There are two main data structures that are implemented in more detail to make explicit the information they contain:

- the buffer at Line 18 (L18 for short) is a *map* that must contain pairs of packet and input port indexed by their PacketId.
- the flow table flowT (L17) is implemented as a *map* indexed by the so-called *match field* [18] represented by type MatchF in Fig. 2. The match field is composed by information stored in the header of a Packet (retrieved by function *getHeader*) and the input port. For a given matching, the flow table contains the Action the switch has to perform upon the reception of the Packet. An action l can be of three types: i) send the packet to a host h, ii) send the packet to the port o of a switch s, iii) drop the packet. Given an action l, function *isSwitch* resp. *isHost* succeeds if the action is of type ii) resp. i), and functions *getSwitch*, *getHost* and *getPort* return the s, h and o resp. The full implementation must allow duplicate entries (non-deterministically

selected), and the use of wildcards in the match fields, but these aspects are unrelated to the encoding of SDN actors, and skipped for simplicity.

Upon creation, hosts receive their identifier and a reference to the switch and the port identifier they are connected to (defined as class parameters that are initialized at the actor creation). Their method sendIn is used to send a packet to the switch, and method hostHandlePacket to receive a packet from the switch. Switches receive upon creation their identifier and a reference to the controller. They have as additional fields: (a) the flow table flowT (as described above) in which they store the actions to take upon receiving each kind of package, and (b) a buffer in which they store packets that are waiting for a response from the controller. Switches can perform three operations: (1) switchHandlePacket receives a packet, looks up in the flow table the action to be made L20, and, if there is an entry for the packet in the table, it asynchronously makes the corresponding action (either send it to a host L22 or to a switch L21). Otherwise, it sends a controlHandleMessage request and puts the packet and input port in the buffer (L23 and L24) until it can be handled later upon receipt of a sendOut; (2) sendOut receives a packet identifier that corresponds to a waiting packet, retrieves it from the buffer (L26), looks up the action l to be performed in the flow table, and makes the corresponding asynchronous call (as in switchHandlePacket); (3) switchHandleMessage corresponds to a message received from the controller with an instruction to update the flow table. Other switch operations like *forward packet*, that is similar to sendOut but directly tells the switch the action to be performed, or *flood*, that sends a packet through all ports except the input port, can be encoded similarly and are used in the experiments in Sec. 5.

Example 2. In main, after L8, we add h0!sendIn(p), where p is a packet to be sent to the IP address of the replica servers (the information on the destination is part of the packet header). This is the only asynchronous task that main spawns. Its execution in turn spawns a new task s1!switchHandlePacket(p,0) at L13, that does not find an entry in flowT at L20 and spawns a controlHandleMessage task on the controller at L24, whose code is presented in the next section.

3.3 The controller

After creating the controller actor, the method addConfig is invoked synchronously to initialize the references to switches and hosts and set up the initial network topology (see L8). A simple controller is presented in Fig. 3, removing the blue lines 35, 36, 41, 44, 46, 48, 49 which provide the implementation of barriers. When a switch asynchronously invokes controlHandleMessage, the controller applies the current policy—applyPolicy must be implemented for each different type of controller. The implementation of the policy typically requires the definition of new data structures in the controller to store additional information (see Sec. 5). When applying the policy, we obtain a list of switch identifiers and corresponding actions to be applied to them. The while loop at L42 in controlHandleMessage asynchronously invokes switchHandleMessage at L45 on each of the switches in

```
33 class Controller() {
34   Map<SwitchId,Switch> srefs={}; Map<HostId,Host> href={}; List<Link> ntw=[];
35   Map<SwitchId,List<Fut<Unit>> barrierMap={};
36   Set<SwitchId> barrierOn = ∅;
37   Unit addConfig(Map<SwitchId,Switch> sr, Map<HostId,Host> hr, List<Link> n){
38         /* references to switches and hosts and network topology initialized */ }
39   Unit controlHandleMessage(SwitchId sid, PortId o, PacketId p, PacketH h){
40        List<(SwitchId,MatchF,Action)> l=applyPolicy(sid,o,h);
41        List<SwitchId> ls = [];
42        while (not(isEmpty(l))) {
43             SwitchId s1; Action a1; MatchF m1; (s1,m1,a1)=head(l);
44             barrierWait(s1);
45             Fut<Unit> f=lookup(srefs,s1)!switchHandleMessage(m1,a1);
46             barrierMap=putAdd(barrierMap,s1,f); ls = add(ls,s1);
47             l=tail(l);}
48         while (not(isEmpty(ls))) {barrierRequest(head(ls)); ls=tail(ls);}
49         barrierWait(sid);
50         lookup(srefs,sid)!sendOut(p);
51    }
52   List<(SwitchId,MatchF,Action)> applyPolicy(SwitchId sid, PortId o, PacketH h) {
53         /* implementation of specific policy */}
54    }
```

Fig. 3. Controller class w/o barriers in black (w/ barriers extended in blue)

the list, and passes as parameter the corresponding action to be applied for the given match entry. Finally, it notifies at L50 the switch from which the packet came that the packet can already be sent out. More sophisticated controllers that build upon this encoding are described in Sec. 5.

Example 3. In the example, applyPolicy corresponds to the load-balancer described in Sec. 2, which directs external requests to a chosen replica in a round-robin fashion. For the call applyPolicy(s1,0,h), it chooses r1 and thus, it returns in L40 two actions: (s1→s2), (s2→r1), i.e., one action to install in s1 the rule to send the packet to s2, and the second to install in s2 the rule to send it to r1. For simplicity, we assume that the Action just contains the location to which the packet has to be sent (without including the port). The while loop thus spawns two asynchronous calls, s1!switchHandleMessage(m1,s2) and s2!switchHandleMessage (m1,r1). Besides, it sends a s1!sendOut(p) in L50. Several problems may arise in this implementation. One problem, as explained in Sec. 2, is that the packet is sent from s1 to s2 before the control message is processed by s2. Then, s2 gets the packet and it does not find any matching rule, thus it sends a controlHandleMessage to the controller. Applying the above policy, the controller chooses now as replica r2 and returns the actions: (s2→s1), (s1→s3), (s3→r2), i.e., the packet should be sent to r2 by first sending from s2 to s1 (first action), and so on. This might create the circularity depicted in Fig. 1.

```
55    Unit barrierWait (SwitchId sid){
56        await not(contains(barrierOn,sid))?;
57    }
58    Unit barrierRequest (SwitchId sid){
59        barrierOn=add(barrierOn,sid);
60        List<Fut<Unit>> futSid=lookup(barrierMap,sid);
61        while (not(isEmpty(futSid)) {
62            Fut<Unit> fi=head(futSid);
63            await fi?;
64            futSid=tail(futSid); }
65        barrierOn=delete(barrierOn,sid);
66    }
```

Fig. 4. Implementation of barriers (part of class Controller)

The following theorem ensures the soundness of our modeling. Essentially we guarantee that, for a given SDN network that follows the OpenFlow specification, any execution in the network has an equivalent execution in the SDN-Actor model. An execution in the network is characterized by the messages in the queues of the switches, hosts, and controller and the state of their data structures. An *equivalent* execution in the model will thus ensure the same messages in the actors queues and the same state in actors data structures.

Theorem 1 (soundness). *Given a SDN network N, consider its SDN-Actor model N^a with an initial configuration* main *obtained by Def. 1, and the* Switch, Host *and* Controller *classes in Figs. 2 and 3. Then, for each execution in N, there exists an* equivalent *execution trace in N^a using the semantics of App. A.*

3.4 Barriers

Barriers [18] have been designed to force a switch to handle previous control messages, and thus avoid problems such as the one described above.

Definition 2 (OF barrier). *Following OpenFlow [18], upon receipt of a* barrier message*, the switch must finish processing all previously-received controller messages, before executing any messages beyond the* barrier message*.*

Figs. 3 and 4 show our modeling that intuitively consists in the controller not sending further messages to any switch on which a barrier has been activated, until this switch acknowledges that all previous control messages have been already processed. The main points in the implementation are: (1) the controller creates a future variable at L45 for every asynchronous task that it posts on all switches; (2) it keeps in barrierMap the list of future variables (not yet acknowledged) for each of the switches (putAdd in L46 adds the future variable to the list indexed by s1 in the map); (3) it keeps in barrierOn the set of switches with an active barrier; (4) a barrier on a switch consists in the controller awaiting

on the list of future variables that the switch needs to acknowledge to ensure that its control messages have already been processed (method barrierRequest); (5) all control messages must be now preceded by an invocation to barrierWait that checks if the corresponding switch has an active barrier, L56. This is because while suspended in a barrier, the controller can start to process another controlHandleMessage unrelated to the previous one, but which affects (some of) the same switches for which a barrier was set. So, we cannot send messages to them until their barriers are set to off. Note that this is not a restriction on the type of controllers we model, but rather an effective way to encode barriers using actors and **await** instructions that ensures the behaviour of OpenFlow barriers.

Theorem 2 (soundness of barriers). *Methods barrierRequest and barrierWait provide a sound encoding of the OF* barrier *messages in Def. 2.*

4 DPOR-based model checking of SDN-Actors

Model checking tools deal with a combinatorial blow-up of the state space (a.k.a. the state space problem) that must be faced to solve real-world problems. As for model checking SDN programs, the problem is exacerbated because of the concurrent and distributed nature of networks: all network components (switches, hosts, controllers) are distributed nodes that run in parallel and whose concurrent tasks can interact. As we have seen, a controller message sent from a switch can change the state of another switch, and affect the route of an incoming packet. Thus, a model checker needs to explore all possible reorderings of *dependent* tasks (i.e., those whose execution might interfere with each other) leading to a huge number of possible executions even for networks with few nodes and few packets. Besides, the space is unbounded because hosts may generate unboundedly many packets that could be simultaneously traversing the network.

There are two *incomplete* approaches to handle unbounded inputs: one is to impose a bound k on the number of packets of each type (as e.g. in [7]) and the other one is to use abstraction (as e.g. in [17]). In the former, the search space is exhausted for the considered input, but there could be bugs that only show up when more packets are considered. In the latter, abstraction requires to lose information and bugs may only show up when the omitted information is considered. Therefore, the sources of incompleteness are different, and the approaches can complement each other. Our implementation uses the former, e.g., in Ex. 2 we have considered one packet (limit $k = 1$). The rest of the section presents the key features of our approach assuming such a k bound.

4.1 DPOR-based model checking in actors

DPOR [12] is able to dynamically identify and avoid the exploration of redundant executions and prune the search space exponentially. It is based on the idea of initially exploring an arbitrary interleaving of the various concurrent tasks, and *dynamically* tracking dependent interactions between them to identify backtracking points where alternative paths in the state space need to be explored.

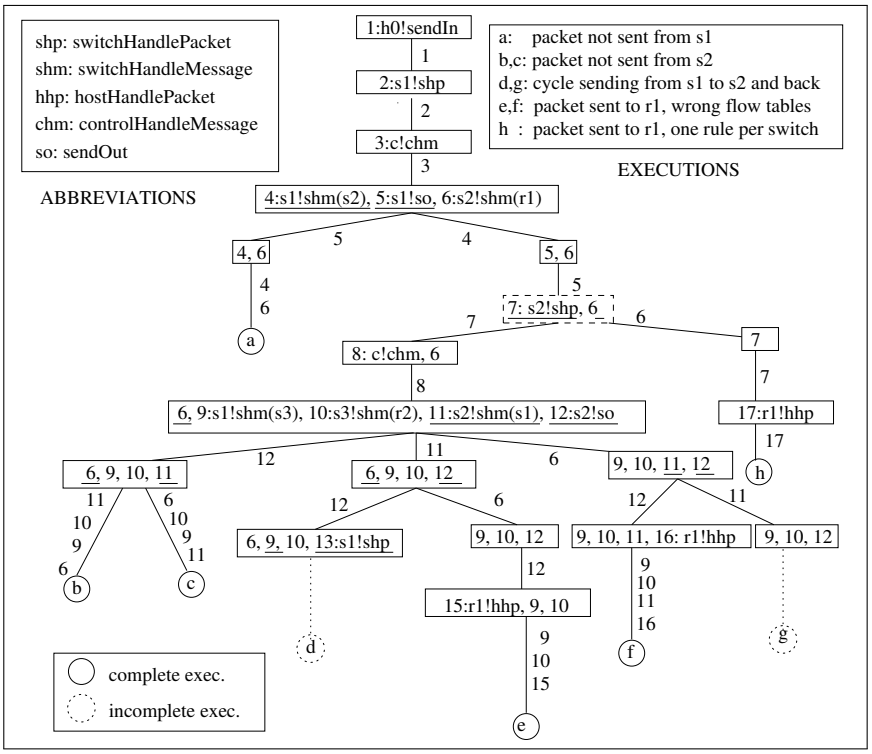


Fig. 5. Search tree for running example w/o barriers (rightmost branch w/ barriers)

Two tasks are *independent* when changing their order of execution will not affect their combined effect. When DPOR is applied to actor systems, there are inherent reductions [22] because: (i) we can atomically execute each task (without re-orderings) until a return or an **await** instruction are found, because concurrency is non-preemptive and the active task cannot be interrupted. This avoids having to consider the reorderings at the level of instructions (as one must do in thread-based concurrency), and allows us to work at the level of tasks. (ii) Besides that, two tasks can have a dependency only if they belong to the same actor. This is because only the actor itself can modify its private memory.

Example 4. Fig. 5 shows the search tree computed by DPOR for our SDN-Actor program without barriers. It has no redundancy, i.e., each execution corresponds to a different behavior on the packet arrival and/or the actions installed in the flow tables (see top right descriptions). At each node (i.e., state), we show the available tasks. A task is given an identifier the first time it appears, and afterwards only its identifier is shown. Method names are abbreviated as shown in the top left, and parameters are omitted except in tasks executing switchHandleMessage, for which we only include the switch identifier that is part of the Action to be installed. For instance, 4:s1!shm(s2) is a task with identifier 4, that will execute method switchHandleMessage on s1 and will add to its flow

table the information that the packet must be sent to s2. Labels on the edges show the task(s) that have been executed. At each state, we underline the tasks which have an interacting dependency. The execution starts by executing the main method in Ex. 1 with the instruction sendIn added in Ex. 2 which appears in the root. The next two steps have one task available, but in the fourth state we have tasks 4 and 5, belonging to the same actor, whose reordering needs to be considered (leading to branching), while 6 is independent of them. Out of the 8 branches of the tree, only the rightmost execution ⓗ corresponds to the correct behavior in which the packet is actually sent to r1 and the actions are installed in the flow tables in the expected order. In execution ⓐ the packet does not arrive at the destination because the sendOut is executed before the action has been installed. Executions ⓓ and ⓖ correspond to the cycle described in Sec. 2, each of them with different installations of actions.

Importantly, we do not need specific optimizations to use the DPOR algorithm in [3] to model check SDN-Actors. The use of **await** (is already covered by DPOR and) does not require any change either and, as expected, the search tree for the implementation with barriers only contains branch ⓗ. The difference arises from task 3 in the tree: in the presence of barriers, this leads to a state in which we have the asynchronous calls 4 and 6 and task 3 suspended at the **await** in L63 (awaiting first the termination of 4 and then that of 6). Therefore, the dependent tasks 4 and 5 will not coexist because 5 is not spawned until 4 and 6 terminate.

4.2 Entry-level and context-sensitive independence

When two tasks that belong to the same actor are found, in the context of DPOR techniques independence is commonly over-approximated by requiring that actor fields accessed by one task are not modified by the other. In our model, all tasks posted on a given switch access its flow table, namely sendOut and switchHandlePacket read it and switchHandleMessage writes it. Thus, in principle, any task executing switchHandleMessage is considered dependent on the other two. This explains the tasks underlinings in the figure and the branching in the tree. When there are multiple packets traversing the network it is usually the case that the different packets access distinct entries in the flow table. This results in the inaccurate detection of many dependencies hence producing redundant executions. Using Context-sensitive DPOR [3], we alleviate this state explosion:

1. *Entry-level independence.* We adopt a finer-grained notion of *entry-level independence* for which an access to entry i is independent from an access to j if $i \neq j$. This aspect is not visible when considering a single packet as in the example, as all accesses to the flow table refer to the same entry. However, by simply adding another packet to the erroneous program, the state explosion is huge and the system times out if entry-level independence is not implemented, while it computes 92 executions (exploring 761 states) with entry-level independence.
2. *Context-sensitiveness.* Even when two tasks t and p access the same entry, Context-sensitive DPOR introduces some further checks that execute the

considered tasks from the current state S in the two orders $t \cdot p$ and $p \cdot t$. If they lead to the same state, one of the derivations is pruned and further exploration from it is thus avoided. For instance, executing two consecutive switchHandleMessage on the same entry might lead to the same state if the flow table may contain duplicate entries, as our implementation allows.

4.3 Comparison of DPOR reductions with related work

Other model checkers for SDN programs have used DPOR-based algorithms before [7,17]. According to the experiments in the NICE tool, DPOR only achieves a 20% reduction of the search space because even the finest granularity does not distinguish independent flows. The reason for this modest reduction might be that it does not take advantage of the inherent independence of the code executed by the distributed elements of the network (switches, host, clients), nor to the fact that barriers allow removing dependencies, as our actor-based SDN model does. In Kuai [17], a number of optimizations are defined to take advantage of these aspects. Such optimizations must be (1) identified and formalized in the semantics, (2) proven correct and, (3) implemented in the model checker. Instead, due to our formalization using actors, the optimizations are already implicit in the model and handled by the model checker without requiring any extension. Another main difference with Kuai is that they make two important simplifications to the kind of SDNs they can handle: (i) they assume a simplified model of switches in which a switch gets suspended (i.e., does not process further packets nor controller messages) while awaiting a controller request. The error showed in Ex. 1 would thus not be captured. We do not make any simplification and thus a switch can start to process a new packet while awaiting the controller and can also receive other controller actions (triggered by other switches). (ii) It works on a class of SDNs in which the size of the controller queue is one. Therefore, it will not capture potential errors that arise due to the reordering of messages by the controller. In contrast, our model checker works on the general model of SDN networks.

5 Checking SDN properties in case studies

We have built the extension for property checking on top of SYCO [4], a system that implements context-sensitive DPOR exploration. To evaluate our approach, we have implemented a series of standard SDN benchmarks used in previous work [5, 11, 17]. In order to check property P we add to the controller a new method called `error_message` and encode P as a Boolean function F_p using the programming language itself. Then, in all places where the property has to hold, we add an `if` statement checking the negation of F_p and if it holds we call asynchronously to `error_message` on the controller. Then property holds for the given input if and only if there is no trace in the execution tree including a call to `error_message`.

Our goal is on the one hand to show the versatility of our approach to check properties that are handled using different approaches in the literature (e.g.,

Name	Switch x Host x Packet	Execs	States	Time
LB	3x3x1/3x4x2	8/92	64/761	15/263
LBB	3x4x2/3x7x5/3x10x8/3x12x10	3/21/171/683	48/482/3996/16028	13/212/3542/22941
SSHE	2x2x(1ssh/1oth/2each/2cor)	9/21/2648/1201	56/135/24116/9406	14/37/12308/3276
SSHB	2x2x2/2x2x3	27/2013	318/23643	119/13261
MI	1x5x(8/10/11)	122/753/1506	2710/17613/35870	1003/11800/34894
MIB	1x5x(8/10/11)	138/831/1653	3215/20640/41512	1668/18591/53349
LE	3x3x(2/5)/6x3x2	10/46/40	178/1269/1239	59/467/649
	6x3x5/9x2x2	132/944	5765/12339	3798/12230

Table 1. Experimental evaluation

programming errors in the controller as in [5], safety policy violations as in [5,17], or loop detection as in [11]). And, on the other hand, to show that we are able to handle networks at least as large as (and sometimes larger than) in related systems [17], but without requiring simplifications to the SDN models, nor extensions for DPOR reduction, and in spite of using a non-distributed model checker. We should note that a precise comparison of figures is not possible due to the differences described in Sec. 4.3 and the use of different implementations of controllers. Our system can be tried online at `http://costa.ls.fi.upm.es/syco` using the POR algorithm `CDPOR` and disabling the automatic generation of independence constraints. The benchmarks can be found in the folder FM18.

Table 1 shows a summary of the experimental results. Times are obtained on an Intel Core i7 at 3.4Ghz with 8GB of RAM (Linux Kernel 3.2). For each benchmark, we show in the second column the number of switches, hosts and packets, **Execs** corresponds to the number of different executions (i.e., branches in the search tree), **States** to the number of nodes in the search tree, and **Time** is the time taken by the analysis in ms. Although entry-level independence can be proved automatically, this is not yet implemented in SYCO and we have used annotations to declare it. As an example, in method switchHandleMsg, the annotation: [indep(switchHandlePacket(pin,pkt),!matchHeaderAndPort(getHeader(pkt), pin,m))] states that tasks executing switchHandleMessage(m,a) are independent of those executing switchHandlePacket(pin,pkt) if the match field of the message does not match the header and the input port of the packet (the condition is checked by the auxiliary function matchHeaderAndPort).

Controller with load balancer [11] (LB/LBB). This corresponds to the controller of [11], similar to our running example. It performs stateless load balancing among a set of replica identified by a virtual IP (VIP) address. When receiving packets destined to a VIP, the controller selects a particular host and installs flow rules along the entire path. For a buggy controller without barriers (LB) and a network with 3 switches and 3 hosts, we detect that there is a forwarding loop (i.e., that a packet reaches a switch more than once) in 15ms. For this, we have added to the switches a field to store the packet identifiers that they have already received, and when the same packet reaches it, it sends an error message, which is observable from the final state. When we add a second packet with the same header and another host, as expected, the number of dependencies increases and, many reorderings need to be tried, leading to 92 different executions and 761 states. Once we check the correct version with

barriers (LBB), we are able to scale up to 12 hosts and 10 packets. As it can be observed, for the largest network, 16028 states are explored and in all cases we verify that the traffic is balanced. The experiments in [11] do not specify the time to detect the bug for this controller (they only mentioned that their analysis finishes in less than 32s in the vast majority of cases). Nevertheless, the underlying techniques to find the bugs are unrelated (see Sec. 6), and thus time comparison is not meaningful.

SSH controller [17] (SSHE/SSHB). This is based on a controller that dynamically modifies the behaviors of the switches as follows: it can update the switches with a rule that states that no SSH packets are forwarded, and another that states that all non-SSH packets are forwarded. We have two versions of the SSH controller. In the row SSHE, the first three evaluations correspond to an erroneous SSH controller that installs the rule to forward packets and the rule to drop SSH packets with the same priority, and thus the safety policy can be violated. As in [17], we evaluate a network with 2 switches and 2 hosts. As for packets, we write 1ssh, 1other, and 2each to indicate that we send one SSH packet, one non-SSH packet and one of each type. We detect the error by checking in the switch if two contradictory drop and forward packet actions are received for the same entry. The results that we obtain for 1 packet are in the same order of magnitude as [17]: they produce 13 executions, while we produce 9 or 21, depending on the type of packet. Analysis times are also similar: 0.1s in their case versus 0.014s or 0.037s in our case. This is as expected because there is almost no redundancy using plain DPOR, thus no need for our entry-independence or context-sensitiveness. When we add more packets, the number of dependencies grows exponentially. This is because the controller receives 2 requests from the 2 messages, and sends dependent control messages to all switches. Therefore, all reorderings must be tried and the state explosion is huge. The last evaluation 2cor corresponds to the correct SSH controller for which we achieve a notable improvement as we have now less tasks that match the same entry (as priority is different). The row SSHB is a correct implementation with barriers that reduces the number of executions for 2 packets notably because it guarantees that forward rules are installed and thus switches will not send further requests.

Firewall with migration [5] (MI/MIB). MI is the implementation of a firewall that supports migration of trusted hosts. A host is trusted if it either sent/received (on some switch) a message through/from port 1. Thus, when a trusted host migrates to a new switch, the controller will remember it was trusted before and will allow communication from either port. For the same network 1x5 as [5], we can scale the number of packets up to 11 packets that actually modify the data base for trusted hosts. We can keep on adding more packets if those do not affect the shared data base. In MIB, we introduce the same bug in the controller as [5], which forgets to check if trusted on events from port 2. We detect the error by checking in the final state of the derivations that a packet arrives to a host that is not in the trusted data base. The scalability of MI and MIB are rather similar. Both [5] and us find the bug in a negligible time.

Network authentication with learning [5, 17] (LE). This implements a composition of a learning switch with authentication in [5]. Also, [17] evaluates a MAC learning controller but using a different implementation. LE implements a controller with barriers for which we can verify that the packet flows satisfy the intended policy and that the flow tables are consistent. We have considered configurations of 3x3, 6x3 and 9x2. When compared to [17], we handle similar sizes for networks but we explore less **States** in less **Time**. We note that this might be due to different implementations of the controller and the differences pointed out in Sec. 4.3.

6 Conclusions and Related Work

We have proposed an actor-based framework to model and verify SDN programs. A unique feature of our approach is that we can use existing advanced verification algorithms without requiring any specific extension to handle SDN features. The last years have witnessed the development of many static and dynamic techniques for verification that are closely related to our approach. Using static approaches, one has the main advantage that, when the property can be proved, it is ensured for any possible execution, while using dynamic analysis only guarantees the property for the considered inputs. As a counterpart, in order to cover all possible behaviors, static analysis needs to perform abstraction, that can give a don't-know answer, and, possibly, false positives. In [5], the work on Horn-based verification is lifted to the SDN programming paradigm, but excluding barriers. Using this kind of verification, one can prove safety invariants on the program. Using our framework, we can furthermore check liveness invariants (e.g., loop detection) by inspecting the traces computed by the model checker. In [19], a particular type of attacks in the context of SDN networks has been modeled in Maude using the so-called hierachically structured composite actor systems described in [10]. This work does not provide a general model for SDN networks and, besides, barriers are not considered. On the other hand, it applies a statistical model checker, which requires to have a given scheduler for the messages. Such scheduler determines the exact order in which messages are handled while our framework captures all possible behaviours. Hence, both their aim and their SDN model are radically different from ours. As regards dynamic techniques, our work is mostly related to the model checkers NICE and Kuai for SDN programs, which have been compared in detail in Sec. 4.3. Our approach could be adapted to apply abstractions that bound the size of buffers [17] and to consider environment messages [21]. The approach of [11, 14] is fundamentally different from ours because it is based on analyzing dynamically given snapshots of the network from real executions. Instead, our approach tries to find programming errors by inspecting only the SDN program and considering all possible execution traces, thus enabling verification at system design time.

References

1. The ABS tool suite. `http://abs-models.org`.

2. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
3. Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In *CAV*, volume 10426, pages 526–543, 2017.
4. Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: a systematic testing tool for concurrent objects. In *CC*, pages 269–270, 2016.
5. Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: towards verifying controller programs in software-defined networks. In *PLDI*, pages 282–293, 2014.
6. Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable refinement checking for concurrent objects. In *POPL*, pages 651–662, 2015.
7. Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, and Jennifer Rexford. A NICE way to test openflow applications. In *NSDI*, pages 127–140, 2012.
8. Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *ICST*, pages 154–163, 2013.
9. Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A Complete Guide to the Future. In *ESOP*, volume 4421, pages 316–330, 2007.
10. Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Statistical model checking for composite actor systems. In *WADT*, pages 143–160, 2012.
11. Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. SDNRacer: concurrency analysis for software-defined networks. In *POPL*, pages 402–415, 2016.
12. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
13. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, pages 142–164, 2010.
14. Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.
15. Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Basset: a tool for systematic testing of actor programs. In *SIGSOFT FSE*, pages 363–364, 2010.
16. Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *POPL*, pages 385–399, 2016.
17. Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *FMCAD*, pages 163–170, 2014.
18. Openflow switch specification, October 2013. Version 1.4.0. `http://www.opennetworking.org/software-defined-standards/specifications`
19. Túlio A. Pascoal, Yuri Gil Dantas, Iguatemi E. Fonseca, and Vivek Nigam. Slow TCAM exhaustion ddos attack. In *SEC*, pages 17–31, 2017.
20. Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In *FASE*, pages 339–356, 2006.
21. Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *FMCAD*, pages 145–148, 2013.
22. Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *FMOODS/FORTE*, pages 219–234, 2012.

$$\frac{a(o,\bot,h,\mathcal{Q}) = selectAct(S), \text{TK}(tk,m,l,s) = selectTask(a(o,\bot,h,\mathcal{Q})), s \neq \epsilon, S \overset{o\cdot tk}{\rightsquigarrow^*} S'}{S \longrightarrow S'} \text{(MSTEP)}$$

$$\frac{\mathsf{tk} = \text{TK}(tk,m,l,\mathtt{x_f} = y\ !\ m_1(\bar{z}); s), o_1 = l(y), tk_1 = \text{fresh}(), l_1 = newlocals(\bar{z}, m_1, l)}{a(o,tk,h,\mathcal{Q} \cup \{\mathsf{tk}\}) \cdot a(o_1,tk',h',\mathcal{Q}') \rightsquigarrow a(o,tk,h,\mathcal{Q} \cup \{\text{TK}(tk,m,l[\mathtt{x_f} \mapsto t_1],s)\}) \cdot a(o_1,tk',h',\mathcal{Q}' \cup \{\text{TK}(tk_1,m_1,l_1,body(m_1))\})} \text{(ASY)}$$

$$\frac{\mathsf{tk} = \text{TK}(tk,m,l,x = \textbf{new}\ D(\bar{y}); s), o_1 = \text{fresh}(), \quad h' = newheap(D), l' = l[x \to o_1], \textbf{class}\ D(\bar{f})\{\ldots\}}{a(o,tk,h,\mathcal{Q} \cup \{\mathsf{tk}\}) \rightsquigarrow a(o,tk,h,\mathcal{Q} \cup \{\text{TK}(tk,m,l',s)\}) \cdot a(o_1,\bot,h'[\bar{f} \mapsto l(\bar{y})],\emptyset)} \text{(NEW)}$$

$$(\text{AWAIT})_1\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,\textbf{await}\ \mathtt{x_f}; s), l(\mathtt{x_f}) = tk_1,\ \text{TK}(tk_1,m_1,l_1,\epsilon) \in S}{a(o,tk,h,\mathcal{Q} \cup \{\mathsf{tk}\}) \rightsquigarrow a(o,tk,h,\mathcal{Q} \cup \{\text{TK}(tk,m,l,s)\})}$$

$$(\text{AWAIT})_2\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,\textbf{await}\ \mathtt{x_f}; s), l(\mathtt{x_f}) = tk_1,\ \text{TK}(tk_1,m_1,l_1,\epsilon) \notin S}{a(o,tk,h,\mathcal{Q} \cup \{\mathsf{tk}\}) \rightsquigarrow a(o,\bot,h,\mathcal{Q} \cup \{\text{TK}(tk,m,l,\textbf{await}\ \mathtt{x_f}; s)\})}$$

$$(\text{RETURN})\ \frac{\mathsf{tk} = \text{TK}(tk,m,l,\epsilon)}{a(o,tk,h,\mathcal{Q} \cup \{\mathsf{tk}\}) \rightsquigarrow a(o,\bot,h,\mathcal{Q} \cup \{\mathsf{tk}\})}$$

Fig. 6. Semantics of concurrent primitives of actor programs

A Semantics of Actor Language

The grammar below describes the syntax of our programs:

$$\begin{aligned} M &::= T\ m(\bar{T}\ \bar{x})\{s;\} \\ s &::= s\ ;\ s \mid x = e \mid \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ b\ \textbf{do}\ s \mid x.m(\bar{z}) \\ &\quad \mid x = \textbf{new}\ C(\bar{y}) \mid f = x!m(\bar{z}) \mid \textbf{await}\ f? \mid \textbf{await}\ b? \end{aligned}$$

where x, y, z denote variables names, f a future variable name, and s a sequence of instructions. For any entity A, the notation $\bar{A}$ is used as a shorthand for $A_1, ..., A_n$. We use the special actor identifier `this` to denote the current actor. For the sake of generality, the syntax of expressions e, boolean conditions b and types T is not specified. As in the object-oriented paradigm, a class denotes a type of actors including their behavior, and it is defined as a set of fields and methods. The "." in method calls, such as in $x.m(\bar{z})$, denotes standard (synchronous) method calls, while "!" is used for asynchrony.

Fig. 6 presents the semantics of the actor model. An *actor* is a term $a(o, tk, h, \mathcal{Q})$, where o is the actor identifier, tk is the identifier of the *active task* that holds the actor's lock or $\bot$ if the actor's lock is free, h is its local heap and $\mathcal{Q}$ is the set of tasks in the actor. A *heap* h is a mapping $h : fields(C) \mapsto \mathbb{V}$, where $\mathbb{V}$ stands for the set of references and values. A *task* tk is a term $\text{TK}(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to $\mathbb{V}$, and s is the sequence of instructions to be executed. As actors do not share their states, the semantics can be presented as a macro-step semantics [20] (defined by means of the transition "$\longrightarrow$") in which the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it gets to a *release point*, i.e., a point in which the actor's processor becomes idle due to the return or an await instruction. In this case,

rule (MSTEP) is applied to select an available task from an actor, namely function $selectAct(S)$ is applied to select *non-deterministically* an actor $a(o, \bot, h, \mathcal{Q})$ in the state with a non-empty queue $\mathcal{Q}$, and, $selectTask(a(o, \bot, h, \mathcal{Q}))$ to select *non-deterministically* a task of $\mathcal{Q}$. Micro-step transitions are written $\rightsquigarrow$ and define evaluations within a given macro-step. The sequential instructions are standard and thus omitted. In (NEW), an active task tk in actor o creates a new actor of class D with a fresh identifier $o_1 = \text{fresh}()$, which is introduced to the state with a free lock. Here $h' = newheap(D)$ stands for a default initialization on the fields of class D. Rule (ASY) spawns a new task (the initial state is created by *newlocals*) with a fresh task identifier tk_1 which is stored in the future variable $\mathtt{x_f}$. We assume $o \neq o_1$, but the case $o = o_1$ is analogous, the new task tk_1 is simply added to the queue $\mathcal{Q}'$ of actor o_1. In rule $(\text{AWAIT})_1$, the future variable $\mathtt{x_f}$ we are awaiting for points to a finished task and thus the **`await`** can be completed. The finished task identified with tk_1 is looked up in all actors in the current state (written as $\text{TK}(tk_1, m_1, l_1, \epsilon) \in S$). Otherwise, $(\text{AWAIT})_2$ yields the lock so that any other task of the same actor can take it. The behaviour of AWAIT on boolean conditions is analogous. When rule (RETURN) is executed, the task is *finished*, but it remains in the queue so that rules $(\text{AWAIT})_1$ and $(\text{AWAIT})_2$ can be applied. A derivation $E \equiv S_0 \longrightarrow \cdots \longrightarrow S_n$ is *complete* if S_0 is the initial state and all actors in S_n are of the form $a(o, \bot, h, \mathcal{Q})$, where for all $\mathsf{tk} \in \mathcal{Q}$ it holds that $\mathsf{tk} \equiv \text{TK}(tk, m, l, \epsilon)$.

B Soundness Proofs

Theorem 1. *Given an SDN network N, consider its SDN-Actor model N^a with an initial configuration* main *obtained by Def. 3.1, and the* Switch, Host *and* Controller *classes in Figs. 2 and 3. Then, for each execution in N, there exists an* equivalent *execution trace in N^a using the semantics of App. A.*

Proof. (sketch) The proof consists of two parts: (1) proving that the messages sent in any execution of the SDN network N have an equivalent state in N^a in which the corresponding actors have the same messages in their queues, and (2) proving that the state of the data structures of all network elements in N coincides with the state of the corresponding data structures in N^a. In order to prove (1), it is straightforward from the specification of the network elements by the OpenFlow that the implementation of the Switch, Host and Controller send the same messages. It remains to be proved that the messages in N^a are processed in all possible orders as determined by the OpenFlow specification, that state that they can be re-ordered. This is guaranteed by the rule (ASY) in the semantics (that inserts the message in the corresponding actor queue without any delay) and rule (MSTEP) that non-deterministically selects one message from the queue. Thus, any re-ordering that the network elements may do is captured. As regards point (2), the state of the data structures of the Switch, Host and Controller classes in Figs. 2 and 3 is modified by the processing of the messages. Thus, it directly follows from (1), since it is guaranteed that the same message processing is

captured by the model. A more formal proof would require having an OpenFlow formal semantics, which does not exist.

Theorem 2. *Methods* barrierRequest *and* barrierWait *provide a sound encoding of the* barrier *messages of OpenFlow.*

Proof. (sketch) The proof can be done by contradiction, assuming that the OpenFlow specification of barrier messages does not hold: either (1) a message sent after the barrier is processed before the barrier, or (2) a message sent before the barrier is processed after the barrier. On the one hand, since all the asynchronous (message) calls to the switches have already been done when the await is placed on the futures (of these calls), all execution orders (on the tasks) are still possible, and hence no feasible behaviour is lost. On the other hand, the properties of the barrier in OpenFlow are fulfilled since the awaits on the futures introduced by the barrierRequest method ensure that all previously received messages are processed (contradicting assumption 1) and the use of the barrierWait method ensures that no message is sent to the switch before the barrier request is finished, and hence no such message is processed (contradicting assumption 2). A detailed proof can be developed on traces (applying the semantics rules in App. A on the actor-based implementation of barriers) that lead to the above contradictions.