Using Alloy to Formally Model and Reason About an OpenFlow Network Switch

Saber Mirzaei smirzaei@bu.edu
Sanaz Bahargam bahargam@bu.edu
Richard Skowyra rskowyra@bu.edu
Assaf Kfoury kfoury@bu.edu
Azer Bestavros best@bu.edu

Computer Science Department
Boston University

***Abstract*—Openflow provides a standard interface for separating a network into a data plane and a programmatic control plane. This enables easy network reconfiguration, but introduces the potential for programming bugs to cause network effects. To study OpenFlow switch behavior, we used Alloy to create a software abstraction describing the internal state of a network and its OpenFlow switches. This work is an attempt to model the static and dynamic behaviour a network built using OpenFlow switches.**

I. Introduction

Software-defined networking (SDN) is a technique used in computer networking to decouple the control plane and the data plane of the network. SDN employs a centralized software program to automatically configure and control network switches. switches. SDN helps researchers to write high level programs to control network behavior instead of manually configuring and manipulating policies in traditional network switches. OpenFlow is the first standard communication interface defined between the control and forwarding layers of an SDN architecture. OpenFlow is an open API to remotely control forwarding tables of switches by adding and removing flow entries [14]. OpenFlow provides an easy interface for changing network configurations, but also enables software bugs to impact network behavior. This raises a number of questions: ”Does changing configurations via OpenFlow cause any security breaches or inconsistencies in switches? Can it cause undesired network behaviors unknown to the network operator?”

In this report, we will use Alloy, a lightweight modeling language, to create a software abstraction describing the static structure and dynamic behavior of the OpenFlow network. Using this model, in future work we aim to answer the aforementioned questions about OpenFlow switch networks. The remainder of the paper is laid our as follows.

In section II we describe Software Defined Networks in more detail, and introduce the fundamental concepts of the Alloy language.

Section III discusses the key components of an OpenFlow switch. We explain the static model of an OpenFlow switch in IV and the dynamic model in V. Related work on SDN design and verification is reviewed in Section VI. Finally, Section VII concludes the paper and discusses our future work.

II. Background

A. Software Defined Networks

In SDN the data forwarding plane is separated from control plane , which is managed by a network OS. The network OS (such as Nox [5], POX [13],Beacon [1], or ONIX [10]) controls the whole network from a central point. It controls the data plane via interfaces such as OpenFlow [14]. Accordingly, the functionality of a network can be defined and changed after SDN has been physically deployed. Hence, changing network switch’s rules, prioritizing, de-prioritizing or even blocking/re-routing packet flows is facilitated a very fine-grained level of control.

A software defined controller allows us to trace and manage specific flows in a flexible approach based on packets’ header information (such as packet’s source/destination address).

OpenFlow is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow facilitates the software defined routing of packets through the network of switches and also provides sophisticated traffic management. The basic idea of OpenFlow is to exploit the concept of flow-tables (already used in Ethernet switches) for different applications such as implementing firewalls, QoS and NAT. Employing this notion, OpenFlow provides a protocol in order to program the flow-tables for routing packets and managing flow traffics. More importantly, using OpenFlow, network administrators can separate production and research traffic. Hence this gives the researcher the ability to implement and test new routing protocols, security models or even alternatives to IP [14] on real-world networks. Figure 1 shows a network of OpenFlow switches.In this example, all the switches are managed by only one controller. In general based on OpenFlow specification, a switch can be controlled by more than one Controller.
Every OpenFlow switch consists of at least three parts [17]: (1) A set of Flow Tables (at least one table). Each table has a set of flow entries. Each entry comprises a set of actions that will be applied to the packet when it matches that entry, (2) A Secure Channel for communication with corresponding controller(s), (3) and OpenFlow as the standard protocol for communication with controller(s) [14].

More details on the OpenFlow switch are presented in section III.

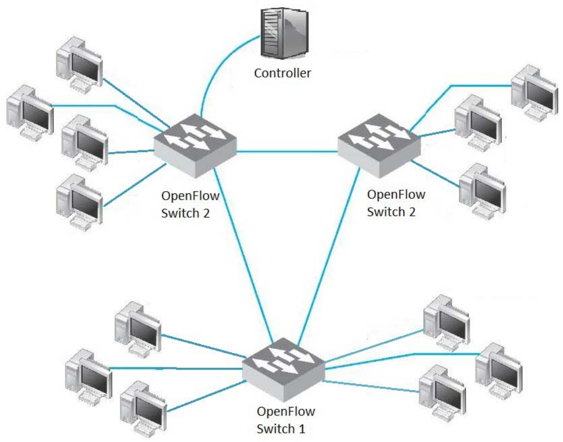

Fig. 1: Example of a network with three switches and one controller.

B. Alloy Modelling Language

Alloy is a declarative specification language for modeling complex structures and behaviors in a system. Alloy is based on first order logic [7] and is designed for model checking. An Alloy model is a relational model which can consist of:

- Signatures: represent the entities in a system.
- Relations: which relates a signature to another signature.
- Facts: specify the constraints on the signatures which are assumed to always hold.
- Predicates: specify the constraints which can be used to show operations.
- Functions: are alternative names for expressions that return some result.
- Assertions: constraints which are checked over the model.

The Alloy Analyzer takes an Alloy model and checks it against its constraints. It translates the Alloy model into a boolean expression which is analyzed by the SAT solver embedded in the analyzer. The Alloy Analyzer generates instances of model invariants satisfying model constraints and then checks specified properties on these instances. It just returns the models within the user-specified scope, consisting of a finite number of objects. The result is turned into a graphical representation of a model instance.

Besides finding satisfying instances, the Alloy Analyzer also checks assertions. If there is a model within the scope which does not satisfy the assertions, it will return the model as a counterexample. However if no instance is found, the assertion may still be not valid in a larger scope.

C. Examples:

The general syntax for defining a signature in Alloy is as follows:
`sig A fields`
This line defines signature `A` with some fields. For instance Figure 2 shows two entities in Alloy which are defined using keyword **sig**. Line 1 defines the `Switch` entity with a field `Table`. Signature `SwState` shows the state of the network at each time epoch. `SwSatet` has two relations, `switch` and a set of buffered messages (`BuffMsg`) mapping ports to Messages.

```
1  some sig Switch{tables: some Table}
2  sig SwState{sw: one Switch, BuffMsg: set (Port->Message)}
```

Fig. 2: Example of a signature in Alloy.

```
pred receive(s,s' : SwitchState, m:Message){
        s'.switch = s.switch
        s'.sInBuffMsg = s.sInBuffMsg + m
        s'.sOutBuffMsg = s.sOutBuffMsg
        s'.sTables = s.sTables
}
```

Fig. 3: Example of a predicate in Alloy.

Operations that modify the state of network may be modeled as *predicates* using the **`pred`** keyword. The syntax is:
`pred Name [parameters] f`
This line defines a predicate, with the given name and (possibly empty) parameters. A predicate always produces true or false, so no type is needed. The result is defined by the formula f, which may reference the parameters. Figure 3 specifies how the state changes when a new message is received. s indicates the current state and s' indicates the next state after receiving a new message. Upon receiving a message the only change is the relations between switch state and the set of buffered messages. The new message is added to the buffer, but the other relations stay unchanged.

Facts express constraints on signatures and the relations that must always hold in the model. The following syntax shows how to define a fact:
`fact Name e`
You can name a fact if you wish. The analyzer will ignore the names. The expression `e` is a constraint that the analyzer will assume is always true. For instance the fact in Figure 4 implies the next table of every switch table should be in the same switch. Using following syntax a function can be defined in Alloy:
`fun` `Name` **`[`**`parameters`**`]`** `: type` **`e`**
This line defines a function, with the given name and (possibly empty) parameters, and outputting a relation (or a set, or scalar) of the given type `type`. The result is defined by the expression e, which may reference the parameters. For instance Figure 5 presents an example of a function in Alloy. This function finds the set of switches' ports (from set of all input ports `ports`) connected to a set of controllers (`c`).

An Alloy assertion can be defined using the following syntax:
`assert Name f`
This line defines a assertion, with the given name. Assertions take no parameters. An assertion always produces true or false,

```
fact {all s:Switch, t: s.tables | t.nxTable in s.tables}
```

Fig. 4: Example of an Alloy fact

```
fun findCPort(c: set Controller, ports: set Port): set Port
 {
    (connect.(c.ports)) & ports
 }
```

Fig. 5: Example of a function in Alloy.

```
assert Acyclic{ no t: Table | t in t.^nxTable }
check Acyclic for 5
```

Fig. 6: Example of an assertion in Alloy.

so no type is needed. The property that is going to be checked is defined by the formula `f`. Figure 6 depicts an example of an assertion in Alloy. `Acyclic` asserts there is no loop in the tables' chain. This assertion is checked in all models with at most 5 elements of each signature.

So far Alloy's basics which we are going to use in our code are explained. In the following sections the Alloy code for modelling the OpenFlow switch is presented.

III. Switch Structure

A. OpenFlow Tables

Any Openflow switch has at least one Flow Table. Each table may have a pointer to another table as the next table. There is no pointer to the first table, and this table is called the root table.

A Flow Table consists of some flow entries. Different components of each flow entries are:

- Match fields: to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
- Counters to update for matching packets.
- Instructions to apply to matching packets.

1) Match Fields: In our model of an OpenFlow switch a very simplified version of the match fields is modeled. For a complete set of match fields that a packet is compared against, see the current OpenFlow switch specification [17]. Currently a packet is compared against the ingress port, source IP and destination IP. Each match field in a flow table has a priority field. If more than one match fields match with an incoming packed, the one with higher priority will be triggered.

2) Counters: Counters are stored statistics that can be maintained for each flow, port, table, etc. In current work, a very simplified version of counters in the static model of OpenFlow switches is presented.

3) Overview: A set of instructions is associated with every flow entry. This set of instructions is executed whenever an incoming packet matches the corresponding entry. In this version of the OpenFlow switch model the following instructions are modeled:

- **Apply-Action action(s):** A specific set of actions is immediately applied while the current action set (associated with the packet) remains unchanged.
- **Clear-Action:** clear the associated action set.
- **Write-Action action(s):** Add a specific set of actions to the current actions set.
- **Goto-Table next-table-id:** continue the pipe-lining process (described in next section) from table with ID equal to next-table-id. The next-table-id must be greater than the current table-id.

4) Action Set: Each incoming packet has an action set which is initially empty. Using Write-Action and Clear-Action, the action set can be modified whenever the packet matches an entry. An action set contains at most one action of each type. In order to have multiple actions of the same type, the Apply-Action instruction can be used. There are currently two types of action in the OpenFlow specification: required and optional [17]. No optional action is modeled in the current work. Supported required actions in this model are:

output:

- **Forward-to-Port port:** forward the packet to port.
- **Forward-to-Controller controller:** forward the packet to a specific controller.
- **Forward-to-Ingress:** forward the packet to the ingress port (the port that packet has been received from).
- **Forward-to-All:** forward the packet to all outgoing ports.

Drop: Drop the packet. This action also can be applied implicitly for those packet whose action sets have no output action.

B. Matching a packet with flow entries

If a packet matches a flow entry in a flow table, the corresponding instruction set is executed. The instructions in the flow entry may explicitly direct the packet to another flow table, where the same process is repeated again. A flow entry can only direct a packet to a flow table number which is greater than its own flow table number, in other words pipeline processing can only go forward and not backward. Obviously, the flow entries of the last table of the pipe-line can not include the Goto instruction. If the matching flow entry does not direct packets to another flow table, pipe-line processing stops at this table and the corresponding action set will be executed. Packet flow through an OpenFlow switch is presented in Figure 7.

IV. Static Model of OpenFlow Switch

Our Alloy model enables us to model states of one switch and its interaction with other nodes in the network. We first describe the entities in our model and then we introduce the constraints on the entities. These constraints (modeled using Alloy facts) help us to model the correct structure of an OpenFlow network.

A. Entities

An OpenFlow network consists of a set of nodes(controller and switches). Every node has a set of ports which connect nodes together. In addition to ports, switches also have a set

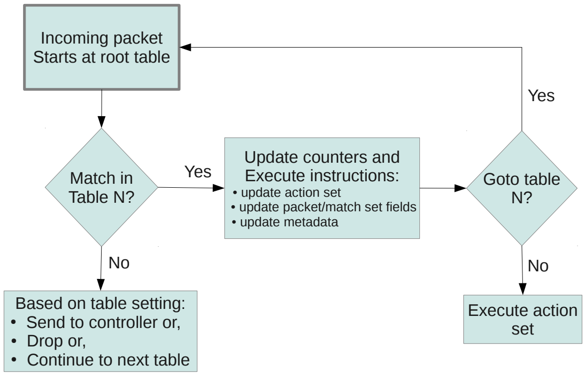

Fig. 7: Packet flow through an OpenFlow switch.

```
1  abstract sig Node{contained:one Network, ports:some Port}
2  some sig Switch extends Node{tables: some Table, root: one
3   RootTable}
4  some sig Controller extends Node{control:some Switch}
5  sig Port{connect:lone Port}
6  one sig Network{content:some Node}
7  sig Packet{srcIP: one IP, destIP: one IP, preTable,
8   postTable :  lone Table }
9  some sig Message{msgInPort: one Port, packet: one Packet,
10  type: one MsgType}
11 abstract sig MsgType{}
12 one sig ControllMsgType extends MsgType{}
13 one sig HostMsgType extends MsgType{}
```

Fig. 8: A network model consists of nodes, packets and messages. Lines 12 and 13 model two types of incoming messages.

of flow tables. The root table is the first table which the pipe-lining process for an incoming packet starts with. Every packet has a source and a destination IP. If packet is coming from the controller, it also includes a `preTable` and a `postTable` which reflect the modification need to be done in switch's flow tables. Using this modification (`preTable` will be replaced by `postTable`). In our model each packet is encapsulated in a message which specifies the type of message and the port the packet was received on. The Alloy model of different elements of an OpenFlow network is presented in Figure 8.

The most important part of the switch model is the flow tables. The tables are uniquely determined by their tableID and all the tables (except the last table) have a pointer to the next table. Every flow table includes a set of flow entries, which messages are checked against. If none of the entries match a message, a table miss occurs. Table miss can be continuing pipe-line process with the next table, forwarding message to some controllers or dropping the packet. Tables and related elements are modeled depicted in Figure 9.

As described in section III each flow entry contains match fields, counter and instructions. Our simplified modeling of different types of actions and instructions and their relation with each other and with respect to the flow entries is presented in Figure 10. For more detail on instructions and actions and their relation you can refer to section III or OpenFlow specification [17].

```
sig Table {nxTable:lone Table, entries: set FlowEntry,
 tableID: one Int,  miss: lone TableMiss}
sig RootTable extends Table {}
abstract sig TableMiss{}
one sig MissDrop extends TableMiss{}
one sig MissNext extends TableMiss{}
```

Fig. 9: Every flow table contains some flow entries and has a corresponding action for packet miss occurrence.

```
 sig FlowEntry {match: one MatchField, counters: some
  Counter, instructs: one Instruction, flowID: one Int}
 abstract sig Instruction{}
 sig ApplyActionsIntruct extends Instruction{appActions:
  set Action}
 sig WriteActionsIntruct extends Instruction{wrtActions:
  set Action}
 sig ClearActionIntruct extends Instruction{}
 sig GotoTableIntruct extends Instruction{gotoTable: one
  Table}
 abstract sig Action {}
 sig ForwardtoPort extends Action{toPort: one Port}
 one sig ForwardtoAll extends Action{}
 one sig ForwardtoIngress extends Action{}
 sig ForwardtoController extends Action{toController: one
  Controller}
one sig Drop extends Action{}
```

Fig. 10: Definition of flow entries, different instructions and actions.

B. Constraints on Network Entities

The model of OpenFlow switch imposes some constraints on the network entities. These constraints help us to get a correct model of structure of an OpenFlow network. In this section we will try to capture those important rules. The fact in Figure 11 ensures that all of the switches are controlled by at least one controller and the controller and switch are connected. Based on the OpenFlow specification, each switch has at least one flow table. There is exactly one table as the root table which the pipe-lining process starts from. The set of tables of a switch create a loop free chain. Each table has some (at least one) flow entry. Also each table has a table miss action. This table miss action can be explicit or implicit. Implicit means there is no specific action assigned for a table, hence the default action must be applied. Different table miss actions and the behaviour of a switch for each one is presented in section V. In order to make sure that our model conforms to all these rules, a group of Alloy rules are needed. The combination of these rules presented in Figure 12 ensure a model representative of the specification. For instance, lines 11-18 take care of following rules: 1) there is only one connected acyclic chain of tables in each switch, 2) all of the tables in the chain belong to the same switch and 3) there is only one last table in this chain which its pointer to next table is empty. All constraints on flow entries are presented in Figure 13. Each flow entry is contained in only one table and every counter is contained in at least one flow entry.

```
fact{all s: Switch |some p1: s.ports, c:Controller, p2:
 c.ports | s in c.control && p2 in p1.connect}
```

Fig. 11: Every OpenFlow switch is controlled by at least one controller and they are connected to each other.

```
fact{all t: Table | one s: Switch |  t in s.tables}
fact{all t: RootTable | one s: Switch |  t in s.root}
fact {all m:TableMiss | some t:Table | m in t.miss}
fact {all t:Table | #(t.entries) > 0}
fact {all s : Switch | one r: RootTable | r in s.tables}
fact{all s:Switch | s.root in s.tables}
fact {all t:RootTable | t.tableID = 0}
fact {all t:Table | t.tableID >= 0}
fact nextTableID{all t,t’:Table | (t’ = t.nxTable) implies
 (t’.tableID = t.tableID.plus[1])}
fact {all s: Switch, t:s.tables | !(t in  RootTable)
 implies (one t’:Table | t’.nxTable = t)}
fact {all s:Switch, t: s.tables | t.nxTable in s.tables}
fact{no t: Table | t.nxTable = t}
fact {all s:Switch | one t: s.tables | no t.nxTable}
fact acyclicTable { no t: Table | t in t.^nxTable }
fact {all s:Switch, t, t’: s.tables | !(t = t’) implies
 !(t.nxTable = t’.nxTable)}
fact {all t:Table | #(t.nxTable) = 0 implies !(MissNext
 in t.miss)}
 }
```

Fig. 12

```
1  fact {all e: FlowEntry | one t: Table | e in t.entries}
2  fact {all c: Counter | some e: FlowEntry | c in
3   e.counters}
4  fact {all a: Action | (a in ApplyActionsIntruct.appActions)
5   or (a in WriteActionsIntruct.wrtActions)}
```

Fig. 13: All related facts on flow entries of a table.

Line 4 also implies that every action is used in at least one instruction. Obviously in any network model each node (switch or controller) has some ports. Every port belongs to exactly one node and it cannot be connected to another port in the same node. Also each port can be connected to at most one other port and all connections are two ways. All these structural rules are applied using facts in Figure 12. In our model we are not interested in relations between controllers, hence line 8 imposes that controller’s ports should be connected only to switch’s ports. The first fact in Figure 15 implies every instruction should belong to at least one flow entry. In addition in the `GotoTableIntruct` instruction, the pointed table must be in the corresponding switch and it must be a table with larger ID. In Figure 16 the set of facts on different type of actions is given. The second fact in line 4 imposes that the `toPort` of an `ForwardtoPort` action must be in the same switch. Finally lines 8 and 10 make sure there is no repeated actions in our model.

```
1  fact {all p: Port | one n:Node | p in n.ports}
2  fact {no n:Node, p: n.ports | p.connect in n.ports}
3  fact {all p: Port | #(p.connect) <=1}
4  fact {all p, p’: Port | (p = p’.connect) <=>
5   (p’ = p.connect)}
6  fact{all s: Switch |some p1: s.ports, c:Controller, p2:
7   c.ports | s in c.control && p2 in p1.connect}
8   fact {all p: Port | p in Controller.ports implies
9    p.connect in Switch.ports}
```

Fig. 14: Constraints on the ports and connections between them.

```
fact {all i: Instruction | some f: FlowEntry | i in
 f.instructs}
fact {all s:Switch, t:s.tables, e: t.entries, i:
 e.instructs | (i in GotoTableIntruct) implies
 ((i.gotoTable in s.tables) and (i.gotoTable.tableID >
 t.tableID) )}
```

Fig. 15: Facts on flow entry’s instructions.

```
1   fact {all a: Action | (a in
2    ApplyActionsIntruct.appActions) or (a in
3    WriteActionsIntruct.wrtActions)}
4   fact {all a: ForwardtoPort, s:Switch | ( (a in
5    s.tables.entries.instructs.appActions) or  (a
6    in s.tables.entries.instructs.wrtActions) ) implies
7    (a.toPort in s.ports) }
8   fact {all f, f’:ForwardtoPort | !(f = f’) implies
9    !(f.toPort = f’.toPort) }
10  fact {all f, f’:ForwardtoController | !(f = f’) implies
11   !(f.toController = f’.toController) }
```

Fig. 16: Facts on actions in a flow table.

```
sig SwitchState{
 switch: one Switch, sTables: some Table,
 sInBuffMsg: set  Message,
 sOutBuffMsg: set(Port-> Message),
 sTEntries: set FlowEntry, pInHistory: set Message,
 pOutHistory: set (Port->Message),
 nxtInPLTable: lone Table,
 inPL: lone Message, actionSet: set Action,
 outPL: lone Message
}
```

Fig. 17: Definition of a switch’s state. Important elements of a `SwitchState` are `sInBuffMsg`, `sOutBuffMsg`, `pInHistory` and `pOutHistory`. Respectively these elements present the set of buffered for processing, buffered for forwarding, saved in input history and saved in output history messages. Beside, `nxtInPLTable`, `inPL`, `actionSet` and `outPL` are used to handle the behavior of switch during the switch’s table pipe-lining process. `nxtInPLTable` keeps the next table that must be used for flow entry matching. `inPL` and `outPL` denote the message that is checked against flow entries in pipe-lining process. `actionSet` keeps a record of actions that are being added whenever a match is found for the message. Also notice that since the by arrival of a *control Messages*, the flow entries of the tables may change. Hence the set of tables and entries of a switch is also kept in the switch’s state (modeled by `sTables` and `sTEntries` respectively).

V. Dynamic Behavior of the Model

In the previous section the static model of an OpenFlow switch network was presented. Assuming that this model presents the (simplified) real structure of a OpenFlow switch network correctly, the dynamic behavior of OpenFlow switch can be modeled now. Without loss of generality, the internal behavior of only one switch in relation with other elements of network is considered. The internal state of a specific switch changes by various events such as receiving, sending or processing of a message and *etc*. In Figure 17 you can see how the state of an OpenFlow switch is captured. As you see every element that may change in a switch is considered as a part of the internal state of a switch. In order to have a correct chain of switch states, the first `SwitchState` must be initialized. For instance the set of input/output buffered messages in this state must be empty. This rule is applied to the first `SwitchState` using Alloy fact presented in Figure 18. The core part of modeling the dynamic behavior of the switch is to capture the possible transitions between different states.

```
fact {
 no first.sInBuffMsg && no first.pInHistory &&
 no first.pOutHistory && no first.sOutBuffMsg  &&
 first.switch.tables = first.sTables  &&
 no first.sTEntries && no first.nxtInPLTable &&
 no first.inPL && no first.actionSet &&
 no first.outPL
}
```

Fig. 18: Using this fact the first state is initialized. `first` is a reserved keyword in `order` library referring to the first `SwitchState`.

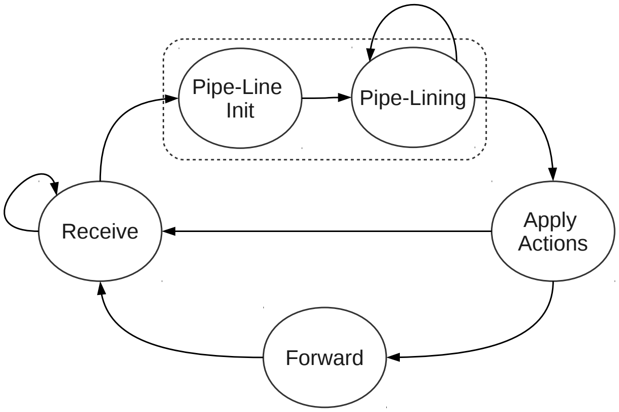

Fig. 19: Five possible state transitions that are considered in our modeling.

Possible transitions that we have considered in our model are schematically presented in Figure 19. In our model the pipe-lining procedure is considered as an atomic transition. Hence until the whole pipe-lining is done, no new message will be received or forwarded. In order to make modeling easier the pipe-lining is broken down into two parts, *Pipe-Line initiation* and *Pipe-Lining*. All these five state transitions are implemented and described by five major Alloy predicates as we explain in following subsections.

A. Receiving a Message

The first state change happens by receiving a new message. Hence the only change in switch state is adding a new message into the input buffer and input history of the switch. All other elements of `SwitchState` are remained unchanged by receiving a new message. These rules are applied by two conditions: `ss'.sInBuffMsg = ss.sInBuffMsg + m` and `ss'.pInHistory = ss.pInHistory + m`. This transition is modeled using the receive predicate presented in Figure 20.

```
1  pred recieve(ss,ss' : SwitchState, m:Message){
2   #(Port->m & (ss'.pOutHistory)) = 0
3   #(Port->m & (ss.pOutHistory)) = 0
4   #(Port->m & (ss'.sOutBuffMsg)) = 0
5   #(Port->m & (ss.sOutBuffMsg)) = 0
6   !(m in ss.sInBuffMsg)
7   ss'.sInBuffMsg = ss.sInBuffMsg + m
8   !(m in (ss.pInHistory))
9   ss'.pInHistory = ss.pInHistory + m
10   ss'.sInBuffMsg in ss'.pInHistory
11   ss.sInBuffMsg in ss'.pInHistory
12   ss'.pOutHistory = ss.pOutHistory
13   ss'.sOutBuffMsg = ss.sOutBuffMsg
14   ss'.sTables = ss.sTables
15   ss'.sTEntries = ss.sTEntries
16   no(ss'.actionSet)
17   no(ss.actionSet)
18   no(ss'.outPL)
19   no(ss.outPL)
20   no(ss'.nxtInPLTable)
21   no(ss.nxtInPLTable)
22   no(ss'.inPL)
23   no(ss.inPL)
24   ss'.switch = ss.switch
25 }
```

Fig. 20: Predicate receive: `ss` represent the state of switch before receiving the message `m` and `ss'` is the state of switch after receiving message `m`. The first four lines apply the condition that the new message `m` is not a repeated message. In line 7 and 9 it's stated that the only difference between `ss` and `ss'` is addition of message `m` to input buffer and input history. Other elements of the switch's state are conditioned to remain unchanged in this predicate.

B. Pipe-Line Initiation

As mentioned before a received message is buffered in the switch and recorded in the input history. In general, a message is either a control message or a host message. Control messages are used to change the flow entries or flow tables in the switch. If the message is a host message, however, pipelining can be initiated. Predicate `pipeLineInit` as presented in Figure 21 shows this state transition. In this predicate condition `ss'.sInBuffMsg = ss.sInBuffMsg - m` makes sure that message `m` is removed from the input buffer. Lines 17 to 27 handle the message based on its type. Lines 17 to 22 model the behavior of switch for a control message. In these lines the set of tables of switch (modeled by `sTables`) changes by replacing the `preTable` with `postTable` of arrived message from controller. The way that flow entries of tables are manipulated in our model is slightly different from the OpenFlow specification [17]. In our model we have simplified this process. Instead of removing or adding a single entry in a specific table, the table is replaced with the updated table. In lines 24 to 27 if the message is of type host message it is set to start pipe-lining in the next state.

C. Pipe-Lining Predicate

The core of the modeling of the dynamic behavior of a OpenFlow switch is the pipe-lining process. In this procedure the message is pulled out of the input buffer and, starting from the root table, is compared with flow entries. This predicate has two important parts. In every round of pipe-lining either there is exactly one flow entry in the current table in pipeline that matches the message, or no entry matches it. In the former case using predicate `applyInstruction` (presented in Figure 24) the set of instructions for the matched flow entry is applied to the message. On the other hand, in the latter case, we are facing a table miss. Hence the appropriate action (based on the table's miss action) must be carried out. the table's miss action can be dropping the message, carrying pipe-lining up using subsequent table or the default action, which is sending the message to connected controller(s).

```
1  pred pipeLineInit(ss,ss' : SwitchState, m:Message){
2   ss'.switch = ss.switch
3   ss'.sInBuffMsg = ss.sInBuffMsg - m
4   ss'.sOutBuffMsg = ss.sOutBuffMsg
5   ss'.pInHistory = ss.pInHistory
6   (m in ss.sInBuffMsg)
7   (m in ss.pInHistory)
8   #(Port->m & (ss.pOutHistory)) = 0
9   #(Port->m & (ss.sOutBuffMsg)) = 0
10  ss'.pOutHistory = ss.pOutHistory
11  no(ss.nxtInPLTable)
12  no(ss.inPL)
13  no(ss.outPL)
14  no(ss.actionSet)
15  no(ss'.actionSet)
16
17  m.type = ControllMsgType implies
18   ((ss'.sTables = ss.sTables - m.packet.preTable
19   + m.packet.postTable) and (ss'.sTEntries = ss.sTEntries
20   - m.packet.preTable.entries + m.packet.postTable.entries)
21   and (no ss'.nxtInPLTable) and (no ss'.inPL) and
22   (no ss'.outPL))
23
24  m.type = HostMsgType implies
25   (ss'.sTables = ss.sTables and ss'.inPL = m and
26    ss'.outPL = m and ss'.nxtInPLTable = ss.switch.root and
27    ss'.sTEntries = ss.sTEntries)
28 }
```

Fig. 21: Predicate `pipeLineInit`: Same as predicate `recieve`, `ss` represent the state of switch before happening of pipe-line initiation and `ss'` is the state of switch after initializing message `m` in pipe-line. In lines 24 to 27 if the message is of type host message, using condition `ss'.inPL = m and ss'.outPL = m` message it is set to start pipe-lining. Also the condition `ss'.nxtInPLTable = ss.switch.root` implies that the pipe-lining must starts with the root table of the switch.

The pipe-lining procedure is modeled using predicate `pipeLining` presented in Figure 22.

D. Message Forwarding Predicate

After being pipelined, a message is usually forwarded via a switch port. Hence a message may be added to some ports' output buffer. `forward` predicate (Figure 25) models the process of removing one message from output buffer, and forwarding it via the corresponding port.

E. Transition Between States

In our model it is assumed that transitions are atomic actions. Namely from any current switch's state `ss` to the exact next state `ss'`, only one of aforementioned five transitions can happen. In order to apply this rule, we add some conditions in a fact shown in Figure 26. In this fact it is applied that between any two consecutive state `ss` and `ss'`, exactly one of the discussed predicates can be correct. In addition each transition happens only for one message.

VI. Related Works

In Veriflow [9] the problem of checking invariants in software defined networks' data plane in real time is addressed. The authors proposed to divide the network into equivalent classes so that checking invariants and violations become easier and more efficient. Conflicting rules can be detected in real time but rules that rewrite packets cannot be checked by Veriflow. Anteater [12] statically analyzes the

```
1  pred pipeLining(ss,ss' : SwitchState){
2   ss'.switch = ss.switch
3   ss'.sTables = ss.sTables
4   ss'.sInBuffMsg = ss.sInBuffMsg
5   ss'.sTEntries = ss.sTEntries
6   ss'.pInHistory = ss.pInHistory
7   ss'.pOutHistory = ss.pOutHistory
8   #(ss.inPL) = 1
9
10  (
11   one f:ss.nxtInPLTable.entries |
12    match[ss, ss.inPL, f, ss.nxtInPLTable] and
13    applyInstruction[ss, ss', f] and
14    ss'.sOutBuffMsg = ss.sOutBuffMsg
15    and ss'.outPL = ss.outPL)
16                   or
17   all f:ss.nxtInPLTable.entries|
18    !match[ss, ss.inPL, f,  ss.nxtInPLTable] and
19    (
20     ((ss.nxtInPLTable.miss in MissDrop) and
21      (no ss'.nxtInPLTable) and (no ss'.inPL) and
22      (no ss'.actionSet) and
23      (ss'.sOutBuffMsg = ss.sOutBuffMsg) and
24      (no ss'.outPL))
25                        or
26     ((ss.nxtInPLTable.miss in MissNext) and
27      (ss'.nxtInPLTable = ss.nxtInPLTable.nxTable) and
28      (ss'.inPL = ss.inPL) && (ss'.actionSet =ss.actionSet)
29      (ss'.sOutBuffMsg = ss.sOutBuffMsg) and
30      (ss'.outPL = ss.outPL))
31                        or
32     ((no ss.nxtInPLTable.miss) and (no ss'.nxtInPLTable)
33      and (no ss'.inPL) and  (no ss'.actionSet) and
34      (ss'.sOutBuffMsg = ss.sOutBuffMsg +
35      findCntrlPort[Controller, ss.switch.ports]->ss.inPL)
36      and (no ss'.outPL))
37    )
38  )
39 }
```

Fig. 22: Predicate `pipeLineInit`: Same as predicate `recieve`, `ss` represent the state of switch before carrying out pipe-lining on one table and `ss'` is the state of switch after pipe-lining. In each round of pipe-lining a message is compared againt the flow entries of table `nxtInPLTable`. If a match is found, using predicate `applyInstruction` (lines 11 to 15) corresponding instructions are applied to the packet. If no match is found, then based on table's miss action one of conditions of lines 17 to 37 must be true.

```
pred match(m:Message,f:FlowEntry,t:Table){
 simpleMatch[m,f,t] and
 (no f': FlowEntry |
   !(f = f') and simpleMatch[m, f', t] and
   f'.match.priority < f.match.priority)
}

pred simpleMatch(m:Message, f: FlowEntry, t:Table){
 f in t.entries
 f.match.matchPort = m.msgInPort or no f.match.matchPort
 f.match.srcIP = m.packet.srcIP or no f.match.srcIP
 f.match.destIP = m.packet.destIP or no f.match.destIP
 (#(f.match.matchPort)!=0 or
         #(f.match.srcIP)!=0 or #(f.match.destIP)!=0))
}
```

Fig. 23: Predicate `match`: This predicate checks if message `m` matches the flow entry `f` in table `t` and more importantly there is no other flow entry in `t` with lower priority that matches `m` (based on matching rule in OpenFlow specification [17]).

```
pred applyInstruction(ss,ss’ : SwitchState, f: FlowEntry){
 (#(f.instructs & ClearActionIntruct) > 0) implies
  (ss’.actionSet =
    (f.instructs & WriteActionsIntruct).wrtActions)
 else
  (ss’.actionSet = ss.actionSet +
    (f.instructs & WriteActionsIntruct).wrtActions)

 (#(f.instructs & GotoTableIntruct) = 1) implies
  (ss’.nxtInPLTable =
    (f.instructs & GotoTableIntruct).gotoTable and
    ss’.inPL = ss.inPL)
 else
  ((no ss’.inPL) and (no ss’.nxtInPLTable)        )
}
```

Fig. 24: Predicate `applyInstruction`: This predicate applies the set of instructions of flow entry `f` to the message that is currently in pipe-line process.

```
1  pred forward(ss,ss’ : SwitchState, m:Message, p:Port){
2   ss’.switch = ss.switch
3   ss’.sTables = ss.sTables
4   ss’.sTEntries = ss.sTEntries
5   (p in ss.switch.ports)
6   (p in ss’.switch.ports)
7   (p->m in ss.sOutBuffMsg)
8   !(m in ss.sInBuffMsg)
9   (m in (ss.pInHistory))
10  !(p->m in (ss.pOutHistory))
11  ss’.sOutBuffMsg = ss.sOutBuffMsg - p->m
12  ss’.pOutHistory = ss.pOutHistory + p->m
13  ss’.pInHistory = ss.pInHistory
14  ss’.sInBuffMsg = ss.sInBuffMsg
15  no ss’.nxtInPLTable
16  no ss.nxtInPLTable
17  no ss’.inPL
18  no ss.inPL
19  no ss’.actionSet
20  no ss.actionSet
21  no ss’.outPL
22  no ss.outPL
23 }
```

Fig. 25: Predicate `forward`: `ss` represent the state of switch before forwarding message `m` and `ss’` is the state of switch after forwarding it via port `p`. As you see the only changes between `ss` and `ss’` are modeled conditions in lines 11 and 12. Respectively in these lines message is removed from output buffer and added to output history.

dataplane configuration to check isolation errors and lack of connectivity due to misconfigurations. Anteater translates high level network invariants into SAT instances and use a SAT solver to check them and returns the counterexamples. Anteater cannot scale well to dynamic network changes and it takes too long to check invariants. In [8] network reachability is examined. To achieve this, a minimal set of packets which are required to cover all rules or links in the network is computed. Then in [18] these packets are sent periodically to all the nodes to check for network failures and errors. [6] introduces ndb, a prototype network debugger inspired by gdb, which implements breakpoints and packet backtraces for SDN which enables debuggers to track down the cause of an error. In [15] introduces consistent network updates in which behaviors are guaranteed to be preserved when a packet it traversing the network. It will enable us to check consistency after transitioning between configurations, but it requires us to store a huge number of extra rules in switches.

```
fact switchStateTransition{
 all ss: SwitchState, ss’ : ss.next {
  (
   (one m:Message | recieve[ss, ss’, m] ) and
   (no m:Message, p:Port | forward[ss, ss’, m, p] ) and
   (no m:Message | pipeLineInit[ss, ss’, m] ) and
   !pipeLining[ss,ss’] and
   !applyActionSet[ss,ss’]
  )
       or
  (
   (no m:Message | recieve[ss, ss’, m] ) and
   (one m:Message, p:Port | forward[ss, ss’, m, p] ) and
   (no m:Message | pipeLineInit[ss, ss’, m] ) and
   !pipeLining[ss,ss’] and
   !applyActionSet[ss,ss’]
  )
       or
  (
   (no m:Message | recieve[ss, ss’, m] ) and
   (no m:Message, p:Port | forward[ss, ss’, m, p] ) and
   (one m:Message | pipeLineInit[ss, ss’, m] ) and
   !pipeLining[ss,ss’] and
   !applyActionSet[ss,ss’]
  )
       or
  (
   (no m:Message | recieve[ss, ss’, m] ) and
   (no m:Message, p:Port | forward[ss, ss’, m, p] ) and
   (no m:Message | pipeLineInit[ss, ss’, m] ) and
   pipeLining[ss,ss’] and
   !applyActionSet[ss,ss’]
  )
       or
  (
   (no m:Message | recieve[ss, ss’, m] ) and
   (no m:Message, p:Port | forward[ss, ss’, m, p] ) and
   (no m:Message | pipeLineInit[ss, ss’, m] ) and
   !pipeLining[ss,ss’] and
    applyActionSet[ss,ss’]
  )
 }
}
```

Fig. 26: Predicate `switchStateTransition`: between any two switch’s state `ss` and `ss’` only one transition happens. Therefore only one predicate is logically true.

[11] introduces SOFT, an approach to test interoperability of Openflow switches and inconsistency between different Openflow agents and the cause of inconsistency.

FlowChecker [3] tries to find intra-switch misconfigurations. FlowChecker translates FlowTable configurations into boolean expressions using Binary Decision diagrams(BDD) and then checks network invariants by using model checkers.

In [2] the whole network is modelled as a finite state machine in which packet header and the location determines states. The goal of paper is to check correctness of network reachability per packet.To achieve this, authors have used BDDs and model checking on properties specified in computation tree logic(CTL) to test all future and past states of packet in the network .

[16] studies inconsistencies caused by updating switch configurations. Authors tries to keep Openflow network consistent at the cost of increasing state in switches to store duplicate table entries.

In addition to switch behavior, a considerable amount of literature has been published with a focus on Openflow controllers. For example, NICE [4] is a controller verification tool which uses model checking and symbolic execution to automate testing Openflow applications. Nice attempts to explore the state space of the whole network and find the invalid system states.

VII. Conclusions and Future Works

The final goal of this research is to use the Alloy Analyzer to generate a model for lightweight verification of OpenFlow switches in order to help researchers in analyzing OpenFlow switch networks' properties. So far the modeling of static structure and also modeling as well as the internal states and dynamic behavior of OpenFlow switch have been considered, specifically matching rules and actions in table entries as a part of a real size network. Consequently, in future works, there are some desired properties that we aim to investigate. Some important properties that will be investigated are:

- Conflicting rules in aggregate flow table
- Existence of loops in the set of forwarding actions
- Scheduling problem in the existence of different controllers

In the follow up research, using Alloy Analyzer, we will try to check these properties of OpenFlow switch networks in our model.

References

[1] Beacon: a java-based openflow control platform.

[2] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. Network configuration in a box: towards end-to-end verification of network reachability and security. In Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on, pages 123–132, 2009.

[3] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: configuration analysis and verification of federated openflow infrastructures. In Proceedings of the 3rd ACM workshop on Assurable and usable security configuration, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM.

[4] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A nice way to test openflow applications. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, NSDI'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[5] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. SIGCOMM Comput. Commun. Rev., 38(3):105–110, July 2008.

[6] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In Proceedings of the first workshop on Hot topics in software defined networks, HotSDN '12, pages 55–60, New York, NY, USA, 2012. ACM.

[7] Daniel Jackson. Software Abstractions: Logic, Language, and Analysis. The MIT Press, 2006.

[8] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[9] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. In Proceedings of the first workshop on Hot topics in software defined networks, HotSDN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[10] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[11] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. A soft way for openflow switch interoperability testing. In Proceedings of the 8th international conference on Emerging networking experiments and technologies, CoNEXT '12, pages 265–276, New York, NY, USA, 2012. ACM.

[12] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In Proceedings of the ACM SIGCOMM 2011 conference, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[13] J Mccauley. Pox: A python-based openflow controller.

[14] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69–74, 2008.

[15] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. SIGCOMM Comput. Commun. Rev., 42(4):323–334, August 2012.

[16] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: change you can believe in! In Proceedings of the 10th ACM Workshop on Hot Topics in Networks, HotNets-X, pages 7:1–7:6, New York, NY, USA, 2011. ACM.

[17] OpenFlow Specification. v1. 1.0.

[18] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In Proceedings of the 8th international conference on Emerging networking experiments and technologies, CoNEXT '12, pages 241–252, New York, NY, USA, 2012. ACM.