# Modelling and Specifying Software Systems with Alloy* (Tutorial)

María-del-Mar Gallardo and Laura Panizo
Universidad de Málaga, Andalucía Tech,
Departamento de Lenguajes y Ciencias de la Computación,
Campus de Teatinos s/n, 29071, Málaga, Spain
{gallardo, laurapanizo}@lcc.uma.es

ALLOY is both a modelling and specifying language of software systems, and a tool to automatically check the consistency of descriptions. ALLOY uses sets and set relations to describe the high-level structure of systems, and a first-order relational logic to enrich models with more specific properties. A similar methodology is followed by software engineers when using OCL to complete the UML descriptions with more detailed features that improve the specifications. ALLOY is able to construct and analyze the consistency of *static* specifications of complex systems, producing snapshots of possible model instances. In addition, ALLOY can also describe *dynamic* specifications that show how the system may evolve over time when executing transitions. In this tutorial, we present the main features of the language and illustrate them by means of an example consisting in the specification and modelling of Software-Defined Networks (SDNs). The example is sufficiently complex to permit the use of the most significant ALLOY constructors.

## 1 Introduction

The use of formal methods is increasingly becoming an essential task during the design and development phases of complex software to ensure its correctness, at least, with respect to the more critical properties. The use of a specific formal method usually consists of three activities: (1) modelling the system to be analyzed, (2) specifying the properties desirable on the system, (3) using some (semi-) automatic tool that supports the developer who is carrying out the verification work. The model of the system should be an over-approximation in such a way that the properties proved on the abstract model can be transferred to the original system. The use of mathematical-based tools allows formal methods to guarantee correctness, and helps developers to find errors as soon as possible.

Although there are many different formal methods aimed at analyzing diverse aspects of software systems, in this tutorial, we classify them into two groups. First, the approaches that exhaustively explore the state space generated by the system using model checking techniques. Second, the methods that utilize deductive theorem provers to decide on the system correctness using SAT or SMT solvers.

Model checking techniques [1] are very good at locating errors caused by unexpected interactions of processes during the execution of concurrent software. These type of errors are very hard to detect by manual procedures and, in this context, model checkers become essential to guarantee correctness. However, model checking techniques are not suitable to analyze structurally complex systems composed of several interrelated components. In these cases, the systems produce a huge state space, and model checking tools quickly run out of memory. For this type of systems, it is more reasonable to use deductive methods such as theorem provers. These approaches also have drawbacks such as they are not completely

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

Submitted to:
PROLE 2019

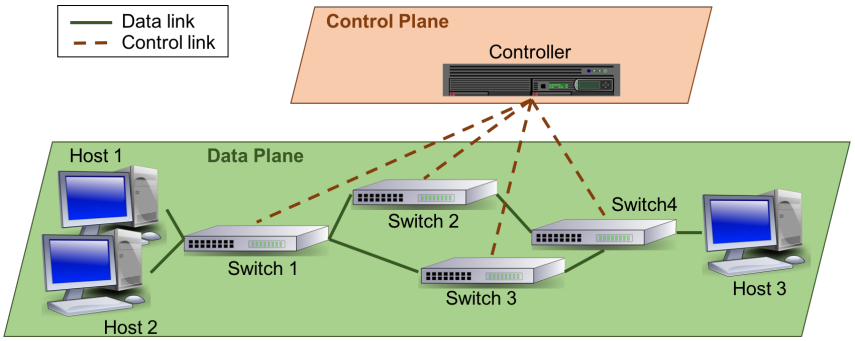


Figure 1: Example of SDN

automatic, that is, sometimes they need the user help to proceed the deductive reasoning. In addition, these tools are sometimes hard to manage, which prevents their use for non-expert developers.

ALLOY [2, 4] has reached a good compromise between the construction of a completely automatic tool, following the model checking philosophy, and the capacity of describing complex systems with non-trivial state spaces as theorem provers do. The core of ALLOY is a theorem prover but its use is transparent to the developer who can use the tool without dealing directly with the complexity of the constraint solvers. The price to pay is that the 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.

ALLOY is a declarative language. The user describes the system using sets, relations and logic formulae and, based on them, ALLOY generates different instances satisfying the specification. If the user finds an instance that does not match the expected behaviour, or if the tool does not find any model instance due to the specification inconsistency, she/he can modify the specification to correct the error. The construction of a model using ALLOY consists of two phases. The first one is to build a static model with the structure of the system. In the second phase, the static model is extended in order to specify how the system changes over time.

In this paper, we show the ALLOY language and tool. The article is organized as follows. Section 2 introduces Software-Defined Networks (SDNs), which is the example that will guide the tutorial. Section 3 presents the ALLOY language, covering signatures, relations, and the use of logic formulae to correctly describe the models. All these language elements are used to model a static view of SDNs. Section 4 extends relations with time and presents some operations to be carried out by the SDNs. Section 5 discusses different ways to run the tool. Finally, Section 6 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 dynam-

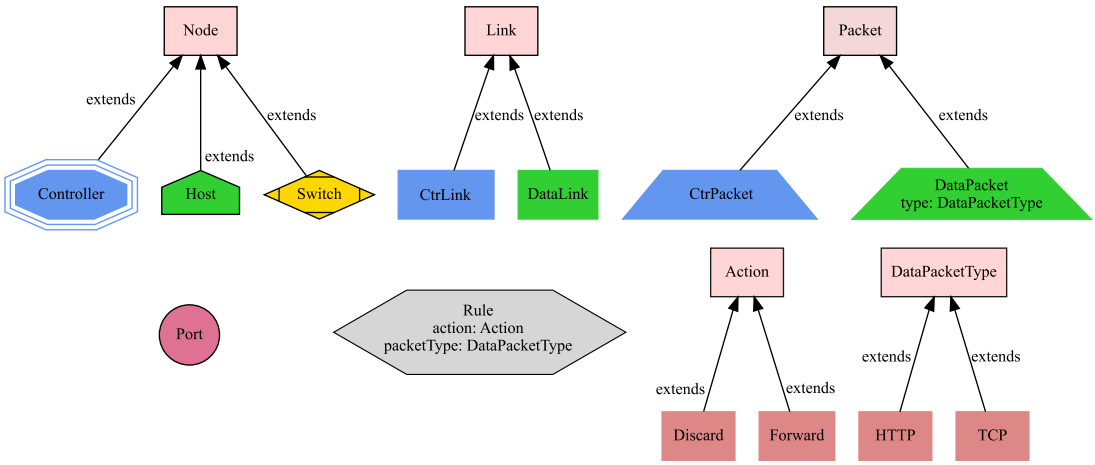


Figure 2: Meta-model of the SDN model

ically 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* 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.

The model of a SDN must only include the relevant information. Figure 2 shows the basic components of the SDN model that we will implement in Section 3. 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 a 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. In this case, the relevant information are 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 tutorial, we want to explore the capabilities of ALLOY to generate valid network topologies (including the data and control plane) whose evolution over time fulfills some desired properties.

# 3 Static definition of the SDN structure in ALLOY

In this section, we introduce the ALLOY specification language. In order to make the presentation more intuitive, we will implement an ALLOY SDN model based on the meta-model shown in Figure 2. The meta-model contains three abstract classes Node, Link and Packet that represent the main system actors. The model also includes concrete classes, such as Controller, Switch or Host, that extend these abstract classes. In addition, the meta-model contains two abstract classes that define enumerate types: Action that describes how a switch processes an incoming data packet (Forward or Discard), and DataPacketType, that represents the type of a data packet (HTTP or TCP). It is clear that we could consider more elements in these classes.

## 3.1 Signatures and relations

We start by defining the SDN components in ALLOY.

```
sig Port{}
abstract sig Link{
   p1,p2: Port
}

sig CtrLink extends Link{}
sig DataLink extends Link{}
```

Listing 1: Signatures and relations I

```
abstract sig DataPacketType{}
one sig TCP extends DataPacketType{}
one sig HTTP extends DataPacketType{}
abstract sig Node{
   ports: some Port,
   connected: some Node
}
```

Listing 2: Signatures and relations II

Listing 1 shows the most common constructors of the language: *signatures* and *relations*. In ALLOY, *signatures are sets* and constitute the primordial component of the language. The elements of signatures are called *atoms*. In the example, `sig Port` defines the *set* of *ports* of the model. As usual, abstract signatures 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. The extensions of an abstract signature constitute a partition of the original abstract signature; that is, links must be exclusively `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`. Their type is `p1, p2`$\subseteq$ `Link` $\times$ `Port` and 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`, as it will be shown later.

Listing 2 shows how to define enumerated types in ALLOY. `abstract sig DataPacketType` defines the type of data packets as elements of signatures `TCP` or `HTTP`. The keyword `one` defines the multiplicity of the sets limiting to one the number of atoms in sets `TCP` and `HTTP`.

`abstract sig Node` represents the main components of SDNs: `Host`, `Switch` and `Controller`. 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`). Observe that, in this case, the multiplicity of relations is `some`, meaning that all nodes have at least one port and are at least connected with one node.

```
abstract sig Packet{
  position: lone Port
}
sig DataPacket  extends Packet{
  type: DataPacketType,
  src,dest:Host
}
sig CtrPacket extends Packet{
  newRule: lone Rule,
  request: lone DataPacket
 }
```

Listing 3: Signatures and relations III

`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. Observe that the multiplicity of this relation is `lone`, which means that a packet might 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. Observe that `sig DataPacket` and `sig CtrPacket` have no explicit multiplicities defined. In these cases, the default multiplicity `set` is applied, which means that these two sets may have any number of atoms in every model instance.

```
abstract sig Action{}
one sig Forward extends Action{}
one sig Discard extends Action{}
sig Rule{
   packetType:DataPacketType,
   iPort: Port,
   action: Action,
   oPort : lone Port
}
one sig Controller extends Node{}
sig Host extends Node{
  iBuffer: set DataPacket,
  oBuffer: set DataPacket
}
sig Switch extends Node{
  table: set Rule
}
```

Listing 4: Signatures and Relations IV

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 (relation `packetType`), to a specific input port through which the packet has to arrive (relation `iPort`), and the action to be realized with the packet (relation `action`). When the action is `Forward`, relation `oPort` contains the port through which the packet has to be forwarded.

Finally, signatures `Controller`, `Host` and `Switch` are defined in Listing 4. The specification establishes that there is only `one` controller node. The host nodes have two relations `oBuffer` and `iBuffer`, which contain the data packets to be sent or received, respectively. Switches have a `table` relation that stores a set of rules.

Figure 3 shows two instances of the ALLOY model just constructed. 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.

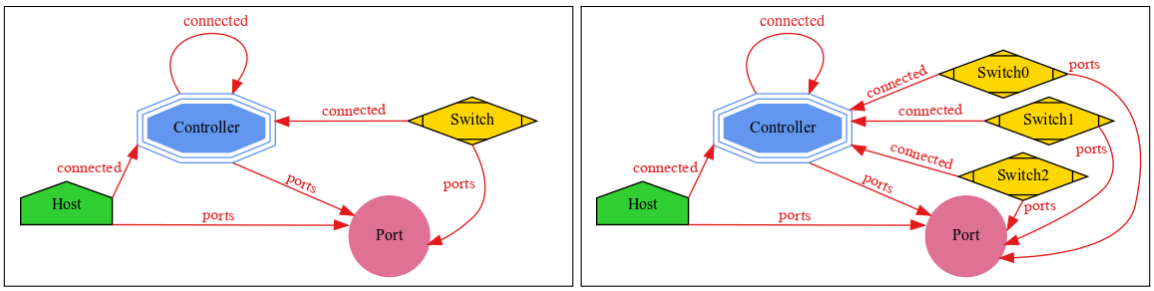


Figure 3: Examples of instances of the ALLOY model

## 3.2 Adding constraints to the ALLOY SDN model

We now add constraints to generate structurally correct SDNs. As commented above, ALLOY uses a first-order relational logic to define constraints. It 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. We will explain its use in ALLOY with the SDN example.

```
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}
}
```

Listing 5: Functions I

```
fun reachableNodes(n:Node):Node{
  n.^connected
}
fun dataLinks(n:Node): Link{
  nodeLinks[n] & DataLink
}
fun remotePort(p:Port): Port{
  (link[p].(p1 + p2))-p
}
```

Listing 6: Functions II

ALLOY supports the definition of functions to ease the definition of constraints. Listings 5 and 6 show some functions used in the SDN example. 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` (`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. Operator `+` represents the union of sets. Function `nodeLinks`, which returns the links of a node `n`, is defined by intension as the set of links `l` having a port `l.(p1+p2)` in common with the ports of `n`. The symbol `&` represents the intersection of sets. Function `reachableNodes` uses the transitive closure of binary relation `connected`, written as `^connected`, to find all nodes reachable from a given node. Function `dataLinks` uses `nodeLinks` to calculate the data links of a given node, and finally, function `remotePort` uses `link` to compute the remote ports connected with a given port by links. As usual, `-` is the difference of sets.

Constraints are added as `facts` to models. All model instances have to satisfy all constraints in facts. Listing 7 shows some examples. 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 LinksAndPorts{
  //1- Link ports  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 is at most in a link
  all p:Port | lone link[p]
  //4-  Links connect different nodes
  all l: Link | node[l.p1]!=node[l.p2]
}
```

Listing 7: Constraints on ports and links

Fact 2 uses the 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 the two ports of a link belong to different nodes; that is, a link connects two different nodes.

Listing 8 shows facts referring to the correct connection of control and data links. Fact 5 defines exactly the elements of relation `connected`. Each `Node` is connected with those nodes with which it shares a link. Facts 6 and 7 assert, respectively, that a `CtrLink` connects the `Controller` with a `Switch`, and that a `DataLink` connects a `Switch` with other `Switch` or with a `Host`. Fact 8 establishes that the `Controller` has only control links. Word `in` is the subset operator. Fact 9 indicates that each `Switch` has to be connected with the `Controller` by means of exactly one link. Finally, 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.

```
fact Links{
//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:CtrLink| 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:DataLink| 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
}
```

Listing 8: Facts on data and control links

```
fact Nodes{
//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
}
```

Listing 9: Facts on nodes

Listing 9 includes facts to avoid the existence of isolated nodes in the model instances. Thus, fact 11 says that each switch has to reach at least two different hosts. Fact 12 says that switches have to be connected directly with at least two different nodes. Facts 13 and 14 impose that hosts are not directly connected with the controller, but they have to be directly connected with exactly `one` switch.

Facts in Listing 10 have been included to 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.

```
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 packet:DataPacket| one packet.position or one (iBuffer+oBuffer).packet
}
```

Listing 10: Facts on packets

```
fact DataControlLinks{
//17- A data link contains only DataPackets
all l:DataLink|
        l.(p1+p2)[position] in DataPacket
//18- A control link contains only CtrPackets
all l:CtrLink|
        l.(p1+p2)[position] in CtrPacket
}
```

Listing 11: Facts on control links

Facts on Listing 11 refer to the content of data links. Thus, 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`. Fact 18 says the same for the control links.

Facts on Listing 12 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.

```
fact Packets{
//19-The source and destination hosts of a DataPacket are different
  all pack:DataPacket| pack.src != pack.dest
//20-Each control packet has a new rule or a request
  all pack:CtrPacket | one pack.(newRule + request)
//21-A control packet with a new rule arrives to a Switch port
 all pack:CtrPacket| (one pack.position & Switch.ports) implies one pack.newRule
//22-A control packet with a request arrives to a Controller port
 all pack:CtrPacket|(one pack.position & Controller.ports) implies one pack.request
//23-iBuffer is well defined
 all h:Host | h.iBuffer in {pack:DataPacket | (no pack.position) and pack.dest = h}
//24-oBuffer is well defined
 all h:Host | h.oBuffer in {pack:DataPacket | (no pack.position) and pack.src = h}
//25-Packets in an oBuffer can reach the destination host
 all h:Host, pack:h.iBuffer | pack.dest in reachableNodes[h]
}
```

Listing 12: Facts on Packets

Facts on Listing 13 refer to `Rules` and routing `tables`. Fact 26 is very simple, it says that each `Rule` can be at most in a table. The next one (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 a universal 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. The rest of facts concern the consistency of rules. For instance, fact 28 asserts that the input port of each rule in a switch `s` has to be a port of `s` and must be linked by means of a `DataLink` with another remote port. Facts 29 and 30 refer to the content of rules. Fact 29 establishes that each rule has an output port iff its action is `Forward` and the output port is linked to another remote node. Fact 30 deals with rules without output ports for which their `action` has to be `Discard`. 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. The last constraint (fact 32) imposes that the output and input ports of each rule have to be ports of the `Switch` in which the rule is installed.

```
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 have output port
all r:Rule | one r.oPort iff r.action = Forward and
            one (r.oPort & dataLinks[Switch].(p1+p2))
//30-Discard rules have 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 iPort and oPort of a rule in a switch belong to the switch
all s:Switch, r:s.table| node[r.(iPort + oPort)] = s
}
```

Listing 13: Facts on rules and routing tables

As it can be observed with the list of constraints given in this section, the construction of an ALLOY model is not a trivial task. It 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.

## 4 Dynamic ALLOY instances

The ALLOY models described in the previous section are *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 static ALLOY models into dynamic ones. 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.

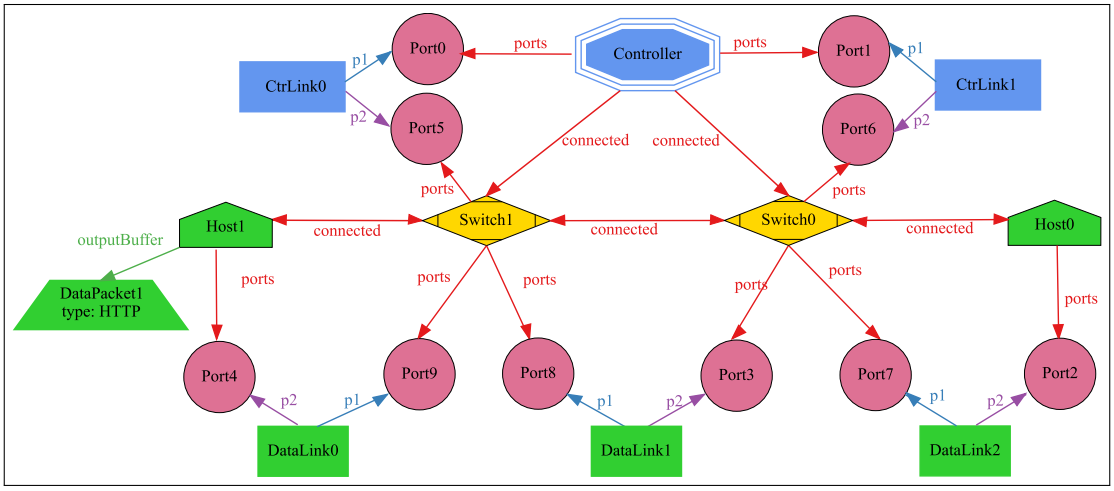


Figure 4: Instance of the ALLOY model

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.

The rest of this section is devoted to describing the previous tasks using the SDN example of Section 3.

## 4.1 Extension of relations with Time

```
open util/ordering[Time]
sig Time{}
abstract sig Packet{
  position: Port lone -> Time
}
sig Host extends Node{
  iBuffer: DataPacket set -> Time,
  oBuffer: DataPacket set -> Time
}
sig Switch extends Node{
  table: Rule set -> Time
}
```

Listing 14: Relations extended with time

Listing 14 introduces the new signature `Time` and transforms the binary relations `position`, `iBuffer`, `oBuffer` and `table` (see Listings 3 and 4) into ternary by adding the 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 of `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`.

### 4.2 Introducing time in facts

Listing 15 contains the facts of the static model (see Section 3) 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.

```
//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- A control link contains only control packets
  all t:Time, l:CtrLink| l.(p1+p2) [position.t] in CtrPacket
//21- A control packet with a new rule arrives to a Switch port
  all t:Time, pk:CtrPacket| (one pk.position.t & Switch.ports) implies one pk.newRule
//22- A control packet 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 iBuffer 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 input port
  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 rules of a switch have to be different ports of the switch
  all t:Time, s:Switch, r:s.table.t| node[r.(iPort + oPort)] = s
}
```

Listing 15: Facts extended with time

### 4.3 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 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.

```
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
  allTablesUnmodifiedExcept[none,t,t']
  allPacketsUnmodifiedExcept[pk,t,t']
  alloBuffersUnmodifiedExcept[none,t,t']
  alliBuffersUnmodifiedExcept[none,t,t']
}
```

Listing 16: Predicate discard packet

Listing 16 shows the conditions for a switch `s` to discard a packet `pk`. The preconditions are that, at instant `t`, the `packet` 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 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 `pack`. The implementation of frame conditions is given below.

Listing 17 contains the implementation of the predicate that forwards a packet `pk` from a switch `s` 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 `pk`, at time instant `t'`, is `p`.

```
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 & r.action= Forward & 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
  allTablesUnmodifiedExcept[none,t,t'] and allPacketsUnmodifiedExcept[pk,t,t']
  alloBuffersUnmodifiedExcept[none,t,t']
  alliBuffersUnmodifiedExcept[none,t,t']
}
```

Listing 17: Predicate Forward

```
pred sendPacket(t,t':Time, h:Host, pk:DataPacket){
  //pre
  some pk & h.oBuffer.t
  //post
  some p':remotePort[h.ports] | pk.position.t'=p'
  h.oBuffer.t' = h.oBuffer.t - pk
  //frame
  allTablesUnmodifiedExcept[none,t,t']
  allPacketsUnmodifiedExcept[pk,t,t']
  alloBuffersUnmodifiedExcept[h,t,t']
  alliBuffersUnmodifiedExcept[none,t,t']
}
```

Listing 18: Predicate Send

Listing 18 contains the predicate that sends packet `pk` from a host `h`. The precondition is that `pk` is at the output buffer of `h` at instant `t`. The predicate has two post-conditions: 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`.

```
pred receivePacket(t,t':Time,h:Host,pk:DataPacket){
  //pre
  some (pk.position.t & h.ports)
  //post
  h.iBuffer.t' = h.iBuffer.t + pk
  pk.position.t' = none
  //frame
  allTablesUnmodifiedExcept[none,t,t']
  allPacketsUnmodifiedExcept[pk,t,t']
  alloBuffersUnmodifiedExcept[none,t,t']
  alliBuffersUnmodifiedExcept[h,t,t']
}
```

Listing 19: Predicate Receive

Similarly, Listing 19 contains the conditions for a host `h` to receive 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 `req` from a switch to know how to manage a data packet.
- `pred sendRequest(t,t':Time, s:Switch, pk:DataPacket)`, switch `s` sends data packet `pk` to the controller to ask how to manage.
- `pred installRule (t,t':Time, s:Switch, pk:CtrPacket)`, switch `s` receives a control packet `pk` with a rule and installs in its routing table.

Frame conditions can be found in Listing 20. Each frame predicate corresponds to a relation that may change over time. Frame predicates establish the set of relations that must remain unchanged from time instant `t` to `t'`.

```
pred allPacketsUnmodifiedExcept(pp:set Packet, t,t':Time){
   all pk:Packet-pp | pk.position.t = pk.position.t'
}
pred alloBuffersUnmodifiedExcept(hh:set Host, t,t':Time){
   all h:Host-hh | h.oBuffer.t = h.oBuffer.t'
}
pred alliBuffersUnmodifiedExcept(hh:set Host, t,t':Time){
   all h:Host-hh | h.iBuffer.t = h.iBuffer.t'
}
pred allTablesUnmodifiedExcept(ss: set Switch, t,t':Time){
    all s:Switch - ss | s.table.t = s.table.t'
}
```

Listing 20: Frame Conditions

## 5 Generation of Instances

ALLOY can run (simulate) predicates and check assertions. In the first case, ALLOY 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 assertions. In both cases, the ALLOY model is transformed into a set of boolean formulae that are analysed using a SAT solver. Currently, ALLOY integrates the following solvers:

- *SAT4J*[1] is a Java library and a standalone solver for solving boolean satisfaction and optimization problems. SAT4J implements incremental SAT solving algorithms, which are used by ALLOY to

[1] Available at: http://www.sat4j.org/

rapidly show different instances of the model. By default, ALLOY uses this solver because of its portability to different platforms, although it may be less efficient than other solvers.

- *miniSat*[2] is a very efficient SAT solver based on a Conflict Directed Clause Learning (CDCL) algorithm. It was designed to be easily adaptable with new SAT techniques and supports incremental SAT and diverse mechanisms for adding non-clausal constraints. The tool is implemented in C++ but there is also a Java implementation.
- *Glucose*[3] is based on miniSat (C++ implementation), and includes a new scoring scheme for the clause learning mechanism, whose objective is to detect in advance the *useful clauses*, and remove the others to reduce the use of memory. The latest version of Glucose is designed to be parallel, but it is not clear if ALLOY makes use of this feature.
- *Lingeling and Plingeling*[4] are SAT solvers implemented in C. These solvers interleave different pre-processing algorithms, such as failed literal probing, lazy hyper binary resolution, decomposition into strongly connected components for equivalence reasoning, etc. To reduce memory usage, Lingeling stores the clauses in different data structures depending on the number of literals. Plingeling is the multi-thread version of Lingeling. Each worker thread runs a new instance of the SAT solver, and they share the generated unit clauses by means of a coordinator thread.

In this section, we use the run and check approaches to generate instances of the SDN model. First, we define and 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. It is worth noting that these predicates can be used to model non-deterministic execution traces of the dynamic model. Second, we define and check some assertions that prove properties of our model. If the properties are not satisfied, ALLOY returns a counterexample. We have defined the following configurations:

- Configuration 1: the model instances include a 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. Listing 21 shows this predicate. The keyword *run* instructs ALLOY to simulate the predicate *Config1* with at most 10 atoms per signature.
- Configuration 2: the model instances include two or more hosts, and at least a data and a control packet. In addition, signatures can have at most 15 atoms.
- Configuration 3: This configuration 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 configuration of the network in the first time instant. In particular, the data packets are in the output buffer of the source host and the switches' tables have pre-installed all the necessary forwarding rules. Thus, in this configuration, switches should not send requests to the controller. In addition, the predicate defines how the instances can non-deterministically evolve over time. Finally, the predicate is simulated with at most 10 atoms and 9 time instants. Listing 22 shows this predicate.
- Configuration 4: This configuration extends configuration 3 to include the communication between the controller and the switches. In this case, in the initial time instant the switches' tables are empty. Thus, when a switch receives a packet, it automatically has to send a request to the controller in order to know how to manage it. The predicate is simulated for 10 atoms.

[2] http://minisat.se/
[3] http://www.labri.fr/perso/lsimon/glucose/
[4] http://fmv.jku.at/lingeling/

```
pred Config1(){
 #Controller = 1 and #Host = 2 and #Switch>=2 and #DataPacket = 2
 //all ports are in a link
 all p:Port | one link[p]
 //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 21: ALLOY predicates for configuration 1

```
pred initTopology3(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 Config3(){
 initTopology3[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 Config3 for 10 but 9 Time
```

Listing 22: ALLOY predicates for configuration 3

ALLOY displays the number of variables and primary variables, clauses, the time to transform the model into clauses, which is independent of the underlying SAT solver, 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 number of variables and clauses for each configuration predicate, the average execution time of the SAT solvers (calculated on 20 executions of each configuration). ALLOY finds instances of the models for all configurations, but only SAT4J and miniSat can generate more than one instance thanks to their incremental algorithms. We also observe that miniSat and Glucose have the fastest algorithms.

Figure 5 shows simplified views of two network topologies given by configurations 1 and 2. To ease the visualization of the results, ALLOY visualizer can project instances on a given signature. This is specially useful to visualize dynamic models by projecting the instance on time. For example, Figure 6 shows an instance satisfying the configuration 3 in which we can observe how the data packets change their position in each time instant.

By inspecting the instances obtained in the predicate simulations, we can visually detect errors in the model requirements. However, the number of instances can make this task infeasible. As an alternative, ALLOY can check assertions and thus, automatically, determine whether all model instances satisfy

| | Config. 1 | Config. 2 | Config. 3 | Config. 4 |
|---|---|---|---|---|
| Vars | 36,368 | 98,423 | 170,110 | 192,475 |
| Primary vars | 1,488 | 3,283 | 4,799 | 5,208 |
| Clauses | 97,823 | 292,743 | 346,270 | 377,230 |
| Execution Time | | | | |
| SAT4J | 306 | 12,052 | 4,958 | 18,926 |
| MiniSat | 86 | 1,715 | 3,395 | 1,271 |
| Lingeling | 2,532 | 7,025 | 4,791 | 9,681 |
| Plingeling | 1,511 | 8,727 | 13,653 | 10,299 |
| Glucose | 67 | 489 | 1,221 | 4,691 |

Table 1: ALLOY report including execution time (ms) of different SAT solvers

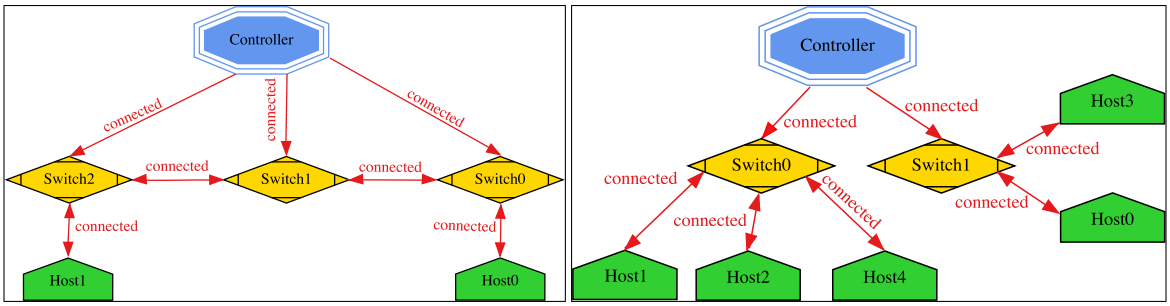


Figure 5: Different network topologies

a property. For example, in configuration 3, we stated that switches should not send requests to the controller because all rules were initially installed in the switches' tables. This property can be checked with the assertion shown in Listing 23, which specifies that for all time instants, there is no control packet in the controller (input) ports. Another desired property of configuration 3 is that data packets should go from the source to the destination hosts. Listing 23 shows the assertion to prove the property. In both cases, ALLOY concludes that the assertion is never violated.

```
//Controller does not receive requests
assert noRequest{ Config3 implies all t:Time | no pk:CtrPacket |
        one (pk.position.t & Controller.ports)}
check noRequest for 10 but 9 Time
//all data packets arrive to the destination host
assert packetArrival{ Config3 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
//Search for instances with more than 5 Hosts
assert Hosts5{ Config2 implies #Host<5}
check Hosts5 for 15
```

Listing 23: Assertions

Checking assertions is also an useful approach to filter model instances with specific features, specially when the SAT solver does not implement incremental algorithms. For instance, to obtain a model instance with 5 hosts (see Figure 5), we can simulate configuration 2 with miniSat (or SAT4J) and inspect the different instances generated. Alternatively, we can also check the assertion shown in Listing 23 that checks whether all model instances have less than 5 hosts, with any of the available solvers.

## 6 Conclusions and future work

In this paper, we have described ALLOY, a declarative language based on sets, relations and first-order logic, and a tool for the formal description and analysis of complex systems. We have presented the main language instructions by means of the specification of a model for software defined-networks. This application constitutes 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 tutorial 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.

## References

[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, doi: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, doi: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, doi:10.1109/SURV.2014.012214.00180.

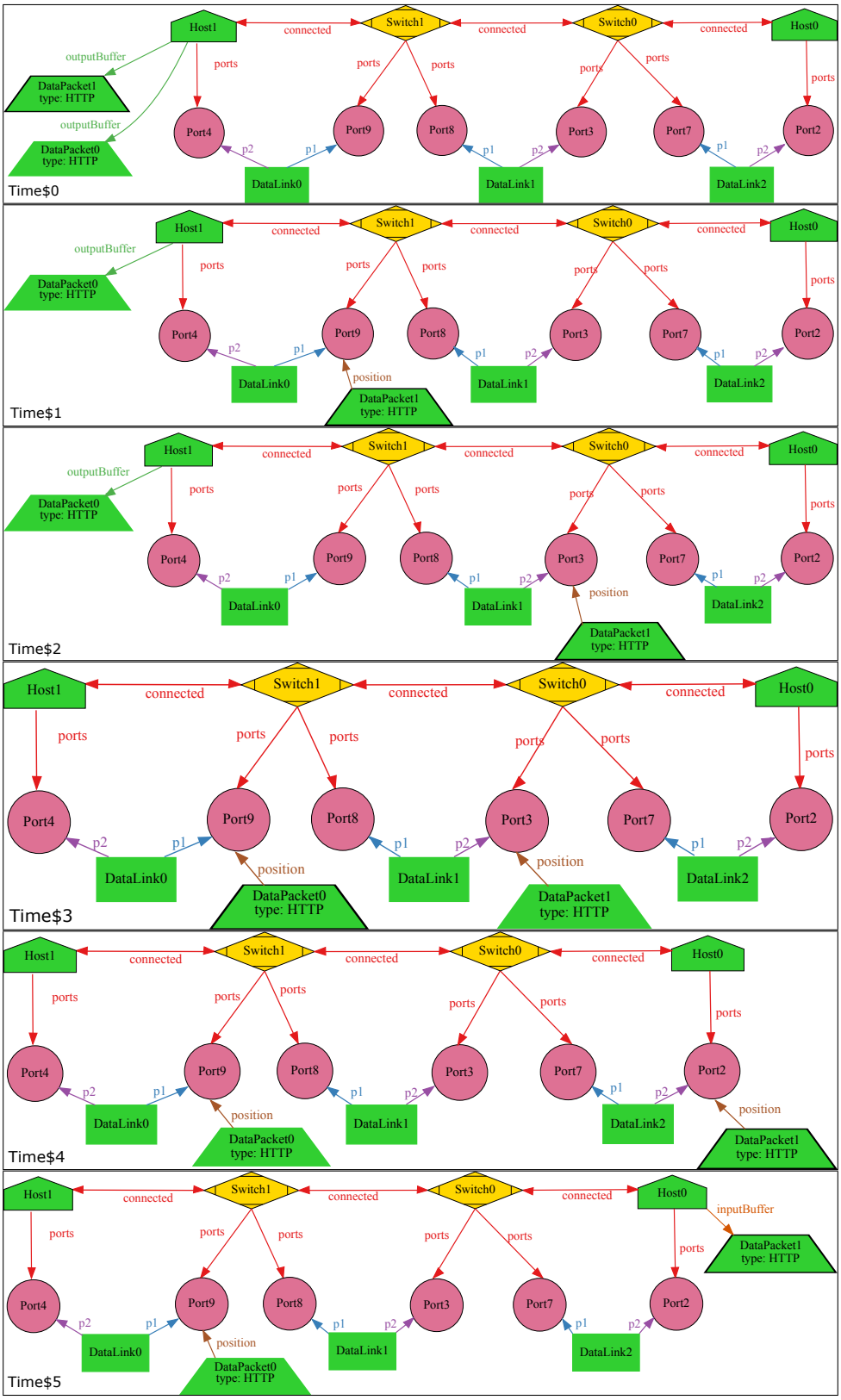


Figure 6: Evolution over time of data packets