Modelling Software-Defined Networks with Alloy*

María-del-Mar Gallardo and Laura Panizo
{gallardo, laurapanizo}@lcc.uma.es

Universidad de Málaga, Andalucía Tech,
Departamento de Lenguajes y Ciencias de la Computación,
Campus de Teatinos s/n, 29071, Málaga, Spain

Abstract. Software defined networks (SDNs) are a recent paradigm to implement flexible networks able to dynamically change the routing rules of data. SDNs present a complex static structure composed of different types of nodes (controllers, switches and hosts), data, links, and rules for transmitting data. In addition, the evolution of a SDN over time is also complicated due to their highly distributed character. Formal methods have proven to be excellent techniques to model, specify and check whether complex software systems as SDNs behave as expected. In this paper, we explore the modelling and analysis of both the static structure and the dynamic behaviour of SDNs using the Alloy formal method.

1 Introduction

Computer networks are evolving towards more agile networks known as Software-defined Networks (SDNs) where the data and control planes are decoupled. On the one hand, the implementation of a SDN has to take into account the complexity of the whole network structure. On the other, it is also necessary to deal with the complex interactions between the different network entities which are usually distributed. Thus, the SDN structure and behaviour should be analyzed against the most critical properties before the network is deployed.

Currently, formal methods have become excellent techniques to model, specify and analyze complex software systems. Each formal method is usually focussed on the modelling and analysis of a particular type of systems and properties. For example, model checking techniques [1] are automatic procedures, very good at locating errors caused by unexpected interactions of processes during the execution of concurrent software as those that may occur during the execution of a SDN. However, they do not behave in the same way when analyzing structurally complex systems due to the well known state-explosion problem. In

* This work has been supported by the Spanish Ministry of Economy and Competitiveness project TIN2015-67083-R and the European Union's Horizon 2020 research and innovation programme under grant agreement No 815178 (5GENESIS)

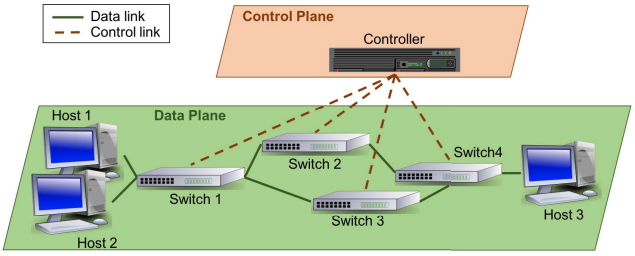

Fig. 1: Example of SDN

contrast, since theorem provers use deductive methods, they are better to analyze complex systems, although they may need the user interaction to proceed at some points during the analysis.

In this paper, we use the Alloy language and tool to model and analyze the two potential sources of SDN errors mentioned above (structural and execution errors). Alloy [2, 4] is a declarative language based on sets and set relations. In addition, it uses first order relational logic to describe properties and refine the models. Although, internally, the core of Alloy use a theorem prover, from the user point of view, it is also a completely automatic tool (similar to a model checker) that generates models correct wrt the specification. The price to pay is that Alloy models are bounded in size, i.e., it is not possible to analyze models of arbitrarily large size. Even though this is an important restriction, in practice, small models are usually sufficient to detect errors in system design.

The article is organized as follows. Section 2 introduces Software-Defined Networks (SDNs). Section 3 presents the Alloy language, covering signatures and relations. Sections 4 and 5 presents the static and dynamic model of a SDN. Section 6 discusses different ways to run the tool. Finally, Section 7 summarizes the conclusions. It is worth noting that the SDN example shown is a modified version of the evaluable experimental project carried out by the students of the "Formal Methods in Software Engineering" subject at the University of Málaga.

2 Description of the problem

A Software-Defined Network [5] (SDN) is a modern networking paradigm that explicitly separates the data and control planes to include intelligence in the network. Figure 1 shows the basic SDN architecture, which is composed of two main kinds of elements: (1) The *Controller* is the core entity of the SDN control plane. The network intelligence is logically centralized in the controller, which is able to dynamically configure the forwarding devices of the data plane in order to achieve a specific goal. (2) *Switches* are data plane components in charge of forwarding the data packets from its source to the destination. In SDNs, each switch has a routing table that contains a set of rules defining how the different incoming packets must be processed (forwarded, discarded, etc.). Finally, *Hosts*

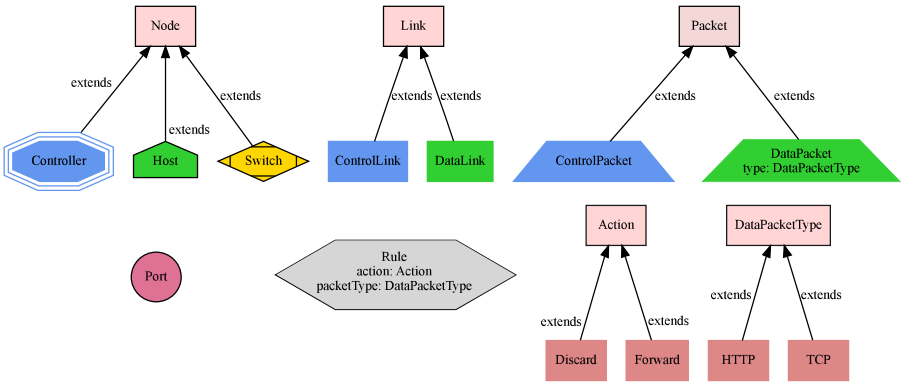

Fig. 2: Meta-model of the SDN model

are the endpoints of the network, they are the source and destination of the data plane traffic. Although hosts are not specific elements of the SDN architecture, in our case study, they are part of the model.

Figure 2 shows the basic components of the SDN model implemented in Section 4. For example, we abstract *hosts*, *switches* and *controllers* as network *nodes* that contain *ports* to connect them with other nodes using port-to-port (bidirectional) *links*. Although it is not explicitly represented in the meta-model, each switch has always an specific link (and thus a port) that connects it with the controller. This connection is mainly used to configure the switches. The data transmitted between nodes are called *packets*. There are two types of packets: control packets include control plane information, such as new rules that must be installed in an specific switch, or a request to know how to process a data packet. Data packets encapsulate information that must be transmitted from one host to another. Data packets also contain the source and destination hosts, the type of data packets, and the current position in the network.

Switches contain *tables* with *rules* that specify how to route data packets. Simplifying, a rule is a structure with a field denoting the type of data packet (e.g. HTTP or FTP) and the input and output ports. The meaning of the rule is as follows. If a data packet of a particular type arrives at port $Port_I$, it must be forwarded through port $Port_O$. In addition, it is also possible to define rules that discard incoming data packets. When a switch has no rule to deal with a data packet, it sends a request to the controller in order to know how to process the packet. Finally, the controller can also send new rules to switches to update the routing tables. This description shows that a SDN has a complex static structure with nodes, links, packages, rules and routing tables. It also has a complex dynamic behaviour when packets move through the network nodes.

In previous work [3], we reviewed the state-of-the-art on SDNs verification. Most of the approaches reviewed focused on the analysis of network invariants (e.g. absence of loops, host reachability, etc.) by means of the analysis of data plane traffic. In the present work, we explore the capabilities of Alloy to gen-

erate valid network topologies (including the data and control plane) whose evolution over time fulfils some desired properties.

3 Background of Alloy

In this section, we give a very simplified presentation of Alloy. In order to make the presentation more intuitive, we will introduce the language using the code of the SDN implementation in Section 4.

The basic constructors of the language are *signatures* and *relations* and *constraints*. *Signatures are sets* whose elements are called *atoms*. For example, `sig Port` in Listing 1.1 defines the *set* of *ports* of the SDN model. Signatures can be `abstract` and, as usual, they cannot have proper elements, but through their extensions. For instance, `abstract sig Link` defines the set of links between ports, but its elements have to belong to one of its extensions `CtrLink` or `DataLink`.

Relations are sets of tuples of the same arity. They are always defined in the context of a signature, which is the type of the first element of the tuples. For example, `abstract sig Link` defines two *binary* relations, `p1` and `p2`, with type `p1, p2`$\subseteq$ `Link` $\times$ `Port`. They are used to associate each link with the ports it connects. For instance, `(L0,P0)` $\in$ `p1` means that the first port of link `L0` is `P0`. By default, the multiplicity of these relations is `one`, i.e., each link has a unique `p1` and `p2` ports. Alloy's relations support other multiplicities such as `lone, some, set`.

`abstract sig DataPacketT` is the set of the type of data packets (elements of non-abstract signatures `TCP` or `HTTP`). The keyword `one` is used to say that there is only one `TCP` and `HTTP` packet type.

Alloy uses a first-order relational logic to define constraints. It also supports the definition of functions to ease the use of these constraints. Listing 1.2 shows some functions used in the SDN model. The `node` function returns the node which a port belongs to. The expression `ports.p` shows the use of the composition operator `.`. Relation `ports`$\subseteq$`Node`$\times$`Port` can be used to obtain the set of ports of a node `n` as `n.ports`, and, also, the nodes to which a port belongs (`ports.p`). Thus, it is possible to compose relations on the left and right sides, which is frequently used in Alloy to express properties. For instance, function `link` returns the links in which a port is placed: `p1.p` and `p2.p` are the sets of links in which `p` is the first and the second port, respectively. Operators `+, &, -` represent the union, intersection and different of sets, respectively. Function `reachableNodes` uses the transitive closure of binary relation `connected`, written as `^connected`, to find all nodes reachable from a given node. Alloy contains quantifiers `all` and `some` to denote the universal and existential quantifiers. In addition, it also provides quantifiers `no` and `one` to simplify the description of some properties. For instance, the constraint 6 in Listing 1.3 establishes that "each control link connects the controller with a switch".

4 Static definition of the SDN structure in Alloy

In this section, we implement an Alloy SDN model using the meta-model shown in Figure 2. Thus, the Alloy model contains three abstract classes `Node`, `Link` and `Packet`, that represent the main SDN entities, and some non-abstract extensions of these classes, such as `Controller`, `Switch` or `Host`. In addition, model contains two abstract classes that define enumerated types: `Action` that describes how a switch processes an incoming data packet (`Forward` or `Discard`), and `DataPacketT`, that represents the type of a data packet (`HTTP` or `TCP`).

4.1 Signatures and relations

We start by defining the SDN components in Alloy. Listings 1.1 and 1.2 contain the definition of the main structural actors of SDNs: `abstract sig Node` is the abstract class, and `Host`, `Switch` and `Controller` are extensions. The abstract signature embodies two binary relations shared by these components, `ports` and `connected`, that, respectively, associate each node with its ports and with all the nodes directly connected with it (excluding the `Controller`). The multiplicity of relations is `some`, meaning that all nodes have at least one port and are at least connected with one node. The specification establishes that there is only `one` `Controller` in the network. A `Host` has two relations `oBuffer` and `iBuffer`, which contain the data packets to be sent or received, respectively. Finally, a `Switch` has a `table` relation that stores a set of rules.

```
sig Port{}

abstract sig Link{
 p1,p2: Port
}
sig CtrLink extends Link{}
sig DataLink extends Link{}

abstract sig DataPacketT{}
one sig TCP extends DataPacketT{}
one sig HTTP extends DataPacketT{}

abstract sig Node{
  ports: some Port,
  connected: some Node
}
one sig Controller extends Node{}
sig Host extends Node{
  iBuffer: set DataPacket,
  oBuffer: set DataPacket
}
sig Switch extends Node{
  table: set Rule
}

abstract sig Action{}
one sig Forward extends Action{}
one sig Discard extends Action{}

sig Rule{
  packetType:DataPacketT,
  iPort: Port,
  action: Action,
  oPort : lone Port
}
```

Listing 1.1: Signatures-relations I

```
abstract sig Packet{
  position: lone Port
}
sig DataPacket  extends Packet{
  type: DataPacketT,
  src,dest:Host
}
sig CtrPacket extends Packet{
  newRule: lone Rule,
  request: lone DataPacket
}
/*Functions*/
fun node(p:Port):Node{ ports.p }
fun link(p:Port):Link{ p1.p + p2.p }

fun nodeLinks(n:Node):Link{
  {l:Link| some l.(p1+p2) & n.ports}
}

fun dataLinks(n:Node): Link{
  nodeLinks[n] & DataLink
}

fun CtrLinks(n:Node): Link{
  nodeLinks[n] & CtrLink
}

fun reachableNodes(n:Node):Node{
  n.^connected
}

fun remotePort(p:Port): Port{
  {pR: Port - p |
     one (pR & link[p].(p1 + p2))}
}
```

Listing 1.2: Signatures-relations II

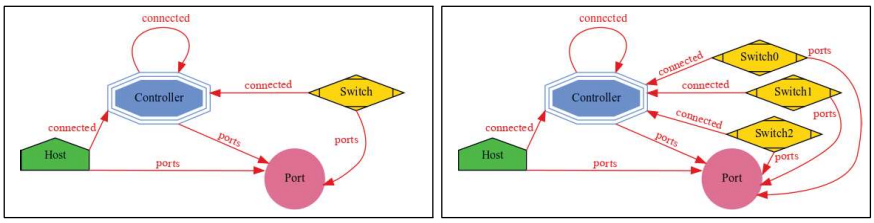

Fig. 3: Examples of instances of the ALLOY model

`abstract sig Packet` defines the packets that move through the network. Relation `position` associates each packet with its position, which is the port of some node. The `lone` multiplicity means that a packet could not be at any port (e.g. discarded packets, or packets in the input/output buffers of a `Host`). This abstract signature has two extensions: `sig DataPacket` and `sig CtrPacket`. On the one hand, `DataPacket` contains the packets carrying user data. It has three binary relations: `type` associates the packet with the data packet type (`HTML` or `TCP`), `src` and `dest` that relate the packet with the source and destination hosts. On the other, `sig CtrPacket` includes the set of packets transmitted between the controller and switches. It contains two relations: `newRule` used by the controller to update with a new rule the routing table of a switch, and `request` used by switches to ask the controller what to do with a data packet.

The abstract signature `Action` and its extensions `Forward` and `Discard` define the enumerated type with the possible actions to be carried out by switches with data packets: forward them to another node or discard them. `sig Rule` defines the forwarding rules. Each rule applies to a specific packet type (`packetType`), to a specific input port through which the packet has to arrive (`iPort`), and the action to be realized with the packet `action`). When the action is `Forward`, relation `oPort` contains the port through which the packet has to be forwarded.

Once a model is constructed, ALLOY can generated model instances according to the constraints defined. Figure 3 shows two instances of the ALLOY model of Listings 1.1 and 1.2. Both figures have been simplified by hiding links, packets and rules. It is easy to observe that these instances do not match the expected structure of a SDN. They contain several errors: the `Port` is shared by different nodes, the `Controller` is connected with itself, `Host` is directly connected to the `Controller`, and so on. In the next section, we enrich the model with constraints that delimit the expected topology of SDN.

4.2 Adding constraints to the Alloy SDN model

We now add constraints to generate structurally correct SDNs. Constraints are added as `facts` to models. All model instances have to satisfy all constraints in facts. Listings 1.3 and 1.4 show the 32 constraints that define the static behaviour of a SDN. In the rest of the section, we briefly describe some of them.

Fact 1 imposes that the two ports of a link have to be different. In Alloy, signatures and relations are dealt with as sets. Thus, `l.p1!=l.p2` asserts that sets `l.p1` and `l.p2` are different. But, since binary relations `p1` and `p2` are defined with multiplicity `one`, `l.p1` and `l.p2` are singleton sets.

Fact 2 uses function `node` to assert that each port must belong to a unique node. Observe that, in this case, `one` is used to indicate that set `node[p]` is a singleton. Fact 3 is similar except for the use of `lone`, which means that for each port `p`, the set `link[p]` must have at most one element. The fourth fact asserts that each link connects two different nodes.

Fact 8 establishes that the `Controller` has only control links. Word `in` is the subset operator. Fact 10 asserts that every `Switch` has at least two `DataLink`. In this case, we have used operator `#` that returns the size of the corresponding set. Observe that all these constraints are needed since Alloy is free to create relations among atoms whenever no constraint prohibits them.

Facts 11 to 14 contain facts to avoid the existence of isolated nodes in the model instances, and facts 15 and 16 simplify the instances generated by the model. On the one hand, fact 15 establishes that each port stores at most a packet. On the other, fact 16 says that data packets are placed in a port or in an input/output host buffer. Observe that the use of multiplicity `one` makes both the disjunction `or` and the union `+` exclusive, that is, each data packet has to be at a port, or else at an input buffer, or else at an output buffer.

Facts 17 and 18 refer to the content of data links. For example, fact 17 imposes that the packets at the extreme of a data link have to be data packets. Expression `l.(p1+p2)[position]` is equivalent to `position.(l.(p1+p2))` that returns the data placed at the extremes `p1` and `p2` of link `l`.

Facts 19 to 25 refer to data and control packets. Fact 19 says that the source and destination hosts have to be different. Fact 20 says that a control pack can have a `newRule` (when the `Controller` is communicating with a `Switch` to update its routing table), or a `request` (when a `Switch` is asking to the `Controller` what to do with a given data packet). Observe that the use of `one` makes it impossible for a control packet to have both a `newRule` and a `request`. The two following facts (21 and 22) establish that the direction of control data interchanged between the `Controller` and the `Switches` is correct. Thus, when a control packet is placed at a switch port the packet must contain a `newRule`. Inversely, when the control packet is at a controller port, it must contain a `request`. Facts 23 and 24 affirm that packets in host buffers are correct. Thus, fact 23 says that data packets placed at the input buffer of a host `h` cannot be placed at any port, and that their destination host must be `h`. Fact 24 is similar. Finally, fact 25 imposes that the data packets at an output buffer of a host `h` must have as destination another host reachable from `h` through data links and intermediate switches.

Facts 26 to 32 describe the rules and consistency of the routing tables. For instance, fact 27 affirms that routing tables have at most a rule for each port and packet type (i.e. routing tables should not contain repeated or contradictory rules). Observe that expression `all s:Switch, disj r1,r2:s.table` is an universal

```
fact LinksAndNodes{
//1-the ending ports of any link are different
 all l:Link| l.p1!=l.p2
//2- each port belongs to a node
 all p:Port| one node[p]
//3-each port belongs at most to a link
 all p:Port| lone link[p]
// 4- The ports of each link belong to different nodes
 all l:Link| node[l.p1]!=node[l.p2]
//5-connected is well defined
 all n:Node| n.connected = {m:Node-Controller| some l:Link|
     node[l.(p1+p2)] = n+m}
//6-Control links connect switches and Controller
 all l:Link| l in CtrLink implies one node[l.(p1+p2)] & Controller and
      one node[l.(p1+p2)] & Switch
//7-Data links connect two switches or a switch and a host
 all l:Link| l in DataLink implies some node[l.(p1+p2)] & Switch and
      Controller not in node[l.(p1+p2)]
//8-all controller links are control links
 nodeLinks[Controller] in CtrLink
//9-the controller has exactly a link to each switch
 all s:Switch| one nodeLinks[Controller] & nodeLinks[s]
//10-switches have at least two data links
 all s:Switch| #nodeLinks[s] & DataLink >=2
//11-Switches can at least reach two hosts
 all s:Switch| #(reachableNodes[s]&Host)>=2
//12-Switches have at least two nodes connected
 all s:Switch| #s.connected >=2
//13-Hosts do not have links to the Controller
 no nodeLinks[Controller]&nodeLinks[Host]
//14-Each host has only one link to a switch
  all h:Host| one s:Switch| h.connected = s
}
fact DataPackets{
//15-at most one packet in a port
 all p: Port| lone position.p
//16-data packets are well placed at ports or buffer hosts
 all pk:DataPacket| one pk.position or one (iBuffer+oBuffer).pk
}
fact DataControlLinks{
//17- DataLink contains only DataPackets
 all l:DataLink| l.(p1+p2)[position] in DataPacket
//18- CtrLinks contains only CtrPackets
 all l:CtrLink| l.(p1+p2)[position] in CtrPacket
}
fact Packets{
//19-The source and destination hosts of a DataPacket are different
 all pk:DataPacket| pk.src != pk.dest
//20-Each control packet has a new rule or a request
 all pk:CtrPacket| one pk.(newRule + request)
//21-A control Packet with a new rule arrives to a Switch port
 all pk:CtrPacket| (one pk.position & Switch.ports) implies one pk.newRule
//22-A controlPacket with a request arrives to a Controller port
 all pk:CtrPacket|(one pk.position & Controller.ports) implies
                  one pk.request
//23-iBuffer is well defined
 all h:Host| h.iBuffer in {pk:DataPacket| (no pk.position) and pk.dest= h}
//24-oBuffer is well defined
 all h:Host| h.oBuffer in {pk:DataPacket| (no pk.position) and pk.src= h}
//25-Packets in an oBuffer can reach the destination host
 all h:Host, pack:h.iBuffer| pack.dest in reachableNodes[h]
}
```

Listing 1.3: Facts on links nodes, and packets

```
fact RulesAndTables{
//26-Each rule belongs to a table at most
 all r:Rule | lone table.r
//27-Each switch has at most one rule for each data type and input port
 all s:Switch, disj r1,r2:s.table| r1.packetType != r2.packetType or
                                   r1.iPort != r2.iPort
//28-The input port of a rule belongs to a switch and a data link
all r:Rule | one (r.iPort & Switch.ports) and
             one (r.iPort & dataLinks[Switch].(p1+p2))
//29-Forward rules has output port
all r:Rule | one r.oPort  iff  r.action = Forward and
             one (r.oPort & dataLinks[Switch].(p1+p2))
//30-Discard rules has no output port
all r:Rule | #r.oPort =0 iff  r.action = Discard
//31-The input and output ports of a rule are different
 all r:Rule |  r.iPort != r.oPort
//32-The input and output ports of a rule in a switch belong to the switch
 all s:Switch, r:s.table| node[r.(iPort + oPort)] = s
}
```

Listing 1.4: Facts on rules and routing tables

quantification on all switches, and all rules in their tables. Word `disj` is used to indicate rules `r1, r2` have to be *different*. Finally, observe that the domain for rules is `s.table`, that is, the set rules in the routing table of the switch given by the `all` quantifier. Fact 31 says that the input and output ports of any rule have to be different. Observe that condition `r.iPort != r.oPort` includes the case when `r` does not have an output port, that is, `r.oPort` is empty.

As it can be seen, the construction of a Alloy model is not a trivial task, and requires skills in the use of sets, relations and first-order logic. Sometimes, when a new constraint is added, the model becomes inconsistent, and the developer has to find the part of the model that does not match the new constraint. Each Alloy model can be analyzed by the Alloy tool, and if the model is consistent, the tool generates different model instances. Figure 4 shows an instance of the SDN model that satisfies all the facts of this section. The instance shows a network topology composed of the controller, two switches and two hosts.

5 Definition of the SDN dynamics

The SDN described in the previous section is *static*. For instance, given a packet on switch port, the model does not specify how the packet position changes to reach the destination host. The goal of this section is to transform the static SDN into a dynamic one. To this end, we have to carry out the following steps:

1. Add a new signature `Time` to represent different time instants in the model.
2. Identify the model relations that could change over time (as a result of some action), and extend their definitions adding a `Time` component.
3. Modify accordingly the facts that deal with the relations extended with time.
4. Define predicates that implement the actions that can change the model.
5. Define predicates (called frame conditions) that specify the part of the model that does not change when an action is carried out.

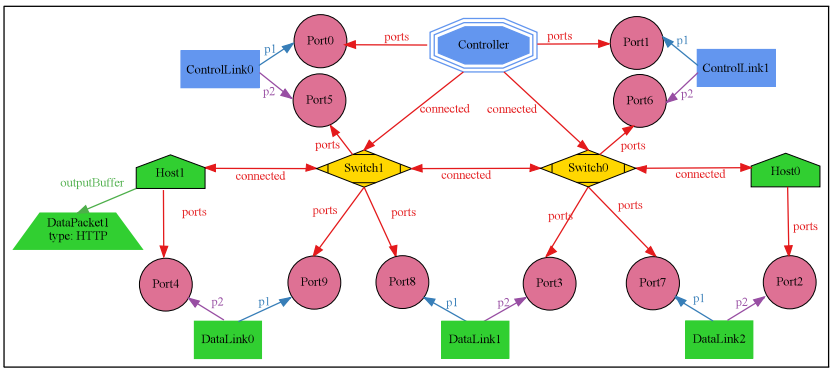

Fig. 4: Instance of the ALLOY model

```
open util/ordering[Time]
sig Time{}

one sig Controller extends Node{}
sig Host extends Node{
  iBuffer: DataPacket set -> Time,
  oBuffer: DataPacket set -> Time
}
sig Switch extends Node{
  table: Rule set -> Time
}
abstract sig Packet{
  position: Port lone -> Time
}
```

Listing 1.5: Signatures and relations extended with time

5.1 Extension of relations with Time

Listing 1.5 introduces the new signature `Time` and transforms binary relations `position`, `iBuffer`, `oBuffer` and `table` into ternary adding a new time component. These are the only relations whose values could change during the evolution of a SDN. In the model, operator `->` is the cartesian product of sets. Now, the elements of each of these relations are 3-tuples. For instance, `(Packet1,Port0,Time0)`, `(Packet1,Port1,Time1)` could be elements `position`.

The use of the time component in tuples allows a packet to be on different ports at different time instants. Thus, `(Packet1,Port0,Time0)` and `(Packet1,Port1,Time1)` mean that `Packet1` is placed at port `Port0` in time instant `Time0`, but it is in `Port1` in `Time1`. The extension of the other relations work similarly. Thus, the new ternary relation `table` allows the routing tables to change over time, and so on. Line `open util/ordering[Time]` is used to import the ALLOY library `ordering` that gives a total order to atoms of `Time`.

```
//15-at most one packet in a port
 all t:Time, p: Port| lone (position.t).p
//16-data packets are well placed at ports or buffer hosts
 all t:Time, pk:DataPacket| (one pk.position.t) or
        (one(iBuffer+oBuffer).t.pk)
//17- DataLink contains only DataPackets
 all t:Time,  l:DataLink | l.(p1+p2)[position.t] in DataPacket
//18- CtrLinks contains only CtrPackets
 all t:Time, l:CtrLink |l.(p1+p2) [position.t] in CtrPacket
//21- A CtrPacket with a new rule arrives to a Switch port
 all t:Time, pack:CtrPacket|
        (one pk.position.t & Switch.ports) implies one pk.newRule
//22- A CtrPacket with a request arrives to a Controller port
 all t:Time, pk:CtrPacket|
         (one pk.position.t & Controller.ports) implies one pk.request
//23- iBuffer is well defined
 all t:Time, h:Host| h.iBuffer.t in {pk:DataPacket|
         (no pk.position.t) and pk.dest = h}
//24- oBuffer is well defined
  all t:Time, h:Host| h.oBuffer.t in {pk:DataPacket|
          (no pk.position.t) and pk.src = h}
//25- Packets in an oBuffer can reach the destination host
  all t:Time, h:Host, pk:h.iBuffer.t | pk.dest in reachableNodes[h]
//26-Each rule belongs to a table at most
  all t:Time, r:Rule | lone table.t.r
//27-Each switch has at most one rule for each data type and iPort
  all t:Time, s:Switch, disj r1,r2:s.table.t|
       r1.packetType != r2.packetType or r1.iPort != r2.iPort
//32-The iPort and oPort of the rules of a switch belongs to the switch
  all t:Time, s:Switch, r:s.table.t| node[r.(iPort + oPort)] = s
}
```

Listing 1.6: Facts extended with time

5.2 Introducing time in facts

Listing 1.6 contains the facts of the static model (see Section 4) that have to be modified taking the `Time` component into account. Observe that each fact is universally quantified with respect to `Time`. This means that the fact has to be true at each time instant. In addition, the time variable is added at the new ternary relations. For instance, fact 15 says that each port may have at most a packet at each time instant. The time component is composed by the right side of the relation (as in `position.t`) because the time component is the last one in the definition of relations. Observe that some facts are now a bit more complicated, but the process of adding time to facts is quite automatic.

5.3 Definition of transitions

The following step to construct a dynamic model is to implement *predicates* that define the model transitions. The header of predicates usually contains two time parameters `t` and `t'` that denote the time instants before and after the predicate is executed. An Alloy predicate is a sequence of constraints that are added to the model only when the predicate is executed. Predicates must contain three blocks of constraints: pre, post and frame conditions. The meaning of pre

```
pred discardPacket(t,t':Time,s:Switch,pk:DataPacket){
 //pre
 some pk.position.t & s.ports
 some r:s.table.t | r.action = Discard and r.iPort = pk.position.t and
                    r.packetType = pk.type
 //post
 pk.position.t' = none
 //frame
 tablesUnmodifiedExc[none,t,t']  and packetsUnmodifiedExc[pk,t,t']
 oBuffersUnmodifiedExc[none,t,t'] and iBuffersUnmodifiedExc[none,t,t']
}
pred forwardPacket(t,t':Time,s:Switch,pk:DataPacket){
  //pre
  some pk.position.t & s.ports
  some r:s.table.t | r.packetType = pk.type and r.action = Forward and
                     r.iPort = pk.position.t
  //post
  let r = s.table.t & packetType.(pk.type) & action.Forward
                                           & iPort.(pk.position.t),
      p = remotePort[r.oPort] | pk.position.t' = p
  //frame
  tablesUnmodifiedExc[none,t,t'] and packetsUnmodifiedExc[pk,t,t']
  oBuffersUnmodifiedExc[none,t,t'] and iBuffersUnmodifiedExc[none,t,t']
}
```

Listing 1.7: Predicates to discard and forward data packets in switches

and post conditions is the usual one. Frame conditions represent the part of the model that is not changed by the predicate. Frame conditions are essential in Alloy, since the tool is free to modify any timed relation if the model does not specify that it must not do it. Now, we list the predicates defined in the SDN model to make packets and rules move during the network execution.

Listing 1.7 shows the conditions for a switch to discard a packet. The preconditions are that, at instant `t`, the packet `pk` must be at a port of `s` and that `s` has a rule in its routing table telling that packets arriving through this port must be discarded. Post condition is that the packet `pk` has no position at instant `t'`. The keyword `none` represents the empty set. The frame conditions are four predicates that establish that the only network component that changes during the `discardPacket` transition is `pk`. The implementation of frame conditions is given Listing 1.9. Each frame predicate corresponds to a relation that may change over time.

Listing 1.7 also contains the implementation of the predicate that forwards a packet from a switch using a rule. The preconditions are: at the time instant `t`, (1) `pk` is located at a port of `s`, and (2) there is a rule `r` in the routing table of the `s` with action `Forward` that applies to packets with the packet type of `pk` and input port the port in which `pk` is placed. Expression `let` is used to define bound variables that simplify the constraints. Thus, variable `r` represents the rule of the routing table to be applied in the predicate, and `p` is the remote port to which the packet is forwarded. Using these two variables, the postcondition establishes that the position of `pack`, at time instant `t'`, is `p`.

Listing 1.8 contains the predicate `sendPacket` and `receivePacket`. The first one specifies how a host `h` sends packet `pk`. The precondition is that `pk` is at

```
pred sendPacket(t,t':Time, h:Host, pack:DataPacket){
  //pre
  some pack & h.oBuffer.t
  //post
  some p':remotePort[h.ports] | pack.position.t'=p'
  h.oBuffer.t' = h.oBuffer.t - pack
  //frame
  tablesUnmodifiedExcnone,t,t'] and packetsUnmodifiedExc[pack,t,t']
  oBuffersUnmodifiedExc[h,t,t'] and iBuffersUnmodifiedExc[none,t,t']
}
pred receivePacket(t,t':Time,h:Host,pack:DataPacket){
  //pre
  some (pack.position.t & h.ports)
  //post
  h.iBuffer.t' = h.iBuffer.t + pack
  pack.position.t'  = none
  //frame
  tablesUnmodifiedExc[none,t,t'] and packetsUnmodifiedExc[pack,t,t']
  oBuffersUnmodifiedExc[none,t,t'] and iBuffersUnmodifiedExc[h,t,t']
}
```

Listing 1.8: Predicates to send and receive data packets on hosts

```
pred packetsUnmodifiedExc(pp:set Packet, t,t':Time){
   all pk:Packet-pp | pk.position.t = pk.position.t'
}
pred oBuffersUnmodifiedExc(hh:set Host, t,t':Time){
   all h:Host-hh | h.oBuffer.t = h.oBuffer.t'
}
pred iBuffersUnmodifiedExc(hh:set Host, t,t':Time){
   all h:Host-hh | h.iBuffer.t = h.iBuffer.t'
}
pred TablesUnmodifiedExc(ss: set Switch, t,t':Time){
    all s:Switch - ss | s.table.t = s.table.t'
}
```

Listing 1.9: Frame conditions

the output buffer of `h` at instant `t`. The predicate has two postconditions: at time instant `t'`, (1) `pk` is at the remote port of the link that connects `h` with the SDN, and (2) the output buffer of `h` is equal to the output buffer at `t` excluding `pk`. The predicate `receivePacket` defines how a host `h` receives a packet `pk`. The precondition is that `pk` is at a port of `h` at instant `t`. The postconditions are that (1) `pk` is at the input buffer of `h`, and (2) `pk` is at no port, at time instant `t'`.

The model also contains the following predicates:

- `pred receiveRequest(t,t':Time, ctrl:Controller, req:CtrPacket)`, the controller receives a request from a switch to know how to manage a data packet.
- `pred sendRequest(t,t':Time, s:Switch, pk:DataPacket)`, switch `s` sends a request to the controllr to know how to manage the data packet `pk`.
- `pred installRule (t,t':Time, s:Switch, pk:CtrPacket)`, switch `s` receives a control packet with a rule and installs in its routing table.

```
pred Config1(){
 #Controller = 1 and #Host = 2 and #Switch>=2 and #DataPacket = 2
 all p:Port| one link[p]   //all ports are in a link
 //all rules are different in the system/topology
 no disj r1,r2:Rule| r1.iPort = r2.iPort and r1.oPort = r2.oPort
        and r1.action = r2.action and r1.packetType = r2.packetType
}
run Config1 for 10
```

Listing 1.10: Alloy predicates for configuration 1

6 Generation of Instances

Alloy can run predicates and check assertions. In the first case, it finds out whether the predicate is consistent and if so, it returns model instances that satisfy the constraints (facts) and the predicate. These instances are very useful to detect misunderstanding in the specification of the model. In the second case, Alloy looks for counterexamples that satisfy the model specification but not the assertion. In both cases, the Alloy model is transformed into a set of boolean formulae that are analysed using a SAT solver.

In this section, we use the run and check approaches to generate instances of the SDN model. First, we run different predicates, that we call *configurations*, that constraint the topology (number of nodes controllers, hosts, and switches), the traffic (number of packets), or the initial status of the routing tables in the static and dynamic model. These predicates can be used to model *non-deterministic* execution traces of the dynamic model. Then, we check some assertions that prove properties of our model. If the properties are not satisfied, Alloy returns a counterexample. We have defined two different configurations:

- `Config1`, in Listing 1.10, is targeted to produce instances of the static model with one controller, two hosts, two or more switches, and two data packets. In addition, all ports must be in a link and all rules are different. The keyword *run* instructs Alloy to execute `Config1` with at most 10 atoms per signature.
- `Config2`, Listing 1.11, is targeted to generate instances of the dynamic model in which a data packet is forwarded from the source to the destination host. It extends configuration 1 with the state of the network at the first time instant. The data packets are in the output buffer of the source host and the switches' tables have pre-installed all the necessary forwarding rules. In addition, the predicate defines how the instances can non-deterministically evolve over time. It is simulated with at most 10 atoms and 9 time instants.

Alloy reports the number of variables and primary variables, clauses, the time to transform the model into clauses, and the execution time of the SAT solver. We have run Alloy 4.2 in a MacBook Air Core i5 with 4GB of RAM. By default, Alloy uses 768MB of memory and 8192kB of stack size. Table 1 summarizes the results and the average execution time of the default SAT solver, the SAT4J solver, calculated on 20 executions of each configuration. Figure 5 shows a simplified view of a network topology satisfying `Config1`. To ease the

```
pred initTopology(t:Time){
 #Controller = 1 and #Host = 2 and #Switch>=2 and #DataPacket = 2
//all DataPackets are initially in the oBuffers of the hosts
 all pk:DataPacket | pk in Host.oBuffer.t
//No CtrPackets in switches or controller ports in T0
 all pk:CtrPacket| pk.position.t= none
//all ports are in a link
 all p:Port| one link[p]
//all switches has some pre-installed rules
 all s:Switch| some s.table.t
//all forwarding rules are pre-installed
 all r:Rule | r.action = Forward implies r in Switch.table.t
//all rules are different in the  system
 no disj r1,r2:Rule | r1.iPort = r2.iPort and  r1.oPort = r2.oPort
        and (r1.action = r2.action) and (r1.packetType = r2.packetType)
}
pred Config2(){
 initTopology[first]
 all t:Time-last| let t'= next[t]| ((some h:Host, s:Switch, pk:DataPacket,
                  ctrl:Controller, ctrPk:CtrPacket|
   sendPacket[t,t',h,pk] or receivePacket[t,t',h,pk] or
   forwardPacket[t,t',s,pk] or discardPacket[t,t',s,pk] or
   installRule[t,t',s,ctrPk] or sendRequest[t,t',s,pk] or
   receiveRequest[t,t',ctrl,ctrPk]))
}
run Config2 for 10 but 9 Time
```

Listing 1.11: Alloy predicates for configuration 2

inspection of the results, Alloy visualizer can project instances on a given signature (such as `Time`). Figure 6 shows an instance of `Config2` in which we can observe how the data packets change their position from time `Time$0` to `Time$5`.

Alloy can also check assertions to automatically determine whether all instances satisfy a property. Listing 1.12 contains two desired properties for the dynamic model given by `Config2`. Property "for all time instants, there is no control packet in the controller (input) ports" should hold, since switches should not send requests to the controller (all rules were initially pre-installed). Property "data packets should go from the source to the destination hosts" should also hold. In both cases, Alloy concludes that the assertion is never violated.

	Config1	Config2
Vars	36,368	170,110
Primary vars	1,488	4,799
Clauses	97,823	346,270
Exec. time(ms)	306	4,958

Table 1: Alloy report

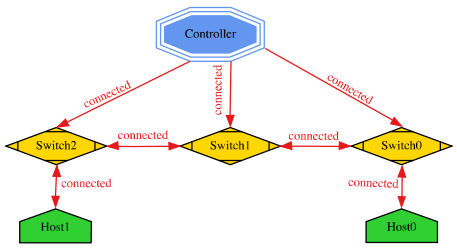

Fig. 5: Network topology

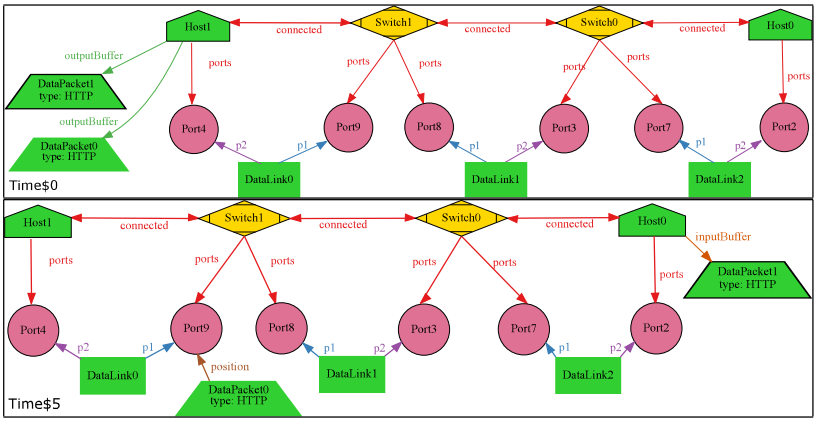

Fig. 6: Evolution over time of data packets

7 Conclusions and future work

In this paper, we have used Alloy to describe the structure and dynamic behaviour of SDN. Alloy is a declarative language based on sets, relations and first-order logic, and a tool for the formal description and analysis of complex systems. SDNs constitute a non-trivial example that shows the power of Alloy for the specification of software systems. On the one hand, it is a structurally complex system containing several types of objects and relations. Alloy proves to be an excellent language for the modelling of the static component of software. On the other, the dynamics of SDNs involves a series of operations to install and uninstall rules on switches, and to distribute data and control packets through the network. The concurrent and distributed execution of these operations may entail many safety and liveness errors. This dynamics can also be implemented and simulated in Alloy. Although the model developed in this paper is quite complete, it could be extended to incorporate other SDN desired properties. For instance, we can define new predicates that model SDN apps; that is, applications that run concurrently on the controller, which dynamically decides how to configure the data plane to achieve different objectives.

```
//Controller does not receive requests
assert noRequest{ Config2 implies all t:Time| no pk:CtrPacket|
        one (pk.position.t & Controller.ports)}
check noRequest for 10 but 9 Time
//all data packets arrives to the destination host
assert packetArrival{ Config2 implies (some disj t1, t2:Time|
         all pk:DataPacket| (one pk & pk.src.oBuffer.t1) and
                            (one pk & pk.dest.iBuffer.t2)}
check packetArrival for 10  but 9 Time
```

Listing 1.12: Assertions

Bibliography

[1] E. M. Clarke, O. Grumberg & D. A. Peled (2000): *Model Checking*. The MIT Press.

[2] Daniel Jackson (2006): *Software Abstractions - Logic, Language, and Analysis*. MIT Press. Available at `http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&#38;tid=10928`.

[3] L. Lavado, L. Panizo, M.M. Gallardo & Pedro Merino (2017): *A Characterisation of verification tools for software defined networks*. *Journal of Reliable Intelligent Environments* 3(3), pp. 189–207, https://doi.org/https://doi.org/10.1007/s40860-017-0045-y.

[4] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang & Daniel Jackson (2017): *Alloy*: a general-purpose higher-order relational constraint solver*. *Formal Methods in System Design*, pp. 1–32, https://doi.org/10.1007/s10703-016-0267-2.

[5] B. A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka & T. Turletti (2014): *A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks*. *IEEE Communications Surveys Tutorials* 16(3), pp. 1617–1634, https://doi.org/10.1109/SURV.2014.012214.00180.