Actor-Based Model Checking for Software-Defined Networks

Elvira Albert[a,d], Miguel Gómez-Zamalloa[d], Miguel Isabel[e,*], Albert Rubio[a,d], Matteo Sammartino[b,c], Alexandra Silva[b]

[a] *Instituto de Tecnología del Conocimiento, Spain*
[b] *University College London, UK*
[c] *Royal Holloway, University of London, UK*
[d] *Universidad Complutense de Madrid, Spain*
[e] *Universidad Politécnica de Madrid, Spain*

Abstract

Software-Defined Networking (SDN) is a networking paradigm that has become increasingly popular in the last decade. The unprecedented control over the global behaviour of the network it provides opens a range of new opportunities for formal methods and much work has appeared in the last few years on providing bridges between SDN and verification. This article advances this research line and provides a link between SDN and traditional work on formal methods for verification of concurrent and 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 SDNs, including consistency of flow tables, violation of safety policies, and forwarding loops. Our model checker for SDNs is available through an online web interface, that also provides the SDN actor-models for a number of well-known SDN benchmarks.

Keywords: Software-Defined Networks, Verification, Concurrency, Actor-based modelling, Model checking

1. Introduction

SDN is a relatively recent networking paradigm which is now widely used in industry, with many companies—such as Google and Facebook—using SDN to control their backbone networks and data-centers. The core principle in SDN is

*Corresponding author: Miguel Isabel, Department of Sistemas Informáticos y Computación , C/ Profesor José García Santesmases, s/n Complutense University of Madrid, E-28040 - Madrid (Spain). Phone/Fax +34 91 3947641 / +34 91 3947529.

Email addresses: elvira@sip.ucm.es (Elvira Albert), mzamalloa@ucm.es (Miguel Gómez-Zamalloa), miguelis@ucm.es (Miguel Isabel), albert@cs.upc.edu (Albert Rubio), matteo.sammartino@rhul.ac.uk (Matteo Sammartino), alexandra.silva@ucl.ac.uk (Alexandra Silva)

the separation of 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 gained a substantial boost since SDN was introduced. In this new paradigm an unprecedented amount of detailed information about network events is centrally available. This facilitates checking for properties of the network behaviour, both statically and dynamically. Moreover, the controller itself is a program which can be analysed and verified before deployment.

The distributed and concurrent nature of network behaviour makes programming and verification tasks challenging. Some of the bugs that can be found in existing (programmable) networks are reminiscent of faults that have appeared in distributed and concurrent systems, and which have inspired much research in the verification and formal methods communities. With this observation as a starting point, this article provides a new bridge between SDN and a strand of formal methods—actor-based modelling [1]— which was originally developed to analyse concurrent systems. Actors, entities equipped with a private memory, form the basic unit of computation in this framework and can interact with each other through *asynchronous* messages. They are a natural formalism for capturing the various components of an SDN (hosts, controllers and switches), the messages they exchange, and the asynchronous actions triggered by those messages. This setup enables reasoning about local properties of the system 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*.

1.1. Summary of contributions

This article makes five main contributions:

1. SDN-semantics: A formalization of the semantics of SDNs which allows us to define the transitions that occur in the network and formalize the concept of execution trace needed to prove soundness of our modelling.

2. SDN-Actors: An encoding of all basic components of an SDN (switches, hosts, controller) into the actor-based language ABS [2] and a soundness proof of our encoding using the semantics of SDNs in point 1.

3. Barriers: One of the most challenging aspects to encode are the OpenFlow *barrier* messages [3], special instructions that the controller can use to force switches to execute all their queued tasks. We provide an implementation of barriers using conditional synchronization and a soundness result.

4. Model checker: A model checker for our SDN models built on top of the SYCO tool [4] that incorporates several Dynamic Partial-Order Reduction (DPOR) algorithms.

5. Case studies: 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, and scale to larger networks than related techniques.

This article extends and improves the conference paper that appeared in the FM'18 proceedings [5] as follows. On the theoretical side, we have formalized the semantics of SDNs and used it to prove soundness of the basic encoding of SDN-Actors, ensuring thus the correctness of our models. On the practical side, we have carried out a new experimental evaluation using the Constrained DPOR algorithm [6]. This DPOR algorithm can take advantage of independence conditions that we have defined specifically for the SDN domain and that allow us to treat larger networks than by using related techniques and than in our FM'18 paper. We have also extended the SYCO tool with a mechanism to detect the violation of the property under check that stops the exploration, while before SYCO was restricted to full exploration.

1.2. Organization of the article

Section 2 describes the basic concepts of the actor-based language ABS [2, 7] and the DPOR algorithm. Section 3 gives an intuition of the main ideas in the article by means of a simple example. In Section 4 we present the semantics of SDN programs and of actor systems, in two parts. First, Section 4.1 introduces a semantics for SDNs that describes the communication patterns in this kind of networks and that allows us to formalize the notion of execution trace in the SDN. Next, we recall the semantics of actor systems from [2] which will constitute the semantics of our models. Section 5 introduces the concept of *SDN-Actor* by providing the encoding of all components in an SDN as actors. We formally prove the soundness of the encoding by relying on the semantics introduced in Section 4.1. Section 6 extends our models to handle barriers and formalizes the soundness of this extension. Section 7 describes our DPOR-based model checker which instantiates an off-the-shelf model checker for actor systems with tailored independence conditions to efficiently verify SDN-Actor models. Section 8 describes the experimental evaluation of the tool. Related work and conclusions appear in Section 9.

2. Background

This section introduces the basic concepts and notations of the actor paradigm and the application of Dynamic Partial Order Reduction (DPOR) to it.

2.1. Actor-based concurrency model

The main idea in the actor-based paradigm [1] is that an actor represents a processor with a heap, a procedure stack, and an unordered queue of pending tasks. Initially, all processors are idle. When an idle processor's task queue is non-empty, some task is *non-deterministically* selected for execution. When a

task completes, its processor becomes idle again, chooses the next pending task, and so on. The heap is local, i.e., it cannot be accessed from other actors. The communication between actors is performed by means of *asynchronous* calls, and actors are executed in parallel. Besides reading and writing its own heap, each task can post tasks to the queues of any processor, including its own, and synchronize with the termination of tasks.

We use the object-oriented actor language ABS [2, 7], where each actor type is specified as a class, consisting of a set of fields and methods. The special method main does not belong to any actor and it is used to create the initial configuration of the system. In ABS, actors are instances of actor classes. For instance, the instruction A a = **new** A($\overline{x}$); creates an instance a of class A whose fields are initialized by arguments $\overline{x}$. The syntax **Fut**<**Unit**> f=a!m($\overline{x}$) spawns an asynchronous task m($\overline{x}$), that is added to the queue of pending tasks of a, where **Unit** means that 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* [8] that acts as a proxy for a result that remains unknown until the computation of its value by the spawned task is completed. The left-hand side of the assignment can be omitted in case the future variable is not needed. This language also allows standard (synchronous) method calls, written n($\overline{x}$), which are only allowed on the actor itself.

The language ABS also provides a convenient **await** primitive that will be used to model barriers. The instruction **await** f? synchronizes 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 and a notation table can be found in Section 4.2 and Appendix A, respectively.

Example 2.1. Consider the code in Figure 1. It contains a main method to create the actors involved in the execution of the program and the implementation of class A. The main task creates a new instance a of class A, whose field x is initialized to 0. After that, it spawns three tasks that are added to the queue of pending tasks to be executed in no particular order. Class A defines three methods: p, q and r. The first two methods write the field and the third one reads it.

Let us assume that our goal is to ensure that task r is executed only after the execution of task p. This can be achieved by means of a future variable and an **await** instruction. The resulting instruction would be **Fut**<**Unit**> f = a!p(); a!q(); **await** f?; a!r();. By doing this, task r will be only spawned at actor a if it has already completed the execution of task p as the future variable in the **await** must be ready.

2.2. DPOR-based model checking in actors

Model checking has been very successful to verify automatically and effectively some properties of concurrent systems. The principle is to explore all pos-

sible behaviours (states and transitions) of the system. However, state spaces increase exponentially with the number of concurrent processes. Partial Order Reduction (POR) techniques were proposed in the 90's to mitigate this state space explosion and their success was due to their ability of scaling to the verification of large applications. Dynamic POR [9] is a type of POR technique that 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. Two tasks are *independent* when changing their order of execution will not affect their combined effect, i.e., they commute and hence their reversed execution should not be explored. When DPOR is applied to actor systems, there are inherent reductions [10] because we can atomically execute each task (without re-orderings) until it finishes or an **await** instruction is found, as concurrency is non-preemptive and the active task cannot be interrupted. This avoids having to consider the interleavings at the level of instructions (as one must do in thread-based concurrency), and allows us to work at the level of tasks.

When considering DPOR, the most widely-used notion of independence is called *unconditional* independence, that is, if a pair of tasks is not independent in all possible executions, they are treated as potentially dependent and their interleavings are explored. This notion of independence is commonly over-approximated by requiring that actor fields accessed by one task are not modified by the other.

Example 2.2. Consider again the code in Figure 1. The execution tree to the right, including the dotted fragments, depicts the exploration of all possible behaviours from an initial state 0 in which there is a single task to execute the `main` method. After the execution of the `main` method, the pending tasks queue's actor `a` contains three tasks `p`, `q`, `r` (at state 1). The arrows are labelled with the name of the selected task from all available ones. As mentioned above an arbitrary task is selected at each step when there are several tasks available. Relying on the usual over-approximation of dependency all three pairs of tasks are dependent because they all access field `x` and always one of them writes on it. Therefore, the algorithm has to explore all possible permutations of the three tasks, hence 6 execution sequences.

Unnecessary exploration can be avoided using the notion of *conditional* independence. E.g., tasks `p` and `r` executing instructions `x = 5;` and `y = x;`, respectively, would be considered dependent even if `x = 5` — this is indeed an *independence constraint* (IC) for these two tasks, written $I_{p,r}$. Conditional independence was early introduced in the context of POR [11, 12]. The work in [13, 14] generated for the first time ICs for processes with a single instruction following some predefined patterns. Recently, Constrained DPOR [6] proposed to generate ICs in a pre-phase, using an SMT solver. It later used the generated ICs within DPOR to decide if two tasks are dependent and, thus, to identify

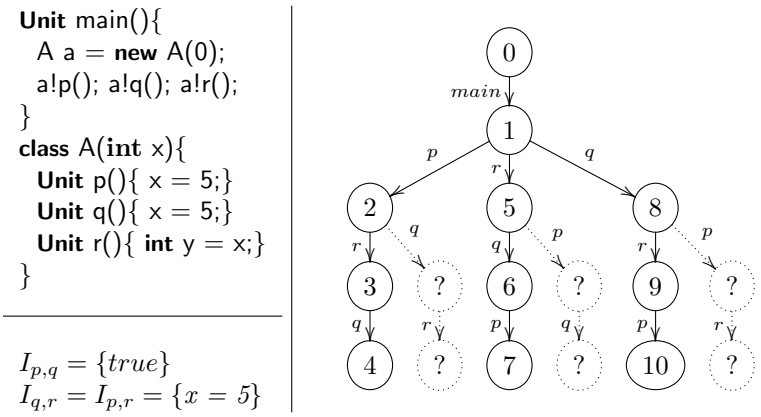

Figure 1: Left: Code of working example (up) and ICs (down). Right: Execution tree starting from $x = 0$. Full tree computed by DPOR, and dotted fragment not computed by Constrained DPOR.

backtracking points where the reversed order between such tasks must not be explored. ICs can be exploited in Constrained DPOR to achieve exponential pruning as follows.

Example 2.3. The SMT solver precomputes *true* as the IC for tasks p and q, since they commute in each state, they can be always considered as independent. On the other hand, both for tasks p and r, and for tasks q and r, the SMT solver computes x = 5 as their IC, since both pairs commute in states where the value of x is already 5. By taking advantage of these ICs (shown at the left bottom of the figure), Constrained DPOR is able to avoid the exploration of the dotted fragment of the tree. While the technical details of the Constrained DPOR algorithm are rather complex and are outside the scope of the paper, we would just like to intuitively explain two of the prunings: (1) the second branch is avoided due to $I_{q,r}$, as in state 2 we have that x = 5 (as p has already been executed) and hence $I_{q,r}$ evaluated to true, i.e., the tasks are independent and their reversed execution in the second branch is unnecessary, (2) the fourth branch is avoided due to $I_{p,q}$ as p and q are always independent and hence do not need to be reversed as in the fourth branch.

3. Overview

This section contains an overview of the technical contributions via an extended example.

3.1. Concurrency errors in SDNs

SDN is a networking architecture where a central software *controller* can dynamically change how network switches forward packets thanks to the information gathered by monitoring the traffic. Switches can be connected to hosts and to other switches via bidirectional channels that may reorder packets. Each

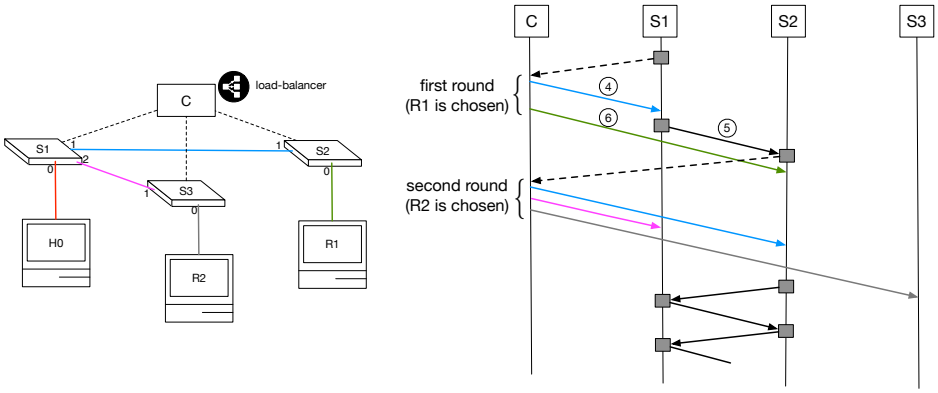

Figure 2: Example SDN load-balancer. On the left: structure of the SDN. On the right: messages exchanged in a possible execution of a naive controller program. Coloured arrows stand for control messages to switches, indicating which flow rule to install (colours specify the link to be used for the forwarding). Grey boxes and arrows among them represent packet forwardings. Dashed arrows indicate messages to the controller.

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 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 [15]) and how potential bugs can easily arise due to the concurrent behaviour 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 Figure 2, 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 Figure 2, 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, respectively. When the controller rules arrive at S1, it will have two contradictory instructions, telling it to forward the packet either to S2 or to S3. In the former case, 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.

3.2. Actor-based modelling of SDNs

We now explain how we can automatically detect the above problem using actors and model checking. Controller, hosts, and switches are modelled as actors. For instance, the instructions Controller ctrl = **new** Controller(); Switch s1 = **new** Switch("S1",ctrl); Host h0 = **new** Host("H0",s1,0); create 3 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 Figure 2 can be modelled using one actor per component (additional data structures for network links will be shown later).

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 Section 5):

$$\boxed{1\colon \mathsf{h0!sendIn}} \xrightarrow{1} \boxed{2\colon \mathsf{s1!switchHandlePacket}} \xrightarrow{2} \boxed{3\colon \mathsf{ctrl!controlHandleMessage}}$$
$$\xrightarrow{3} \boxed{4\colon \mathsf{s1!switchHandleMessage(s2)}, 5\colon \mathsf{s1!sendOut}, 6\colon \mathsf{s2!switchHandleMessage(r1)}}$$

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 Figure 2 with the same numbers. Their execution order is arbitrary: if it is the one shown in Figure 2, 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 SYCO via an exploration of a *reduced* execution tree, which avoids equivalent executions (Section 7).

4. Semantics for SDNs and for Actors

This section presents two semantics that provide the formal basis on which we build our models: we first introduce the semantics of SDNs in Section 4.1, and then the semantics of actors in Section 4.2. The semantics of actors has been already defined in several works (ours is a simplification of [2]). Our formalization of the SDN semantics is similar to that of [16]. We have considered a simplification of the Openflow specification that captures the essence of the communications of SDNs (e.g., we have not included the operation *flood* as it behaves similarly to the considered switch operations).

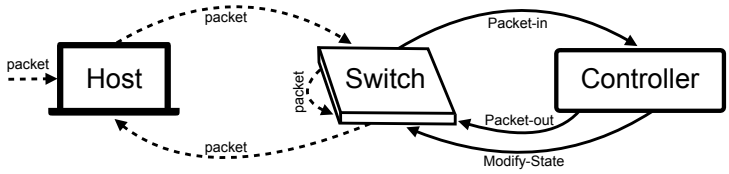

Figure 3: Information flow in SDNs

4.1. Software-Defined Networks

Let us first describe the information flow of packets and messages among the different elements in an SDN that we have depicted in Figure 3. As in standard networks, packets can be sent from hosts to switches and vice versa, and also from switches to switches (see dashed arrows). The leftmost dashed arrow represents the reception by a host of a new packet which is fed into the network. The specific communications of SDNs are performed by means of *Openflow messages* (see regular arrows), which in our simplification can be of three types:

- Packet-in: This message is sent from a switch to the controller when the switch processes a packet for which it has no action rule to apply. The packet is buffered in the switch until a message of type *Packet-out* is received. It is denoted as a term of the form $pktIn(sid, o, pid, ph)$ that includes the switch identifier sid, the receiving port o, the packet identifier pid and the packet header ph.
- Modify-State: This message is sent from the controller to a switch with new action rules to be inserted into the switch's flow-table. It is denoted as a term of the form $modState(a)$ that includes an action rule a.
- Packet-out: This message is sent from the controller to a switch to notify that it must re-try applying an action rule to a buffered packet. It is denoted as a term of the form $pktOut(ph)$ that includes the header ph of the packet to be delivered.

Figure 4 shows the semantics of the flow of communications performed in our simplified SDNs.

- A host is a term of the form $h(id, sid, o, inp)$, where id is the host identifier, sid and o are, respectively, the switch identifier and port to which the host is connected, and inp its input channel.
- A switch is of the form $s(id, ft, b, inp)$, where id is the switch identifier, ft its flow-table, b its internal buffer of packets and inp its input channel.
- The controller is of the form $c(top, inp)$ where top is the topology of the network and inp is its input channel.

- A state of the SDN is a tuple of the form $\langle H, S, C\rangle$ where $H = \{h \mid h \text{ is a host}\}$, $S = \{s \mid s \text{ is a switch}\}$, that is, H is a set of hosts, S is a set of switches, and, C is the controller.

Letter p denotes a packet. Packets in the SDN paradigm are deliberately abstracted. They can be whatever as long as there exist functions $header(p)$ and $id(p)$, that return, respectively, the header and the id of packet p. Flow-tables are represented as mappings from packet header/port pairs to actions and are treated as a black-box through the following functions: $lookup(ft, \langle ph, o\rangle)$ that returns the action associated to the packet with header ph received through port o in the flow-table ft, or $\perp$ if there is no entry for it; and, $put(ft, \langle ph, o\rangle, a)$ that returns the new flow-table after inserting in ft the entry $\langle ph, o\rangle \mapsto a$. For simplicity, we only consider actions of the form $send(id)$ or $\langle send(id), o\rangle$, which indicate that the corresponding packet should be sent, respectively, to the host id, or to the switch with id as identifier using port o. Function $applyPol(top, sid, o, ph)$ represents the application of the controller's policy using the current network topology top in response to a packet received via port o with header ph that the switch with identifier sid has not been able to handle. It returns a set of pairs $\langle id, m\rangle$ where m is a modifyState message with an associated new flow-table entry that has to be forwarded to the switch with identifier id.

A *transition* or *step* in the network corresponds to the processing of a packet or message by a host, switch or the controller. There are six (sets of) transition rules corresponding to the different types of incoming arrows in Figure 3:

- sendIn (abbreviated as SI): This corresponds to a host dispatching (injecting) a new packet which is fed into the network (denoted as $new(p)$), in which case the packet is forwarded to the switch to which the host is connected via the corresponding port. Note that the port is attached to the packet (denoted $o{:}p$) since there is only one input channel in switches.

- hostHandlePacket (HHP): This corresponds to the processing by a host h of a packet received from its switch, in which case the packet is consumed without any further action.

- switchHandlePacket (SHP): When a switch processes a received packet, either from a host or from another switch, it looks up if there is any rule matching with the header of the packet and port in its flow table ft. There are three cases: (cases 1 and 2) there is a send action rule in the switch's flow-table, hence the packet is forwarded to the host (case 1) or switch (case 2) indicated in the action (in the latter case also the switch's port is included in the action); or (case 3) there is no rule for this packet, in which case the packet is buffered and a Packet-in message is sent to the controller.

- sendOut (SO): This corresponds to the processing of a Packet-out message by a switch. After looking up the header of the packet p in its own flow-table ft, there are three cases which are analogous to those of switchHandlePacket except that the packet is in the switch's buffer (instead of

in its input channel), and that if no action rule is found in the switch's flow-table the packet is dropped.

- switchHandleMessage (SHM): It corresponds to the processing of a Modify-State message by a switch, in which case the received action rule is inserted into the switch's flow-table.

- controlHandleMessage (CHM): The controller receives a Packet-in message from a switch s in response to a packet that the switch s has not been able to handle. As a result, the controller sends a set ms of Modify-State messages with new action rules to a selected set of switches (as specified by the controller's policy with the current network topology), and a Packet-out message to switch s.

An execution trace $E \equiv S_0 \rightarrow \cdots \rightarrow S_n$ is *complete* if S_0 is the initial state and $S_n = \langle H, S, C \rangle$ is the final state such that every message and packet in their channels has been processed (their input channels are empty). We use $exec(S)$ to denote the set of all possible executions starting at state S.

4.2. Syntax and Semantics for Actor Programs

The grammar below describes the syntax of the language ABS in which SDN models will be defined:

$$
\begin{array}{lcl}
P & ::= & M\ \bar{C} \\
C & ::= & \textbf{class}\ c(\overline{T\ x})\{\bar{M}\} \\
M & ::= & T\ m(\overline{T\ x})\{s;\} \\
s & ::= & s\ ;\ s \mid x = e \mid \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ b\ \textbf{do}\ s \mid m(\bar{z}) \\
 & & \mid x = \textbf{new}\ C(\bar{y}) \mid f = x!m(\bar{z}) \mid \textbf{await}\ f? \mid \textbf{await}\ b?
\end{array}
$$

Here, 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 identifier **this** to denote the current actor. For generality, the syntax of expressions e, Boolean conditions b and types T is left unspecified. As in the object-oriented paradigm, a class denotes a type of actors including their behaviour, and it is defined as a set of fields and methods. Lastly, $m(\bar{z})$ denotes standard (synchronous) method calls, which are only allowed on the actor itself, whereas "!" is used for asynchronous method calls (see Section 2.1).

Figure 5 presents the semantics of the actor model. An *actor* is a term of the form $a(o, tk, h, \mathcal{Q})$, where o is the actor identifier, tk is the identifier of the *active task* that holds the actor's lock or $\perp$ if the actor's lock is free, h is its local heap and $\mathcal{Q}$ is the queue of tasks in the actor. A *heap* h is a mapping $h : \mathit{fields}(C) \mapsto \mathbb{V}$, where $\mathbb{V}$ stands for the set of references and values. A *task* τ 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. Finally, a global state S is a set of actors. As actors do not share their states, the semantics can be presented as a macro-step semantics [17] (defined by means of the transition

$$(\text{SI})\ \frac{h = h(id, sid, o, inp \cup \{new(p)\}) \quad s = s(sid, ft, b, inp')}{\langle \{h\} \cup H, \{s\} \cup S, C\rangle \to \langle \{h(id, sid, o, inp)\} \cup H, \{s(sid, ft, b, inp' \cup \{o{:}p\})\} \cup S, C\rangle}$$

$$(\text{HHP})\ \frac{h = h(id, sid, o, inp \cup \{p\})}{\langle \{h\} \cup H, S, C\rangle \to \langle \{h(id, sid, o, inp)\} \cup H, S, C\rangle}$$

$$(\text{SHP}_1)\ \frac{\begin{array}{c} s = s(sid, ft, b, inp \cup \{o{:}p\}) \quad h = h(id, sid, o', inp') \\ send(id) = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle \{h\} \cup H, \{s\} \cup S, C\rangle \to \langle \{h(id, sid, o', inp' \cup \{p\})\} \cup H, \{s(sid, ft, b, inp)\} \cup S, C\rangle}$$

$$(\text{SHP}_2)\ \frac{\begin{array}{c} s = s(sid, ft, b, inp \cup \{o{:}p\}) \quad s' = s(sid', ft', b', inp') \\ send(sid', o') = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle H, \{s, s'\} \cup S, C\rangle \to \langle H, \{s(sid, ft, b, inp), s(sid', ft', b', inp' \cup \{o'{:}p\})\} \cup S, C\rangle}$$

$$(\text{SHP}_3)\ \frac{s{=}s(sid, ft, b, inp \cup \{o{:}p\}) \quad s' = s(sid, ft, b \cup \{o{:}p\}, inp) \quad \bot = lookup(ft, \langle header(p), o\rangle)}{\langle H, \{s\} \cup S, c(top, inp')\rangle \to \langle H, \{s'\} \cup S, c(top, inp' \cup \{\mathsf{pktIn(sid, o, id(p), header(p))}\})\rangle}$$

$$(\text{SO}_1)\ \frac{\begin{array}{c} s = s(sid, ft, b \cup \{o{:}p\}, inp \cup \{\mathsf{pktOut(ph)}\}) \\ ph = header(p) \quad h = h(id, sid, o', inp') \quad send(id) = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle \{h\} \cup H, \{s\} \cup S, C\rangle \to \langle \{h(id, sid, o, inp' \cup \{p\})\} \cup H, \{s(sid, ft, b, inp)\} \cup S, C\rangle}$$

$$(\text{SO}_2)\ \frac{\begin{array}{c} s = s(sid, ft, b \cup \{o{:}p\}, inp \cup \{\mathsf{pktOut(ph)}\}) \\ ph = header(p) \quad s' = s(sid', ft', b', inp') \quad send(sid', o') = lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle H, \{s, s'\} \cup S, C\rangle \to \langle H, \{s(sid, ft, b, inp), s(sid', ft', b', inp' \cup \{o'{:}p\})\} \cup S, C\rangle}$$

$$(\text{SO}_3)\ \frac{\begin{array}{c} s = s(sid, ft, b \cup \{o{:}p\}, inp \cup \{\mathsf{pktOut(ph)}\}) \\ ph = header(p) \quad \bot{=}lookup(ft, \langle header(p), o\rangle) \end{array}}{\langle H, \{s\} \cup S, C\rangle \to \langle H, \{s(sid, ft, b, inp)\} \cup S, C\rangle}$$

$$(\text{SHM})\ \frac{s = s(sid, ft, b, inp \cup \{\mathsf{modState}(\langle \mathsf{ph, o}\rangle \mapsto \mathsf{a})\})}{\langle H, \{s\} \cup S, C\rangle \to \langle H, \{s(sid, put(ft, \langle ph, o\rangle, a), b, inp)\} \cup S, C\rangle}$$

$$(\text{CHM})\ \frac{\begin{array}{c} c = c(top, cinp \cup \{\mathsf{pktIn(sid, o, pid, ph)}\}) \quad s = s(sid, ft, b, sinp) \\ ms = applyPol(top, sid, o, ph) \quad ms_{id} = \{m \mid \langle id, m\rangle \in ms\} \\ S' = \{s(sid', ft', b', inp') \mid s(sid', ft', b', inp) \in S, inp' = inp \cup ms_{sid'}\} \end{array}}{\langle H, S \cup \{s\}, c\rangle \to \langle H, S' \cup s(sid, ft, b, sinp \cup ms_{sid} \cup \{\mathsf{pktOut(ph)}\}), c(top, cinp)\rangle}$$

Figure 4: Semantics of SDNs

$$(\textsc{mstep})\ \frac{a(o,\bot,h,\mathcal{Q}) = selectAct(S) \qquad \textsc{tk}(tk,m,l,s) = selectTask(a(o,\bot,h,\mathcal{Q})) \quad s \neq \epsilon \quad S \overset{o\cdot tk}{\rightsquigarrow^*} S'}{S \longmapsto S'}$$

$$(\textsc{asy})\ \frac{\tau = \textsc{tk}(tk,m,l,\mathtt{x_f} = y\ !\ m_1(\bar{z});s) \quad o_1 = l(y) \quad tk_1 = \text{fresh}() \quad l_1{=}newlocals(\bar{z},m_1,l)}{\begin{array}{c} a(o,tk,h,\mathcal{Q}\cup\{\tau\})\cdot a(o_1,tk',h',\mathcal{Q}') \overset{o\cdot tk}{\rightsquigarrow} \\ a(o,tk,h,\mathcal{Q}\cup\{\textsc{tk}(tk,m,l[\mathtt{x_f}\mapsto tk_1],s)\})\cdot a(o_1,tk',h',\mathcal{Q}'\cup\{\textsc{tk}(tk_1,m_1,l_1,body(m_1))\}) \end{array}}$$

$$(\textsc{syn})\ \frac{\tau = \textsc{tk}(tk,m,l,m_1(\bar{z});s) \quad l_1{=}newlocals(\bar{z},m_1,l)}{a(o,tk \quad h,\mathcal{Q}\cup\{\tau\})\cdot \overset{o\cdot tk}{\rightsquigarrow} a(o,tk,h,\mathcal{Q}\cup\{\textsc{tk}(tk,m,l_1,body(m_1);s)\})}$$

$$(\textsc{new})\ \frac{\begin{array}{c}\tau = \textsc{tk}(tk,m,l,x = \mathbf{new}\ D(\bar{y});s) \quad o_1{=}\text{fresh}() \\ h' = newheap(D) \quad l' = l[x\to o_1] \quad \mathbf{class}\ D(\bar{f})\{\ldots\}\end{array}}{a(o,tk,h,\mathcal{Q}\cup\{\tau\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,tk,h,\mathcal{Q}\cup\{\textsc{tk}(tk,m,l',s)\})\cdot a(o_1,\bot,h'[\bar{f}\mapsto l(\bar{y})],\emptyset)}$$

$$(\textsc{await})_1\ \frac{\tau = \textsc{tk}(tk,m,l,\mathbf{await}\ \mathtt{x_f};s) \quad l(\mathtt{x_f}) = tk_1 \quad \textsc{tk}(tk_1,m_1,l_1,\epsilon)\in S}{a(o,tk,h,\mathcal{Q}\cup\{\tau\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,tk,h,\mathcal{Q}\cup\{\textsc{tk}(tk,m,l,s)\})}$$

$$(\textsc{await})_2\ \frac{\tau = \textsc{tk}(tk,m,l,\mathbf{await}\ \mathtt{x_f};s) \quad l(\mathtt{x_f}) = tk_1 \quad \textsc{tk}(tk_1,m_1,l_1,\epsilon)\notin S}{a(o,tk,h,\mathcal{Q}\cup\{\tau\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,\bot,h,\mathcal{Q}\cup\{\textsc{tk}(tk,m,l,\mathbf{await}\ \mathtt{x_f};s)\})}$$

$$(\textsc{return})\ \frac{\tau = \textsc{tk}(tk,m,l,\epsilon)}{a(o,tk,h,\mathcal{Q}\cup\{\tau\}) \overset{o\cdot tk}{\rightsquigarrow} a(o,\bot,h,\mathcal{Q}\cup\{\tau\})}$$

Figure 5: Semantics of concurrent primitives of actor programs

"$\longmapsto$") 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 end of the current task or an **await** instruction. In this case, rule (MSTEP) is applied to select an available task from an actor, namely relation $selectAct(S)$ 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 $\overset{o\cdot tk}{\rightsquigarrow}$ and define evaluations in task tk by actor o within a given macro-step. As before, 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 (SYN) simply replaces in task tk the statement with the method call to m_1 by its body. 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. An execution trace $E \equiv S_0 \longmapsto \cdots \longmapsto S_n$ 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 $\tau \in \mathcal{Q}$ it holds that $\tau \equiv \text{TK}(tk, m, l, \epsilon)$. We use $exec(S)$ to denote the set of all possible executions starting at state S.

5. SDN-Actors: an actor based encoding of SDN programs

We present the concept of *SDN-Actor* in 3 steps: Section 5.1 describes the creation and initialization of the actors according to the topology. Section 5.2 provides the encoding of the operations and communication for Switch and Host actors. Section 5.3 proposes the encoding of the controller. Altogether, our encoding provides an actor-based semantics foundation of SDNs that follow the OpenFlow specification [3] captured by the semantics in Section 4.1.

5.1. Network topology

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

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

from which we automatically generate the initial configuration as follows.

Definition 5.1 (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* init_conf*) 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, respectively, 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 asynchronous call* **Fut**<**Unit**> f = ctrl!addConfig(srefs,hrefs,ntw) *initializes in the controller the topology relations and the references to switches and hosts such that the controller can send control messages to redirect the traffic to the involved links. Future variable* f *is used to ensure that the controller is correctly initialized before the* init_conf *method finishes.*

Example 5.2. By applying Definition 5.1 to the topology in Figure 2, given as the relation: $SHlink(S1, H0, 0)$, $SHlink(S2, R1, 0)$, $SHlink(S3, R2, 0)$, $SSlink(S1, 1, S2, 1)$, and $SSlink(S1, 2, S3, 1)$, we obtain the following initial configuration which constitutes the init_conf method from which the execution starts:

```
1 init_conf() { 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         Fut<Unit> f = ctrl!addConfig(srefs,hrefs,ntw);
9         await f?;}
```

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, take, lookup, etc.) that will appear in italics in the code.

5.2. The switch and host classes

Figure 6 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 implemented in more detail to make explicit the information they contain:

- the buffer at Line 23 (L23 for short) is a *map* that must contain pairs of packet and input port indexed by their PacketId.
- the flow table flowT (L22) is implemented as a *map* indexed by the so-called *match field* [3] represented by type MatchF in Figure 6. 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 $isHost$ succeeds if the action is of type i), and function $isSwitch$ if it is of type ii). Functions $getSwitch$, $getHost$ and $getPort$ return the s, h and o, respectively. The full implementation must allow duplicate entries (non-deterministically selected), and the use

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

```
class Host(HostId hid, Switch s, PortId o) {
 Unit sendIn(Packet p){
    s!switchHandlePacket(p,o);
 }
 Unit hostHandlePacket(Packet p){
    /* output packet */
 }
}
class Switch(SwitchId sid, Controller ctrl) {
   Map<MatchF,Action> flowT={};
   Map<PacketId,(Packet,PortId)> buffer={};
 Unit switchHandlePacket(Packet p, PortId o){
    Action l=lookup(flowT,(getHeader(p),o));
    if (isSwitch(l))
        getSwitch(l)!switchHandlePacket(p,getPort(l));
    else if (isHost(l))
        getHost(l)!hostHandlePacket(p);
        else {
        buffer=put(buffer,getId(p),(p,o));
        ctrl!controlHandleMessage(sid,o,getId(p),getHeader(p));
    }
 }
 Unit sendOut(PacketId pi){
    Packet p; PortId o;
   (p,o)=take(buffer,pi);
    Action l=lookup(flowT,(getHeader(p),o));
    if (isSwitch(l))
        getSwitch(l)!switchHandlePacket(p,getPort(l));
    else if (isHost(l))
        getHost(l)!hostHandlePacket(p);
    /* else packet is dropped */
 }
 Unit switchHandleMessage(MatchF m, Action a){
   flowT=put(flowT,m,a);
 }
}
```

Figure 6: Type declarations (top) and actor-based host and switch classes (bottom)

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. Upon creation switches receive 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 L25, and, if there is an entry for the packet in the table, it asynchronously makes the corresponding action (either send it to a host L28 or to a switch L26). Otherwise, it sends a controlHandleMessage request and puts the packet and input port in the buffer (L31 and L32) 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 (L36), 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 Section 8.

Example 5.3. In init_conf, after L9, 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 init_conf spawns. Its execution in turn spawns a new task s1!switchHandlePacket (p,0) at L14, that does not find an entry in flowT at L25. Then, it spawns a controlHandleMessage task on the controller at L32, whose code is presented in the next section.

5.3. The controller

After creating the controller actor, the method addConfig is invoked asynchronously to initialize the references to switches and hosts and set up the initial network topology (see L8). The **await** instruction at L9 ensures that the controller is always initialized before the termination of the init_conf method. A simple controller is presented in Figure 7. When a switch asynchronously invokes controlHandleMessage, the controller applies the current policy—function *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 Section 8). When applying the policy for a given SwitchId, PortId and PacketH, we obtain a list of

```
49 class Controller() {
50   Map<SwitchId,Switch> srefs={};
51   Map<HostId,Host> href={};
52   List<Link> ntw=[];
53   Unit addConfig(Map<SwitchId,Switch> sr, Map<HostId,Host> hr, List<Link> n){
54        /* references to switches and hosts and network topology initialized */
55   }
56   Unit controlHandleMessage(SwitchId sid, PortId o, PacketId p, PacketH h){
57       List<(SwitchId,MatchF,Action)> l=applyPolicy(sid,o,h);
58       while (not(isEmpty(l))) {
59            SwitchId s1; Action a1; MatchF m1;
60           (s1,m1,a1)=head(l);
61            lookup(srefs,s1)!switchHandleMessage(m1,a1);
62            l=tail(l);
63       }
64        lookup(srefs,sid)!sendOut(p);
65    }
66  }
```

Figure 7: Controller class (without barriers)

switch identifiers and corresponding actions to be applied to them (as a data-structure of type $List$<(SwitchId,MatchF,Action)>). The **while** loop at L58 in controlHandleMessage asynchronously invokes switchHandleMessage at L83 on each of the switches in the list, and passes as parameter the corresponding action to be applied for the given match entry. Finally, it notifies at L64 the switch from which the packet came that this can be sent out. More sophisticated controllers that build upon this encoding are described in Section 8.

Example 5.4. In the example, applyPolicy corresponds to the load-balancer described in Section 3, 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 L57 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 action 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 L64. Several problems may arise in this implementation. One problem, as explained in Section 3, 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 Figure 2.

5.4. Soundness of the Encoding

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. First of all, let us define the equivalence between an input channel with its buffer (inp, b) and a queue of pending tasks with its buffer $(\mathcal{Q}, \text{buf})$. Let us notice here that even though we have used different notation for b and buf, we use $b = \text{buf}$ to denote the equality of information in both structures, that is, they have exactly the same packets and with the same ports.

Proposition 5.5. *Let inp be an input channel, b be its buffer of pending packets, $\mathcal{Q}$ be a queue of pending tasks and buf be its buffer of pending tasks. Then, the channel inp and its buffer b is equivalent to the queue $\mathcal{Q}$ and its buffer buf, written $(inp, b) \equiv (\mathcal{Q}, buf)$, if and only if:*

1. *$inp = \emptyset = \mathcal{Q}$ and $b = buf$ or*
2. *otherwise one of the following holds:*

   pktOut: *$inp = \{\mathsf{pktOut(ph)}\} \cup inp'$, $\exists$ TK, $\exists$ p such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, \mathsf{sendOut}, l, \_)\}$, $b = b' \cup \{o : p\}$, $buf = buf' \cup \{(p, o)\}$, $getId(p) = l[\mathsf{pi}]$ and $getHeader(p) = \mathsf{ph}$, and $(inp', b') \equiv (\mathcal{Q}', buf')$.*

   modState: *$inp = \{\mathsf{modState}(\langle \mathsf{ph,o} \rangle \mapsto \mathsf{a})\} \cup inp', \exists$ TK such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, \mathsf{switchHandleMessage}, l, \_)\}$, $(\langle \mathsf{ph,o} \rangle \mapsto a) = (l[\mathsf{m}] \mapsto l[\mathsf{a}])$ and $(inp', b) \equiv (\mathcal{Q}', buf)$,*

   pktIn: *$inp = \{\mathsf{pktIn(sid,o,pid,ph)}\} \cup inp'$, $\exists$ TK such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, \mathsf{controlHandleMessage}, l, \_)\}$, $sid = l[\mathsf{sid}], pid = l[\mathsf{p}], ph = l[\mathsf{h}]$, $o = l[\mathsf{o}]$ and $(inp', b) \equiv (\mathcal{Q}', buf)$.*

   packet: *$inp = \{o : p\} \cup inp'$, $\exists$ TK such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, \mathsf{switchHandlePacket}, l, \_)\}$, $o = l[\mathsf{o}], p = l[\mathsf{p}]$ and $(inp', b) \equiv (\mathcal{Q}', buf)$.*

   packet-out: *$inp = \{p\} \cup inp'$, $\exists$ TK such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, \mathsf{hostHandlePacket}, l, \_)\}$, $p = l[\mathsf{p}]$, and $(inp', b) \equiv (\mathcal{Q}', buf)$.*

   packet-in: *$inp = \{new(p)\} \cup inp'$, $\exists$ TK such that $\mathcal{Q} = \mathcal{Q}' \cup \{\text{TK}(\_, \mathsf{sendIn}, l, \_)\}$, $p = l[\mathsf{p}]$, and $(inp', b) \equiv (\mathcal{Q}', buf)$.*

Let a be a set and $t \in a$, then a' denotes the set that satisfies $a = \{t\} \cup a'$. Now, we can define the equivalence between an SDN state and an SDN-Actor state.

Definition 5.6 (equivalence). *An SDN state $S = \langle H, Sw, C \rangle$ and an SDN-actor state S_a are equivalent, written $S \equiv S_a$, if and only if:*

Host: *$\forall h(id, sid, o, inp) \in H, \exists! a(\_, \_, h, \mathcal{Q}) \in S_a$ such that $(inp, \emptyset) \equiv (\mathcal{Q}, \emptyset)$, $id = h[hid], sid = h[s]$, and $o = h[o]$.*

Switch: *$\forall s(id, ft, b, inp) \in Sw, \exists! a(\_, \_, h, \mathcal{Q}) \in S_a$ such that $(inp, b) \equiv (\mathcal{Q}, h[buf])$, $id = h[sid]$, and $ft = h[flowT]$.*

Controller: $C = c(top, cinp)$ *and* $\exists!a(id, \_, h, \mathcal{Q}) \in S_a$ *such that* $(cinp, \emptyset) \equiv (\mathcal{Q}, \emptyset)$, $related(top, \{h[srefs], h[href], h[ntw]\})$, *and* $\forall a(\_, \_, h', \_) \in S_a$, $id = h'[ctrl]$.

Let us notice here that we use $related(top, \{h[srefs], h[href], h[ntw]\})$ to clarify that information about the topology is coherent in both the controller and the controller actor.

The following theorem ensures the soundness of our modelling. Essentially we guarantee that, for a given SDN that follows the OpenFlow specification, any execution in the network has an equivalent execution in the SDN-Actor model. The proof can be found in Appendix B. We denote as S_a^{ini} the SDN-Actor state defined in Definition 5.1, i.e., after executing method init_conf() and all asynchronous calls to method sendIn containing the packets to be delivered. Furthermore, $S^{ini} \equiv S_a^{ini}$.

Theorem 5.7. *Let* S^{ini} *and* S_a^{ini} *be an SDN state and an SDN-Actor state, respectively.*

1. *For every execution* $S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \in exec(S^{ini}), \exists S_a^{ini} \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini})$ *such that* $S^n \equiv S_a^n$.
2. *For every execution* $S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini}), \exists S^{ini} \rightarrow ... \rightarrow S^n \in exec(S^{ini})$ *such that* $S^n \equiv S_a^n$.

6. Implementing barriers using conditional synchronization

Barriers [3] have been designed to force a switch to handle previous control messages, and thus avoid problems such as the one described in Example 5.4. Following OpenFlow, upon receipt of a *barrier message*, the switch must finish processing all previously-received controller messages, before executing any messages received after the *barrier message*.

Figure 8 shows our modelling 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 L83 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 L83 adds the future variable to the list indexed by s1 in the map).
3. The controller keeps in barrierOn the set of switches with an active barrier.
4. A barrier on a switch consists in the controller waiting on the list of future variables that the switch needs to acknowledge to ensure that its control messages have already been processed (method barrierRequest).

```
67  class Controller() {
68    Map<SwitchId,Switch> srefs={};
69    Map<HostId,Host> href={};
70    List<Link> ntw=[];
71    Map<SwitchId,List<Fut<Unit>> barrierMap={};
72    Set<SwitchId> barrierOn = ∅;
73    Unit addConfig(Map<SwitchId,Switch> sr, Map<HostId,Host> hr, List<Link> n){
74          /* references to switches and hosts and network topology initialized */
75    }
76    Unit controlHandleMessage(SwitchId sid, PortId o, PacketId p, PacketH h){
77        List<(SwitchId,MatchF,Action)> l=applyPolicy(sid,o,h);
78        List<SwitchId> ls = [];
79        while (not(isEmpty(l))) {
80            SwitchId s1; Action a1; MatchF m1;
81            (s1,m1,a1)=head(l);
82            barrierWait(s1);
83            Fut<Unit>f=lookup(srefs,s1)!switchHandleMessage(m1,a1);
84            barrierMap=putAdd(barrierMap,s1,f);
85            ls = add(ls,s1);
86            l=tail(l);
87        }
88        while(not(isEmpty(ls))) {
89            barrierWait(head(ls));
90            barrierRequest(head(ls));
91            ls=tail(ls);
92        }
93        barrierWait(sid);
94        Fut<Unit>f=lookup(srefs,sid)!sendOut(p);
95        barrierMap=putAdd(barrierMap,sid,f);
96    }
97    Unit barrierWait (SwitchId sid){
98          await not(contains(barrierOn,sid))?;
99    }
100   Unit barrierRequest (SwitchId sid){
101       barrierOn=add(barrierOn,sid);
102       List<Fut<Unit>> futSid=take(barrierMap,sid);
103       while (not(isEmpty(futSid)) {
104             Fut<Unit> fi=head(futSid);
105             await fi?;
106             futSid=tail(futSid);
107       }
108      barrierOn=delete(barrierOn,sid);
109   }
110 }
```

Figure 8: Extension of Controller class with barriers

5. All control messages must be now preceded by a call to barrierWait that checks if the corresponding switch has an active barrier, L98. 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. Similarly, the call to barrierRequest must also be preceded by a call to barrierWait since barrierRequest is indeed modelling the send to the switch of a control message (the *barrier message*).

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 conditional synchronization (by means of the **await** instructions) that ensures the behaviour of OpenFlow barriers.

The next theorem states that our implementation of barriers via methods barrierRequest and barrierWait provide a sound encoding of the OF *barrier*.

Theorem 6.1 (soundness of barriers). *Given a state S in an execution of the SDN-Actor model right before executing L90 with switch sid as parameter (i.e., the state before activating a barrier over sid), and the state S' right before executing L91 (i.e., the state after receiving the acknowledgement of the barrier), the following holds:*

- *All switchHandleMessage and sendOut tasks in the queue of switch sid in state S have been completely executed in state S'.*
- *No switchHandleMessage nor sendOut task have been spawned over switch sid in any middle state between S and S'.*
- *No other barrierRequest call for switch sid is performed between S and S'.*

Proof.
Let us firstly define an invariant which holds for every possible state S'' of any execution of the SDN-Actor model:

$$\forall a(sid, \_, \_, \mathcal{Q}) \in S'' \text{ and } \forall \text{TK}(tk, m, \_, \_) \in \mathcal{Q},\ m \in \{\mathsf{switchHandleMessage}, \mathsf{sendOut}\}$$

$$\exists! a(cid, \_, h, \_) \in S'' \text{ such that } tk \in h[\mathsf{barrierMap}][\mathsf{sid}]$$

The invariant states that every spawned switchHandleMessage or sendOut task tk on a switch sid is recorded by means of a future variable in the list associated to sid in the barrierMap field of the controller (i.e. $tk \in h[\mathsf{barrierMap}][\mathsf{sid}]$). Note the abuse of notation $\in$ to check existence of an element in a *List* data-structure, and [] to access the value of a key in a *Map* data-structure. It can be seen that after making any asynchronous call to method switchHandleMessage (L83) or sendOut (L94), the corresponding future variable is always recorded in the barrierMap field (L84 and L95).

Now, given the controller of the state S, $a(cid, \_, h_c, \mathcal{Q}_c) \in S$, for every task $\text{TK}(tid, \mathsf{controlHandleMessage}, l_c, \mathsf{barrierRequest(l)}; s) \in \mathcal{Q}_c$, we have an execution

trace $S = S_0 \overset{cid.tid}{\rightsquigarrow^*} S_1 \longmapsto ... \longmapsto S_n \overset{cid.tid}{\longmapsto} S_{n+1} = S'$ such that S_1 is the global state after executing the micro-step transitions of such task until it stops at L105, and S_{n+1} is the first state where $h_c[\mathsf{barrierOn}]$ does not contain the switch $sw = l[\mathsf{sid}]$. Let us notice that if such stop is not performed, then every task in sw has already finished and barrierRequest is performed in a single macro-step ($S_0 = S_n$). Then, we know that $\forall i \in \{0, ..., n+1\}, \exists a(sw, \_, \_, \mathcal{Q}_i) \in S_i$ with $\mathcal{Q}_i = SHP_i \cup SO_i \cup SHM_i$ such that:

- SHP_i contains all switchHandlePacket tasks,
- SO_i contains all sendOut tasks, and,
- SHM_i contains all switchHandleMessage tasks.

By the definitions of the states S_1 and S_{n+1}, we know that $\forall i \in \{1, ..., n\}$, $sw \in h_c[\mathsf{barrierOn}]$. Hence, $\forall i \in \{1, ..., n\}$, the condition of the **await** instruction at L98 does not hold, thus, the task is suspended in state S_i, and, consequently, no switchHandleMessage nor sendOut task can be spawned in any state S_i. Therefore, $\forall i \in \{1, ..., n\}, \forall j \in \{i, ..., n\}, SHM_j$ (resp. SO_j) never contains more tasks than SHM_i (resp. SO_i). Similarly, no other call to barrierRequest can be performed due to the call to barrierWait in L89, which implies that there cannot be two active barriers over the same switch.

Finally, since sw no longer belongs to barrierOn in S_{n+1}, we know that $\forall fut \in h_c[\mathsf{barrierMap}][\mathsf{sid}]$, the task $l_c[fut]$ has finished, since for each variable fut, the **await** statement in L105 has succeeded. Moreover, using the invariant, we know that all the tasks in SHM_n and SO_n have their corresponding future variable in $h_c[\mathsf{barrierMap}][\mathsf{sid}]$, and therefore all of them have finished.

□

7. 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 explosion problem) that must be faced to solve real-world problems. This problem is exacerbated in the context of SDN programs, 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 leading to a huge number of possible executions even for networks with a low number of nodes and packets. Additionally, the state 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[18]) and the other one is to use abstraction (as e.g. in [19]). In the former, the search

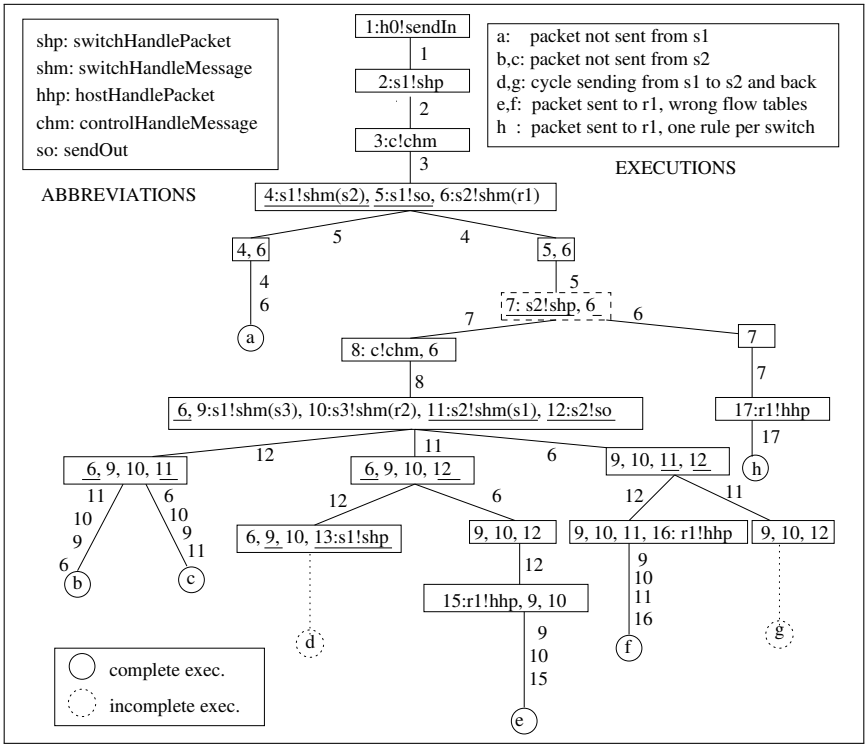

Figure 9: Search tree for running example w/o barriers (rightmost branch w/ barriers)

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. In our case, we impose a bound on the number of packets. The choice of the bound may vary depending on the complexity of the network, the available resources, and the required level of accurateness. For our running example below we use the bound $k = 1$ for simplicity since it allows us to show the necessary details. In our experiments this bound is usually much higher, ranging from $k = 2$ to $k = 300$ packets.

Example 7.1. Figure 9 shows the search tree computed by DPOR for our SDN-Actor program without barriers for $k = 1$. It has no redundancy, i.e., each execution corresponds to a different behaviour 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 init_conf method in Example 5.2 with the instruction sendIn added (after L9) in Example 5.3 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 (h) corresponds to the correct behaviour in which the packet is actually sent to r1 and the actions are installed in the flow tables in the expected order. In execution (a) the packet does not arrive at the destination because the sendOut is executed before the action has been installed. Executions (d) and (g) correspond to the cycle described in Section 3, each of them with different installations of actions.

Importantly, we do not need specific optimizations to use the DPOR algorithm in [20] 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 (h). The difference arises from task 3 in the tree: in the presence of barriers, this leads to a state in which we have asynchronous calls 4 and 6 and task 3 suspended at the **await** in L105 (awaiting for the termination of 4 and then of 6). Therefore, the dependent tasks 4 and 5 will not coexist because 5 is not spawned until 4 and 6 terminate.

7.1. Independence constraints for model checking of SDN-Actors

In our model, all tasks posted on a given switch access its flow table, namely sendOut and switchHandlePacket read it and switchHandleMessage writes it. When considering unconditional independence, 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 usually different packets access distinct entries in the flow table. This results in the inaccurate detection of many dependencies hence producing redundant executions that can be avoided using notions of conditional independence by means of ICs. E.g., tasks executing switchHandleMessage can be considered conditional independent if they write different entries of the flow table – this is indeed an IC for two tasks of this method.

In the context of SDNs, we can distinguish two types of ICs:

1. *Entry-level independence constraints.* When there are several packets being handled by a switch, the tasks for each different packet typically access a distinct entry in the flow table. If we treat the flow table globally, we will not prove independence among the tasks accessing different entries. Instead, the *entry-level independence constraints* claim that two tasks accessing to entry i and j, respectively, can be considered as independent

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 these new ICs are not considered, while it computes 92 executions (exploring 761 states) with Constrained DPOR.

2. *Context-sensitiveness independence constraints.* Even when two tasks t and p access the same entry, the state reached after executing $p \cdot t$ and $t \cdot p$ might be the same. For instance, executing two consecutive switchHandleMessage on the same entry might lead to the same state if the flow table contains duplicate entries, as our implementation allows. An example of independence constraint for these two tasks is the check of duplicate entries in the state.

Using SMT solvers we can semi-automatically obtain these kind of ICs (see [6]). Let us explain the most representative ICs for method switchHandleMessage (m,a):

1. indep(switchHandlePacket(pi,pk),!matchHead&Port(getHeader(pk),pi,m)) denotes that tasks executing switchHandleMessage(m,a) are independent of those executing switchHandlePacket(pi,pk) if the matched field of the message does not match the header and the input port of the packet (the condition is checked by the auxiliary function matchHead&Port).

2. indep(switchHandleMessage(m2,a2),indepSwitchMessageMessage(m,a,m2,a2)) denotes that tasks executing switchHandleMessage(m,a) are independent of those executing switchHandleMessage(m2,a2) if the matched fields m and m2 are independent (they do not match with the same entries in the flow table), but actions a and a2 are equals (the condition is checked by the auxiliary function indepSwitchMessageMessage).

7.2. Comparison of DPOR reductions with related work

Other model checkers for SDN programs have used DPOR-based algorithms before [18, 19]. 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 [19], 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 shown in Example 2 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 SDNs.

8. Implementation and experimental evaluation

This section describes how to use our model checking tool and its visualization capabilities in Section 8.1, and then the experimental evaluation carried out on a series of standard SDN benchmarks in Section 8.2.

8.1. The model checking tool and its visualization capabilities

We have built an extension for property model checking on top of the SYCO tool [4]. It can be used through an online web interface available at: `http://costa.fdi.ucm.es/syco` by selecting the POR algorithm CDPOR and disabling the automatic generation of independence constraints. All benchmarks we are describing in this section can be found in the folder JSS19. In order to run the model checker, the user first opens one of these benchmarks and clicks over the button Apply. By default SYCO makes a full exploration of the execution. However, by using the Settings, it is possible to change the default options. In particular, by selecting Property checking, the exploration finishes after finding an execution trace that violates the property being checked. In order to define the property P under test, 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.

The result of executing the model checker is shown in the console at the bottom, where SYCO first prints the number of executions explored and the output state for each explored execution. The output state contains the actors modelling the controller, the switches and the hosts created during the execution. Each actor is represented as a term with three arguments: the actor identifier, the actor type or class, and the final values of their fields.

8.2. Checking SDN properties in case studies

To evaluate our approach, we have implemented a series of standard SDN benchmarks used in previous work [19, 21, 15]. 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., programming errors in the controller as in [21], safety policy violations as in [21, 19], or loop detection as in [15]).

Moreover, we are able to handle networks larger than in related systems [19], 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 though that a precise comparison of figures is not possible due to the differences described in Section 7.2 and the use of different implementations of controllers.

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 (i.e., the k bound explained in Section 7). Column **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 milliseconds. Results are shown in Figure 10.

Controller with load balancer [15] (LB/LBB). This corresponds to the controller of [15], 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 9ms after exploring 21 states. 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. We are able to scale this version up to 302 hosts and 300 packets. Once we check the correct version with barriers (LBB), we are able to scale up to 127 hosts and 125 packets. As can be observed, for the largest network, 1499 states are explored and in all cases we verify that the traffic is balanced. The experiments in [15] 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 Section 9), and thus time comparison is not meaningful.

SSH controller [19] (SSHE/SSHB). This case study is based on a controller that dynamically modifies the behaviours 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. 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 [19], we evaluate a network with 2 switches and 2 hosts. As for packets, we write `100ssh`, `120other`, and `50each` to indicate that we send 100 SSH packets, 120 non-SSH packets and 50 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 suggest higher performance of our approach: in [19] they find the bug in 0.1s and we do it in 0.004s or 0.007s, depending on the type of packet.

Name	SxHxP	Execs	States	Time
LB	3x52x50	4	313	1305
LB	3x102x100	4	613	7301
LB	3x202x200	4	1213	38203
LB	3x302x300	4	1813	110220
LBB	3x52x50	1	599	11117
LBB	3x77x75	1	899	31644
LBB	3x102x100	1	1199	68059
LBB	3x127x125	1	1499	127740

(a) Controller with load balancer.

Name	SxHxP	Execs	States	Time
SSH	2x2x100ssh	1	407	83824
SSH	2x2x120oth	1	490	146151
SSH	2x2x50each	1	410	117245
SSH	2x2x2cor	179	1691	1340
SSHB	2x2x2	6	120	104
SSHB	2x2x3	65	1419	2506
SSHB	3x3x4	421	10951	33470

(b) SSH controller.

Name	SxHxP	Execs	States	Time
LE	3x3x2	3	71	42
LE	3x3x5	10	383	355
LE	6x3x2	5	217	272
LE	6x3x5	16	1040	2045
LE	9x2x2	10	787	4570
LE	15x2x2	16	2074	49274

(c) Network authentication with learning.

Name	SxHxP	Execs	States	Time
MIb	1x5x12	32	599	1029
MIb	1x5x14	64	1107	2730
MIb	1x5x16	748	9418	24870
MIb	1x8x20	2242	45539	153419
MI	1x5x8	32	1004	865
MI	1x5x10	256	9436	9176
MI	1x5x12	960	17941	29675
MI	1x8x14	1727	55200	119908

(d) Firewall with migration.

Figure 10: Experimental results.

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. They prove the correctness for SSHB-2-2 in 6.4 seconds by exploring 13 states, we explore 15 states (in 6ms) or 18 states (in 8 ms), depending on the type of packet. Furthermore we are able to scale up to 3 hosts and 3 switches.

Network authentication with learning [21, 19] (LE). This implements a composition of a learning switch with authentication in [21]. Also, [19] evaluates a MAC learning controller but using a different implementation. LE implements a controller with barriers for which we can verify flow-table consistency and that the packet flows satisfy the intended policy. We have considered configurations of 3x3, 6x3, 9x2 and 15x2. When compared to [19], we handle larger sizes of networks and for similar sizes, we explore less **States** in less **Time**. We note that this might be due to the differences pointed out in Section 7.2 and different implementations of the controller.

Firewall with migration[21] (MIb/MI). 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 [21], we can scale the number of packets up to 12 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 [21], which forgets to check if trusted on events from port 2. We detect the error by checking in the final state of the execution traces that a packet arrives to a host that is not in the trusted data base. The scalability of MI and MIb are rather similar. However, we can handle larger sizes of networks (1x8). Like [21] we find the bug in a negligible time.

9. Conclusions

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. This has allowed us to model and analyse several SDN scenarios: a controller with load balancer, an SSH controller, a learning switch with authentication, and a firewall with migration. Experiments have given evidence of the versatility and scalability of our approach.

We conclude with a review of related work in verification of software-defined networks and some directions for future work.

9.1. Related work

Static and Dynamic verification.. The last years have witnessed the development of many static and dynamic techniques for verification that are closely related to our approach. Static approaches have 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 behaviours, static analysis needs to perform abstraction, which can give a don't-know answer, and, possibly, false positives. In [21], 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. Our framework can additionally check liveness invariants (e.g., loop detection) by inspecting the traces computed by the model checker. Static algebraic techniques are used in NetKAT [22, 23, 24], to prove properties of SDN programs. NetKAT does not include primitives for concurrency, and has a significantly higher level of abstraction. Therefore capturing features and scenarios we are interested in would be difficult. In [25], a particular type of attacks in the context of SDNs has been modelled in Maude using the so-called hierarchically structured composite actor systems described in [26]. This work does not provide a general model for SDNs and, besides, barriers are not considered. On the other hand, it applies a statistical model checker, which requires us to have a given scheduler for the messages. Such a 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.

Concerning dynamic techniques, our work is mostly related to the model checkers NICE and Kuai for SDN programs, which have been compared in detail in Section 7.2. Our approach could be adapted to apply abstractions that bound the size of buffers [19] and to consider environment messages [27]. The approach of [15, 28] is based on analyzing dynamically given snapshots of the network from real executions. Instead, we try to find programming errors by inspecting only the SDN program and considering all possible execution traces, thus enabling verification at system design time.

Data and Control-plane verification. There is a substantial body of work on verification techniques for SDN focussing specifically on the data or the control plane. Data-plane approaches include: Anteater [29], which uses static analysis via SAT solving; FlowChecker [30], which applies symbolic model-checking to OpenFlow configurations; VeriFlow [31], which provides an infrastructure to check data-plane properties in real-time. Control-plane approaches include: Flowlog [32], a declarative language to program SDN controllers, which uses the Alloy model-checker to perform verification; [33], which uses differential analysis to discover bugs in different versions of the same controller program.

We stress that our approach targets both control and data-plane, and in particular it is capable of detecting bugs that arise from their interaction. Moreover, concurrency and barriers are not considered in the mentioned works.

Quantitative verification. In [34], SDN components are modelled via a quantitative process algebra. Their focus is on quantitative properties, e.g., latency and congestion. In particular, concurrency and barriers are not considered.

Network verification via actors. Another actor-based verification framework is *Rebeca* (see [35] for a survey). Rebeca supports a variety of state-reduction techniques, and has been used to model and verify wireless networks [36, 37]. Our approach uses the ABS language and the SYCO tool. SYCO includes recent DPOR techniques [6, 20] which, by exploiting specific features of SDNs, enabled us to better scale and analyse larger networks.

9.2. Future Work

Although we did not explore it in this article, 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 behaviour can be done now using the ABS toolsuite [7]. Other tools and methods for verification of message-passing and concurrent-object systems could be also easily adapted [38, 39, 40, 41]. 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. This is left for future work. Finally, an interesting direction for future work are quantitative extensions, for the purpose of statistical model-checking and comparative performance analyses of SDN protocols. This could be achieved via tools such as pMaude [42], which support probabilistic actor formalisms.

Acknowledgments This work was partially funded by the Spanish MECD Salvador de Madariaga Mobility Grants PRX17/00297 and PRX17/00303, the Spanish FPU Grant FPU15/04313, the Spanish MCIU, AEI and FEDER (EU) through projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33 and the CM project S2018/TCS-4314, the ERC starting grant Profoundnet (679127) and a Leverhulme Prize (PLP-2016-129).

References

[1] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, 1986.

[2] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: FMCO, 2010, pp. 142–164.

[3] Openflow switch specification, version 1.4.0 (October 2013).

[4] E. Albert, M. Gómez-Zamalloa, M. Isabel, SYCO: a systematic testing tool for concurrent objects, in: CC, 2016, pp. 269–270. doi:10.1145/2892208.2892236.

[5] E. Albert, M. Gómez-Zamalloa, A. Rubio, M. Sammartino, A. Silva, Sdn-actors: Modeling and verification of SDN programs, in: FM, 2018, pp. 550–567. doi:10.1007/978-3-319-95582-7_33.

[6] E. Albert, M. Gómez-Zamalloa, M. Isabel, A. Rubio, Constrained dynamic partial order reduction, in: CAV, 2018, pp. 392–410. doi:10.1007/978-3-319-96142-2_24.

[7] The ABS tool suite, `http://abs-models.org`.

[8] F. S. de Boer, D. Clarke, E. B. Johnsen, A Complete Guide to the Future, in: ESOP, Vol. 4421, 2007, pp. 316–330.

[9] C. Flanagan, P. Godefroid, Dynamic partial-order reduction for model checking software, in: POPL, 2005, pp. 110–121.

[10] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, G. Agha, Transdpor: A novel dynamic partial-order reduction technique for testing actor programs, in: FMOODS/FORTE, 2012, pp. 219–234.

[11] S. Katz, D. A. Peled, Defining conditional independence using collapses, Theor. Comput. Sci. 101 (2) (1992) 337–359.

[12] P. Godefroid, D. Pirottin, Refining dependencies improves partial-order verification methods (extended abstract), in: Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings, 1993, pp. 438–449. doi:10.1007/3-540-56922-7_36.
URL `https://doi.org/10.1007/3-540-56922-7\_36`

[13] C. Wang, Z. Yang, V. Kahlon, A. Gupta, Peephole partial order reduction, in: TACAS, 2008, pp. 382–396.

[14] V. Kahlon, C. Wang, A. Gupta, Monotonic partial order reduction: An optimal symbolic partial order reduction technique, in: CAV, 2009, pp. 398–413.

[15] A. El-Hassany, J. Miserez, P. Bielik, L. Vanbever, M. T. Vechev, Sdnracer: concurrency analysis for software-defined networks, in: POPL, 2016, pp. 402–415. doi:10.1145/2908080.2908124.

[16] A. Guha, M. Reitblatt, N. Foster, Machine-verified network controllers, in: PLDI, 2013, pp. 483–494. doi:10.1145/2491956.2462178.

[17] K. Sen, G. Agha, Automated Systematic Testing of Open Distributed Programs, in: FASE, 2006, pp. 339–356.

[18] M. Canini, D. Venzano, P. Peresíni, D. Kostic, J. Rexford, A NICE way to test openflow applications, in: NSDI, 2012, pp. 127–140.

[19] R. Majumdar, S. D. Tetali, Z. Wang, Kuai: A model checker for software-defined networks, in: FMCAD, 2014, pp. 163–170. doi:10.1109/FMCAD.2014.6987609.

[20] E. Albert, P. Arenas, M. G. de la Banda, M. Gómez-Zamalloa, P. J. Stuckey, Context-sensitive dynamic partial order reduction, in: CAV, Vol. 10426, 2017, pp. 526–543.

[21] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, A. Valadarsky, Vericon: towards verifying controller programs in software-defined networks, in: PLDI, 2014, pp. 282–293. doi:10.1145/2594291.2594317.

[22] N. Foster, D. Kozen, M. Milano, A. Silva, L. Thompson, A coalgebraic decision procedure for netkat, in: POPL, 2015, pp. 343–355. doi:10.1145/2676726.2677011.

[23] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, D. Walker, Netkat: semantic foundations for networks, in: POPL, 2014, pp. 113–126. doi:10.1145/2535838.2535862.

[24] R. Beckett, M. Greenberg, D. Walker, Temporal netkat, in: PLDI, 2016, pp. 386–401. doi:10.1145/2908080.2908108.

[25] T. A. Pascoal, Y. G. Dantas, I. E. Fonseca, V. Nigam, Slow TCAM exhaustion ddos attack, in: SEC, 2017, pp. 17–31.

[26] J. Eckhardt, T. Mühlbauer, J. Meseguer, M. Wirsing, Statistical model checking for composite actor systems, in: WADT, 2012, pp. 143–160.

[27] D. Sethi, S. Narayana, S. Malik, Abstractions for model checking SDN controllers, in: FMCAD, 2013, pp. 145–148.

[28] P. Kazemian, G. Varghese, N. McKeown, Header space analysis: Static checking for networks, in: NSDI, 2012, pp. 113–126.

[29] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, S. T. King, Debugging the data plane with anteater, in: ACM SIGCOMM, 2011, pp. 290–301. doi:10.1145/2018436.2018470.

[30] E. Al-Shaer, S. Al-Haj, Flowchecker: configuration analysis and verification of federated openflow infrastructures, in: SafeConfig, 2010, pp. 37–44. doi:10.1145/1866898.1866905.

[31] A. Khurshid, X. Zou, W. Zhou, M. Caesar, P. B. Godfrey, Veriflow: Verifying network-wide invariants in real time, in: NSDI, 2013, pp. 15–27.

[32] T. Nelson, A. D. Ferguson, M. J. G. Scheer, S. Krishnamurthi, Tierless programming and reasoning for software-defined networks, in: NSDI, 2014, pp. 519–531.

[33] T. Nelson, A. D. Ferguson, S. Krishnamurthi, Static differential program analysis for software-defined networks, in: FM, 2015, pp. 395–413. doi:10.1007/978-3-319-19249-9_25.

[34] V. Galpin, Formal modelling of software defined networking, in: IFM, 2018, pp. 172–193. doi:10.1007/978-3-319-98938-9_11.

[35] M. Sirjani, M. M. Jaghoori, Ten years of analyzing actors: Rebeca experience, in: Formal Modeling: Actors, Open Systems, Biological Systems, 2011, pp. 20–56. doi:10.1007/978-3-642-24933-4_3.

[36] B. Yousefi, F. Ghassemi, R. Khosravi, Modeling and efficient verification of wireless ad hoc networks, Formal Asp. Comput. 29 (6) (2017) 1051–1086. doi:10.1007/s00165-017-0429-z.

[37] E. Khamespanah, M. Sirjani, K. Mechitov, G. Agha, Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking, STTT 20 (5) (2018) 547–561. doi:10.1007/s10009-017-0480-3.

[38] M. Christakis, A. Gotovos, K. F. Sagonas, Systematic Testing for Detecting Concurrency Errors in Erlang Programs, in: ICST, 2013, pp. 154–163.

[39] S. Lauterburg, R. K. Karmani, D. Marinov, G. Agha, Basset: a tool for systematic testing of actor programs, in: SIGSOFT FSE, 2010, pp. 363–364. doi:10.1145/1882291.1882349.

[40] A. Bouajjani, M. Emmi, C. Enea, J. Hamza, Tractable refinement checking for concurrent objects, in: POPL, 2015, pp. 651–662. doi:10.1145/2676726.2677002.

[41] H. Liang, X. Feng, A program logic for concurrent objects under fair scheduling, in: POPL, 2016, pp. 385–399. doi:10.1145/2837614.2837635.

[42] G. A. Agha, J. 'e Meseguer, K. Sen, Pmaude: Rewrite-based specification language for probabilistic object systems, electron. Notes theor. Comput. Sci. 153 (2) (2006) 213–239. doi:10.1016 / j.entcs.2005.10.040.

Appendix A. Notation Table

Notation	Meaning
Section 1	
SDN	Software-Defined Networking
ABS	Object-oriented and actor-based modeling language
SYCO	Systematic Testing Tool for Concurrent Objects
DPOR	Dynamic Partial Order Reduction
SDN-Actor	Actor-based encoding of a SDN element
Section 2	
$\overline{x}$	Shorthand for $x_1, ..., x_n$
$\mathsf{m}(\overline{x})$	Synchronous call to method m with arguments $\overline{x}$
$\mathsf{a!m}(\overline{x})$	Asynchronous call to method m with arguments $\overline{x}$ in actor a
Fut<**Unit**> f=$\mathsf{a!m}(\overline{x})$	Future variable f bound to the asynchronous call $\mathsf{a!m}(\overline{x})$
await f?	Non-blocking wait for the task bound to future variable f
IC	Independence constraint
SMT	Satisfiability Modulo Theories
Section 4	
Pkt	Packet
PktIn	Message Packet−in in SDN
ModState	Message Modify−State in SDN
PktOut	Message Packet−out in SDN
$header(p)$	Function that returns the header of a packet p
$id(p)$	Function that returns the id of a packet p
$\langle ph, o\rangle \mapsto a$	Flow-table entry associating action a with header ph and port o
$send(id)$	Action to deliver the corresponding packet to the host id
$\langle send(id), o\rangle$	Action to deliver the corresponding packet to the switch id using port o
$applyPol$	Function that applies the controller's policy
$new(p)$	New packet p
$o : p$	Packet p with its associated port o
SI	SDN transition rule SendIn
HHP	SDN transition rule HostHandlePacket
SHP	SDN transition rule SwitchHandlePacket
SO	SDN transition rule SendOut
SHM	SDN transition rule SwitchHandleMessage
CHM	SDN transition rule ControlHandleMessage
tk	Task identifier
$\mathcal{Q}$	Queue of pending tasks in ABS semantics
$\mathbb{V}$	Set of references and values in ABS semantics

Notation	Meaning
MSTEP	ABS transition rule Macro Step
ASY	ABS transition rule Asynchronous Call
SYN	ABS transition rule Synchronous Call
NEW	ABS transition rule New Actor Creation
$selectAct(S)$	ABS semantics function that selects an actor in state S
$selectTask(A)$	ABS semantics function that selects a task in actor A
$\longmapsto$	ABS macro-step transition
$\overset{o\cdot tk}{\rightsquigarrow}$	ABS micro-step transition where actor o executes task tk
Section 5	
SHlink	Actor-based encoding of a Switch-Host relation
SSlink	Actor-based encoding of a Switch-Switch relation
buf	Buffer
$A \equiv B$	A is equivalent to B
A'	Given the set A and $a \in A$, A' is the set such that $A = \{a\} \cup A'$
flowT	Flow table
ntw	Network
S^{ini}	The initial SDN state
S_a^{ini}	The SDN-Actor state after executing method init_conf()
Section 6	
OF Barrier	OpenFlow Barrier
Section 8	
LB	Load Balancer benchmark
LBB	Load Balancer with Barriers benchmark
VIP	Virtual IP
SSH	Secure Shell protocol
SSHE	SSH erroneous benchmark
SSHB	SSH with barriers benchmark
LE	Network authentication with learning benchmark
MAC	Media Access Control
MI	Firewall with migration benchmark
SxHxP	Num. Switches x Num. Hosts x Num. Packets

Appendix B. Soundness Proofs

Appendix B.1. Proof of Theorem 5.7

Let α be the one-to-one function that pairs each element of S with its unique actor in S_a such that $S \equiv S_a$.

Assumption 1. *Given an SDN state $S = \langle H, Sw, C\rangle$ and an SDN-Actor state S_a such that $S \equiv S_a$, $C = c(top, cinp)$ and $a(c, tk, h, \mathcal{Q}) \in S_a$ with $\alpha(C) = a(c, tk, h, \mathcal{Q})$, then*

$$applyPol(top, sid, o, ph) = applyPolicy(sid, o, ph)$$

Assumption 2. *$S^{ini} \equiv S_a^{ini}$, where S_a^{ini} is the SDN-Actor state defined in Def.5.1, that is, the state after executing method* init_conf() *and all asynchronous calls to method* sendIn *containing the packets to be delivered.*

Definition Appendix B.1 (succ(S) (succ(S_a))). *Given an SDN(resp. SDN-Actor) state S (resp. S_a), succ(S) (resp. succ(S_a)) denotes the set of final states of the executions in exec(S) (resp. exec(S_a)).*

Lemma Appendix B.2. *Given an SDN state and an SDN-Actor state S_a such that $S \equiv S_a$ and $S \in succ(S^{ini})$ and $S_a \in succ(S_a^{ini})$,*

1. *If $S \to S'$, then $\exists S_a'$ such that $S_a \longmapsto S_a'$ and $S' \equiv S_a'$.*
2. *If $S_a \longmapsto S_a'$, then $\exists S'$ such that $S \to S'$ and $S' \equiv S_a'$.*

Proof.
Let us see reason about both points at the same time. We distinguish several cases depending on the semantics rule applied during the step $S \to S'$. Let S and S' be $\langle H, Sw, C\rangle$ and $\langle H', Sw', C'\rangle$, respectively.

1. If rule (SI) is applied,

   - $H = H'' \cup \{h(id, sid, o, inp \cup \{new(p)\})\}$ and $H' = H'' \cup \{h(id, sid, o, inp)\}$
   - $Sw = Sw'' \cup \{s(sid, ft, b, inp')\}$ and $Sw' = Sw'' \cup \{s(sid, ft, b, inp' \cup \{o{:}p\})\}$
   - $C = C'$

   Moreover, we know that $S \equiv S_a$, hence $\alpha(h(id, sid, o, inp \cup \{new(p)\})) = a(hoid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendIn}, l, \_)\} \cup \mathcal{Q}'$. Thus, task tid can be executed in state S_a by actor $hoid$ and it will send a task shp (where shp stands for switchHandlePacket) to $h[\mathsf{s}]$ such that $h[\mathsf{s}] = \alpha(s(sid, ft, b, inp')) = a(soid, \_, h2, \mathcal{Q}'') \in S_a$ (by the equivalence of S and S_a).

   a) $S_a = S_a'' \cup \{a(hoid, \_, h, \mathcal{Q}), a(soid, \_, h2, \mathcal{Q}''), \}$
   b) $S_a' = S_a'' \cup \{a(hoid, \_, h, \mathcal{Q}'), a(soid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\})\}$, such that $l2[\mathsf{p}] = l[\mathsf{p}]$ and $l2[\mathsf{o}] = h[\mathsf{o}]$.

By such equivalence we know that (1) $\langle H'', Sw'', C\rangle \equiv S''_a$, (2) $(inp, \emptyset) \equiv (\mathcal{Q}', \emptyset)$ and by a) and b) $(inp' \cup \{o : p\}, \emptyset) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}, \emptyset)$ Consequently, $S' \equiv S'_a$.

2. If rule (HHP) is applied,

   - $H = H'' \cup \{h(id, sid, o, inp \cup \{p\})\}$ and $H' = H'' \cup \{h(id, sid, o, inp)\}$
   - $Sw = Sw'$ and $C = C'$

   We also know $S \equiv S_a$, hence $\alpha(h(id, sid, o, inp \cup \{p\})) = a(hoid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{hhp}, l, \_)\} \cup \mathcal{Q}'$ (where hhp stands for hostHandlePacket). Therefore task tid can be executed in state S_a by $hoid$ and the packet is removed from the buffer.

   - $S_a = S''_a \cup \{a(hoid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{a(hoid, \_, h, \mathcal{Q}')\}$.

   Furthermore, $p = l[p]$. By the equivalence between S_a and S, we know that $(inp \cup \{p\}, \emptyset) \equiv (\mathcal{Q}, \emptyset)$ and then, (1) $(inp, \emptyset) \equiv (\mathcal{Q}', \emptyset)$ and also, (2) $\langle H'', Sw, C\rangle \equiv S''_a$. Consequently, $S' \equiv S'_a$.

3. If rule (SHP_1) is applied,

   - $H = H'' \cup \{h(id, sid, o, inp'\}$ and $H' = H'' \cup \{h(id, sid, o, inp' \cup \{p\})\}$
   - $Sw{=}Sw'' \cup \{s(sid, ft, b, inp \cup \{o2 : p\})\}$ and $Sw'{=}Sw'' \cup \{s(sid, ft, b, inp)\}$
   - $C = C'$ and $\langle send(id)\rangle = lookup(ft, \langle header(p), o\rangle)$.

   Moreover, we know $S \equiv S_a$, hence $\alpha(s(sid, ft, b, inp)) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{shp}, l, \_)\} \cup \mathcal{Q}'$. Therefore, task tid can be executed and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $send(id)$. Therefore, the check in line 21 does not succeed, but the check in line 22 does, and, consequently, it spawns a task hhp to $\alpha(h(id, sid, o, inp')) = a(hoid, \_, h2, \mathcal{Q}'') \in S_a$. As a result

   - $S_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}), a(soid, \_, h, \mathcal{Q}')\}$

   where $l[\mathsf{p}] = l2[\mathsf{p}]$. We also know by $S \equiv S_a$, that (1) $\langle H'', Sw'', C\rangle \equiv S''_a$, (2) $(inp \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Since $p = l2[p]$, then $(inp' \cup \{p\}, b) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}, h[\mathsf{buffer}])$. Consequently, we have that $S' \equiv S'_a$.

4. If rule (SHP_2) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b, inp \cup \{o2 : p\}), s(sid', ft', b', inp')\}$
   - $Sw' = Sw'' \cup \{s(sid, ft, b, inp), s(sid', ft', b', inp' \cup \{o : p\})\}$
   - $H = H',\ C = C'$ and $send(sid', o) = lookup(ft, \langle header(p), o2\rangle)$.

Moreover, we know that $S \equiv S_a$, then $\alpha(s(sid, ft, b, inp)) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{shp}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed, and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $send(soid', o)$. Therefore, the check in line 21 succeeds and, consequently, it spawns a task shp to $\alpha(s(sid', ft', b', inp')) = a(soid', \_, h2, \mathcal{Q}'') \in S_a$. As a result

- $S_a = S''_a \cup \{a(soid', \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
- $S'_a = S''_a \cup \{s(soid', \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}), a(soid, \_, h, \mathcal{Q}')\}$

where $l[p] = l2[p]$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(inp \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Finally, we also know that $(inp' \cup \{o : p\}, b') \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}, h2[\mathsf{buffer}])$ because $o = l2[\mathsf{o}], p = l2[\mathsf{p}]$. Consequently, we have that $S' \equiv S'_a$.

5. If rule SHP_3 is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b, inp \cup \{o : p\})\}$ and $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o : p\}, inp)\}$
   - $C = c(top, inp')$ and $C' = c(top, inp' \cup \{\mathsf{pktIn(sid,o,id(p),header(p))}\})$
   - $H = H'$ and $\bot = lookup(ft, \langle header(p), o\rangle)$.

   Moreover, we know that $S \equiv S_a$, then $\alpha(s(sid, ft, b, inp)) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{shp}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $\bot$. Therefore, the checks in line 21 and 22 do not succeed and, consequently, it (1) spawns a task chm (where chm stands for$\mathsf{controlHandleMessage}$) to $h[ctrl] = \alpha(c(top, inp')) = a(coid, \_, h2, \mathcal{Q}'') \in S_a$, and, (2) it stores the packet and the port $o : p$ in $h[\mathsf{buffer}]$. As a result

   - $S_a = S''_a \cup \{a(soid, \_, h, \mathcal{Q}), a(coid, \_, h2, \mathcal{Q}'')\}$.
   - $S'_a = S''_a \cup \{a(soid, \_, h', \mathcal{Q}'), a(coid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{chm}, l2, \_)\})\}$ and $h' := h$ but $h'[\mathsf{buffer}] := h[\mathsf{buffer}] \cup \{(p, o)\}$.

   Furthermore, we know that (1) $\langle H, Sw'', C\rangle \equiv (S''_a \cup \{a(coid, \_, h2, \mathcal{Q}'')\})$ and (2) $(inp, b \cup \{o : p\}) \equiv (\mathcal{Q}', h[\mathsf{buffer}] \cup \{(p, o)\})$. Moreover, we have $(inp', \emptyset) \equiv (\mathcal{Q}'' \cup \{task\ (\_, \mathsf{chm}, l2, \_\}, \emptyset)$ since $soid = l2[\mathsf{sid}]$, $p = l[\mathsf{p}]$ and $o = l[\mathsf{o}]$. Consequently, we have $S' \equiv S'_a$.

6. If rule (SO_1) is applied,

   - $H = H'' \cup \{h(id, sid, o, inp'\}$ and $H' = H'' \cup \{h(id, sid, o, inp' \cup \{o : p\})\}$
   - $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o2 : p\}, inp \cup \{\mathsf{pktOut(ph)}\})\}$ and $Sw' = Sw'' \cup \{s(sid, ft, b, inp)\}$

- $C = C'$, $ph = header(p)$ and $send(id) = lookup(ft, \langle header(p), o\rangle)$.

We also know $S \equiv S_a$, then $\alpha(s(sid, ft, b \cup \{o2 : p\}, inp \cup \{\mathsf{pktOut(ph)}\})) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendOut}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $send(soid, o)$. Therefore, the check in line 28 does not succeed, but the check in line 29 does, and, consequently, it spawns a task hhp to $\alpha(h(id, sid, o, inp')) = a(hoid, \_, h2, \mathcal{Q}'') \in S_a$. As a result

- $S_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
- $S'_a = S''_a \cup \{a(hoid, \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}), a(soid, \_, h', \mathcal{Q}')\}$

where $h' := h$ but $h'[\mathsf{buffer}] := take(h[\mathsf{buffer}], l[\mathsf{pi}])$. We also know by $S \equiv S_a$, that (1) $\langle H'', Sw'', C\rangle \equiv S''_a$, (2) $(inp \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Finally, we also know that $(inp' \cup \{o : p\}, b) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{hhp}, l2, \_)\}, h[\mathsf{buffer}])$ since $(p, o) = take(h[\mathsf{buffer}], l[\mathsf{pi}]) = (l2[\mathsf{p}], o)$. Consequently, we have that $S' \equiv S'_a$.

7. If rule (SO_2) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o2 : p\}, inp \cup \{\mathsf{pktOut(ph)}\}), s(sid', ft', b', inp')\}$
   - $Sw' = Sw'' \cup \{s(sid, ft, b, inp), s(sid', ft', b', inp' \cup \{o : p\})\}$
   - $H = H'$, $C = C'$ and $send(sid', o) = lookup(ft, \langle header(p), o2\rangle)$.

   Moreover, we know $S \equiv S_a$, then $\alpha(s(sid, ft, b \cup \{o2 : h\}, inp \cup \{\mathsf{pktOut(ph)}\})) = a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendOut}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed and, by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $send(soid', o)$. Therefore, the check in line 28 succeeds, and consequently, it spawns a task shp to $\alpha(s(sid', ft', b', inp')) = a(soid', \_, h2, \mathcal{Q}'') \in S_a$. As a result

   - $S_a = S''_a \cup \{a(soid', \_, h2, \mathcal{Q}''), a(soid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{s(soid', \_, h2, \mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}), a(soid, \_, h, \mathcal{Q}')\}$

   where $h' := h$ but $h'[\mathsf{buffer}] := take(h[\mathsf{buffer}], l[\mathsf{pi}])$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(inp \cup \{o2 : p\}, b) \equiv (\mathcal{Q}, h[\mathsf{buffer}])$. Finally, we also know that $(inp' \cup \{o : p\}, b) \equiv (\mathcal{Q}'' \cup \{\text{TK}(\_, \mathsf{shp}, l2, \_)\}, h[\mathsf{buffer}])$ since $(p, o) = take(h[\mathsf{buffer}], l[\mathsf{pi}]) = (l2[\mathsf{p}], l2[\mathsf{o}])$. Consequently, we have that $S' \equiv S'_a$.

8. If rule (SO_3) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b \cup \{o : p\}, inp \cup \{\mathsf{pktOut(ph)}\})\}$
   - $Sw' = Sw'' \cup \{s(sid, ft, b, inp)\}$
   - $H{=}H'$, $C{=}C'$, and $ph{=}header(p)$ and $\bot{=}lookup(ft, \langle header(p), o\rangle)$.

Moreover, we know that $S \equiv S_a$, then $\alpha(s(sid, ft, b \cup \{o{:}p\}, inp \cup \{\mathsf{pktOut(ph)}\}))$ $= a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{sendOut}, l, \_)\} \cup \mathcal{Q}'$. Hence, task tid can be executed, and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, thus the action returned by flowT is the same that the one returned by ft, that is $\perp$. Therefore, the checks in line 28 and 29 do not succeed, and consequently, it drops the packet without spawning any other task. As a result

- $S_a = S''_a \cup \{a(soid, \_, h, \mathcal{Q})\}$
- $S'_a = S''_a \cup \{a(soid, \_, h', \mathcal{Q}')\}$

where $h' := h$ but $h'[\mathsf{buffer}] := take(h[\mathsf{buffer}], l[\mathsf{pi}])$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(inp, b) \equiv (\mathcal{Q}', h'[\mathsf{buffer}])$. Consequently, we have that $S' \equiv S'_a$.

9. If rule (SHM) is applied,

   - $Sw = Sw'' \cup \{s(sid, ft, b, inp \cup \{\mathsf{modState}(\langle \mathsf{ph},\ \mathsf{o}\rangle \mapsto \mathsf{a})\})\}$
   - $Sw' = Sw'' \cup \{s(sid, put(ft, \langle ph, o\rangle, a), b, inp)\}$
   - $H = H',\ C = C'$.

   Moreover, we know that $S \equiv S_a$, then $\alpha(s(sid, ft, b, inp \cup \{\mathsf{modState}(\langle \mathsf{ph,o}\rangle \mapsto \mathsf{a})\}))$ $= a(soid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{switchHandleMessage}, l, \_)\} \cup \mathcal{Q}'$. Therefore, task tid can be executed, and by the equivalence, we know that $ft = h[\mathsf{flowT}]$, hence $put(ft, m, a) = put(h[\mathsf{flowT}], \langle ph, o\rangle, a)$, and $m = \langle ph, o\rangle$ because of Assumption 1, that is, *applyPol* and *applyPolicy* behaves similarly for $\alpha(s)$ and s, $\forall s \in Sw$ and $\alpha(s) \in S$.

   - $S_a = S''_a \cup \{a(soid, \_, h, \mathcal{Q})\}$
   - $S'_a = S''_a \cup \{a(soid, \_, h', \mathcal{Q}')\}$

   where $h' := h$ but $h'[\mathsf{flowT}] := put(h[\mathsf{flowT}], m, a)$. We also know by $S \equiv S_a$, that (1) $\langle H, Sw'', C\rangle \equiv S''_a$, (2) $(inp, b) \equiv (\mathcal{Q}', h[\mathsf{buffer}])$. Consequently, we have that $S' \equiv S'_a$.

10. If rule (CHM) is applied,

    - $Sw = Sw'' \cup \{s(sid, ft, b, inp)\}$
    - $Sw' = Sw''_{ms} \cup \{s(sid, ft, b, inp \cup ms_{sid} \cup \{\mathsf{pktOut(ph)}\}\}$,
    - $H = H'$, $C = c(top, cinp \cup \{\mathsf{pktIn(sid,o,pid,ph)}\})$ and $C' = c(top, cinp)$

    where $ms = applyPol(top, sid, o, ph)$ and $ms_{id} = \{m | \langle id, m\rangle \in ms\}$. Moreover, we know that $S \equiv S_a$, then $\alpha(c(top, cinp \cup \{\mathsf{pktIn(sid,o,pid,ph)}\}))$ $= a(coid, \_, h, \mathcal{Q}) \in S_a$ and $\mathcal{Q} = \{\text{TK}(tid, \mathsf{chm}, l1, \_)\} \cup \mathcal{Q}'$ (where chm stands for chm) such that $soid = l1[\mathsf{sid}]$, $o = l1[\mathsf{o}]$, $pid = l1[\mathsf{p}]$ and $ph = l1[\mathsf{h}]$. Therefore, task tid can be executed, and by the equivalence of S and S_a

and Assumption 1, we know that the list l is equivalent to ms, in the sense that it contains exactly the same switches and the same actions. Hence, in line 41, actor *coid* spawns tasks shm to every switch in the list l (where shm stands for switchHandleMessage) and finally, it spawns a a task sendOut to actor *soid* in line 43. Let us see the equivalence between S'_a and S'.

- $\forall s(sid', ft', b', inp') \in Sw''$ such that $ms_{sid'} = \emptyset$, then $s(sid, ft', b', inp') \in Sw''_{ms}$. Furthermore, if $ms_{sid'} = \emptyset$, thus $sid' \notin l$, hence sid' will not receive any message. Therefore, $\alpha(s(sid', ft', b', inp')) \in S_a \cap S'_a$.
- $\forall s(sid', ft', b', inp') \in Sw''$ such that $ms_{sid'} \neq \emptyset$,then $s(sid, ft', b', inp' \cup ms_{sid'}) \in Sw''_{ms}$. Then, *coid* will spawn as many shm tasks as messages in $\{(m, a) | (soid', m, a) \in l\}$ (and in $ms_{soid'}$). By $S_a \equiv S$, we know $\alpha(s(sid, ft', b', inp')) = a(soid', \_, h2, \mathcal{Q}'') \in S_a$ and $(inp', b') \equiv (\mathcal{Q}'', h2[\mathsf{buffer}])$. Furthermore, $a(soid', \_, h2, \mathcal{Q}'' \cup tks_{l,soid'}) \in S'_a$, where $tks_{l,soid'} := \{\text{TK}(\_, \mathsf{shm}, l', \_) \mid (soid', m, a) \in l, l'[\mathsf{m}] := m,\ l'[\mathsf{a}] := a\}$ which is equivalent to the information contained in $ms_{sid'}$. Then, $(inp' \cup ms_{sid'}, b) \equiv (\mathcal{Q}'' \cup tks_{l,soid'}, h2[\mathsf{buffer}])$.
- Regarding the switch $s(sid, ft, b, inp)$, by the equivalence of S and S_a, we know that $\alpha(s(sid, ft, b, inp)) = a(soid, \_, h1, \mathcal{Q}_{soid}) \in S_a$ and since $soid = l[\mathsf{sid}]$, actor *coid* spawns a task sendOut, and as many shm tasks as messages in $\{(m, a) | (soid, m, a) \in l\}$, and then $a(soid, \_, h1, \mathcal{Q}_{soid} \cup tks_{l,soid} \cup \{\text{TK}(\_, \mathsf{sendOut}, l', \_)\}) \in S'_a$. Again, by the equivalence of S and S_a, we know that $(inp, b) \equiv (\mathcal{Q}_{soid}, h1[\mathsf{buffer}])$ and, since $tks_{l,soid}$ is the equivalent information contained in ms_{sid}, thus we get that $(inp \cup ms_{sid}, b) \equiv (\mathcal{Q}_{soid} \cup tks_{l,soid}, h1[\mathsf{buffer}])$. Finally, pktOut(ph) contains the equivalent information to $t_{out} := \text{TK}(\_, \mathsf{sendOut}, l', \_)$, thus we get $(inp' \cup (ms_{sid'} \cup \{\mathsf{pktOut(ph)}\}, b) \equiv (\mathcal{Q}'' \cup tks_{l,soid'} \cup \{t_{out}\}, h[\mathsf{buffer}])$.
- Regarding the controller C, we know that $(cinp \cup \{\mathsf{pktIn(sid,o,pid,ph)}\}, \emptyset) \equiv (\mathcal{Q} \cup \{\text{TK}(tkid, \mathsf{chm}, l1, \_)\}, \emptyset)$. Furthermore, by Assumption 1, we know that $related(top, \{h[\mathsf{srefs}, \mathsf{href}, \mathsf{ntw}]\})$. As a consequence $(cinp, \emptyset)$
  $\equiv (\mathcal{Q}, \emptyset)$.

All in all, we conclude that $S' \equiv S'_a$.

Let us notice here that even though we have distinguished the different cases depending on the semantics rule for SDNs, the previous reasoning also includes each possible execution of a task in the SDN-Actor model. Hence, each possible execution of a task corresponds exactly with one of the semantics rule for SDNs. □

Theorem 5.7. *Let S^{ini} and S^{ini}_a be an SDN state and an SDN-Actor state, respectively.*

1. *For every execution $S^{ini} \to S^1 \to ... \to S^n \in exec(S^{ini}), \exists S^{ini}_a \longmapsto S^1_a \longmapsto ... \longmapsto S^n_a \in exec(S^{ini}_a)$ such that $S^n \equiv S^n_a$.*

2. *For every execution* $S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini}), \exists S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \in exec(S^{ini})$ *such that* $S^n \equiv S_a^n$.

Proof.
Let us prove both cases by induction on the length n of the execution.

- If $n = 0$, it is straightforward to see that $S^0 = S^{ini} \equiv S_a^{ini} = S_a^0$.
- Let us suppose that both cases are true for n and let us prove them for $n + 1$

    1. We need to prove that for every execution $S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^{n+1} \in exec(S^{ini})$, $\exists S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \longmapsto S_a^{n+1} \in exec(S_a^{ini})$ such that $S^{n+1} \equiv S_a^{n+1}$. Applying the induction hypothesis we know that $\exists S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^n \in exec(S_a^{ini})$ such that $S^n \equiv S_a^n$. Therefore, now we have $S^n \rightarrow S^{n+1}$, and $S^n \equiv S_a^n$, hence, applying Lemma Appendix B.2.1 we get that $\exists S_a^{n+1}$ such that $S_a^n \longmapsto S_a^{n+1}$ and $S^{n+1} \equiv S_a^{n+1}$.
    2. We need to prove that for every execution $S_a^{ini} \longmapsto S_a^1 \longmapsto ... \longmapsto S_a^{n+1} \in exec(S_a^{ini})$, $\exists S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \rightarrow S^{n+1} \in exec(S^{ini})$ such that $S^{n+1} \equiv S_a^{n+1}$. Applying the induction hypothesis we know that $\exists S^{ini} \rightarrow S^1 \rightarrow ... \rightarrow S^n \in exec(S^{ini})$ such that $S^n \equiv S_a^n$. Therefore, now we have $S_a^n \longmapsto S_a^{n+1}$, and $S^n \equiv S_a^n$, hence, applying Lemma Appendix B.2.2 we get $\exists S^{n+1}$ such that $S^n \rightarrow S^{n+1}$ and $S^{n+1} \equiv S_a^{n+1}$.

□