Formal models of software-defined networks

Feng, H.

Chapter 4

A Reo Model of SDNs

In this chapter, we present a formal model of SDNs based on the Reo language. Using Reo we regard components in an SDN as constraints imposed on the interactions among the parties engaged in the processing of network packets. Starting with a small set of simple constraints, we obtain a declarative description of switches in the data plane as well as controllers in the control plane. The composition of these components is supported through other simpler connectors which give a global description of the topology of the network. Using the constraint automata semantics of Reo, the result is a compact finite state model for SDN particularly suited for formal verification, a direction that we will explore in the next chapter.

To scale up to handle large networks, our resulting SDN model is compositional in the sense that the meaning of the entire computer network is obtained by composing that of the individual models of the switches, network topology, and controllers. Furthermore, the resulting model is independent of the (possibly infinite) sequences of packets traversing the network.

4.1 Modeling the data plane

To begin with, we describe the switches of the data plane as Reo circuits, and we translate them into their corresponding constraint automaton. Then, we describe two examples of controllers managing a simple network with two switches. The goal is to send packets from one host to another. We conclude by combining the automata of these two layers with a network topology.

In the context of software-defined networking, a packet refers to a discrete unit of data that is transmitted over a network. It contains the actual data being sent, along with a header that contains the control information necessary for its routing and handling within the SDN infrastructure. We see a packet as a record $\pi : Fields \to Data$ assigning fields from a finite set of $Fields$ to data in $Data$. We denote a packet by $\pi = [f_0 = d_0, f_1 = d_1, ..., f_n = d_n]$, and use the notation $\pi.f$ to denote the value of the field f of the packet π.

We abstract from the concrete information contained in a packet, such as the source and destination IP addresses, protocol information, and source and destination ports. To cater to the latter, we assume that the set $Fields$ includes a field IPt for storing the identity of the input port of the switch where the packet is received and OPt for the output port of the switch where the packet is forwarded. This information is crucial for making forwarding decisions within the SDN network.

4.1.1 The Reo connector of a switch

As packets traverse the network, SDN switches perform specific actions on each packet based on its header information, such as forwarding the packet to specific ports or modifying its headers. Controllers can dynamically program these actions based on their global view of the network. Figure 4.1 introduces the Reo connector representing a switch with an interface consisting of input ports $\{P_0, P_1, ..., P_n\}$ and output ports $\{Q_0, Q_1, ..., Q_m\}$. Here both n and m are greater than or equal to 0 so that a switch has always at least two ports: P_0 and Q_0. Port P_0 is used to receive messages from the controller (or controllers) supervising the switch, whereas port Q_0 is meant for sending packets to the controller. All other ports are connected to other switches or open to the environment for communication with hosts. The input ports receive packets, and the output ports send packets.

We can describe the behavior of the Reo connector representing a switch using three scenarios.

1. The first one is when a packet π is received from a host or another switch. In this case, the input port receiving the packet is P_i for some $1 \leq i \leq n$. The transformer `AddIpt_i` of the channel connected to P_i assign $\pi.IPt$ to i and outputs to node A a triple $(FlowMsg, \pi, \emptyset)$. The first component of the triple is the tag $FlowMsg$ indicating that π is an ordinary network packet with no side effect on the flow table. The last component is the subset of

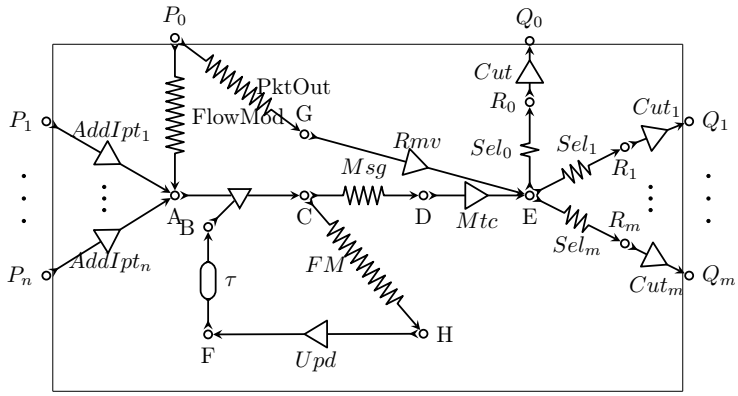


Figure 4.1: The Reo circuit of a switch

output ports of the switch where the packet needs to be forwarded. The above triple is paired with the current flow table stored in τ and received by the filters FM and Msg. These filters check the first component of the triple. In our case, only the filter Msg will succeed, and will pass the triple $(FlowMsg, \pi, \emptyset)$ together with the table τ to the transformer Mtc via node D. This transformer matches the packet π against the table τ, executes the corresponding field assignment modifying π into a new packet π' and outputs the pair (π', F) to node E. Here the set F contains all output ports where the packet π' needs to be forwarded, according to the action of the matching pair in the flow table τ.

The filters Sel_i regulate the forwarding by outputting the pair (π', F) to node R_i if $i \in F$. Note that the same pair may be duplicated to many nodes, and in case $F = \emptyset$ it will be dropped. Also, If $0 \in F$ then the packet is forwarded to the controller. From the node R_i the transformer Cut_i receiving as input the pair (π', F) will output the packet π', removing the information about the forwarding ports.

2. The second situation is when a $PktOut$ message from the controller is received at the input port P_0. A $PktOut$ message is a triple $\langle FlowMsg, \pi, F\rangle$ consisting of a tag $FlowMsg$ as in the previous case, a packet pi and a set of output ports F where π needs to be forwarded. Only the filter $PktOut$ lets this triple flow to the node G, where a transformer receives it, removes the tag, and outputs the pair (π, F) to node E. The selection and forwarding of π to each port in F are as before.

3. The third and last situation is when a $FlowMod$ message from the controller is received at the input port P_0. Also in this case it consists of a triple $\langle t, B, A\rangle$, but unlike the previous cases, this message is meant to update the table stored in τ. More specifically, B is a Boolean condition on $Fields$ matching the pair of τ to be updated, and A is the action for field updating and packet forwarding. The tag t can be either add, $remove$ or $modify$ to add (B, A) on top of table τ, remove the first pair (b, a) of τ with b implying B, or to modify the first pair (b, a) of τ with b implying B into the new pair (b, A). Note that in the case of $t = remove$, the action A does not play any role.

   Of the two filters with input at P_0 only the filter $FlowMod$ will succeed, so the triple $\langle t, B, A\rangle$ can be paired with the current flow table τ and reach node C. Here the filter Msg will fail but FM will succeed, passing all $\langle t, B, A\rangle$ and τ to the transformer Upd. This transformer will update the table τ as described in the triple $\langle t, B, A\rangle$, and will output a new table τ'. The latter is stored as the new current table by the variable channel with input node F.

4.1.2 Constraint automata for switches

While the Reo circuit of a switch may look complicated, its actual constraint automaton is rather simple. It consists of only one state (because all channels used have one single state) and three types of transitions (see Fig. 4.2).

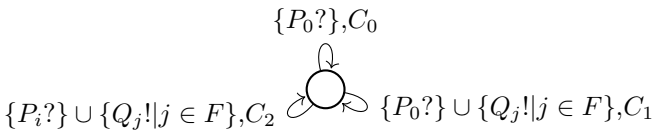


Figure 4.2: Constraint automaton of a switch

The conditions C_0, C_1 and C_2 are:

1. C_0: $P_0 = \langle t, B, A\rangle \wedge t \neq Msg \wedge \tau^\bullet = Upd(\langle \tau, P_0\rangle)$;
2. C_1: $P_0 = \langle Msg, \pi, F\rangle \wedge \bigwedge_{j\in F} Q_j = \pi$;
3. C_2: $Mtc(\langle \tau, \langle Msg, \pi[i/Ipt], \emptyset\rangle\rangle) = \langle \pi', F\rangle \wedge \tau^\bullet = \tau \wedge \bigwedge_{j\in F} Q_j = \pi'$.

Condition C_0 specifies when a $FlowMod$ message is received by a switch so that the flow table is updated. Transitions labeled by condition C_1 or C_2 depend

on the subset of output ports F received as input from P_0 or assigned after a matching action. This means there is a concrete transition for each possible subset of the output ports, but only one will eventually be chosen. Condition C_1 concerns $FlowMsg$ messages received by a controller, while condition C_2 defines the handling of a packet received from a host or another switch.

If we assume that in a switch the number of input ports is n and that the number of output ports is m, then the resulting constraint automata will have one state and $1 + 2^m + (n-1) * 2^m$ transitions.

Each switch in the data plane can be considered as a Reo connector interacting with others only via its input and output ports, while all other nodes and memory cells of the components are hidden. For example, while too large to depict here, the constraint automaton of the data plane composed of two simple switches connected by a synchronous channel as described in Figure 4.3 consists of one state, two memory cells (one for each switch flow table) and 26 transitions, which can be generated using automated tools [6].

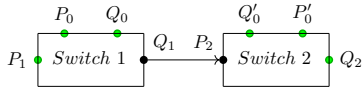


Figure 4.3: Two connected switches

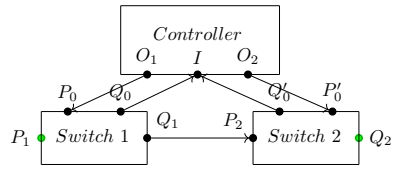


Figure 4.4: A controller and two switches

4.2 Modeling the control plane

The SDN control plane contains a set of controllers. Controllers are typically programmed using various programming languages to define network policies by handling events and configuring flow rules, e.g., matching criteria and actions. Each controller behaves as a reactive system, responding to `PktIn` messages received from switches by sending back either `PktOut` or `FlowMod` messages. We abstract from any full-fledged controller programming language and assume they are specified as Reo connectors, and thus with a behavior described using constraint automata. Input ports and output ports represent the connection of a controller with the switches under its control. Controllers can communicate with each other to allow for synchronization. Figure 4.4 shows a simple example of a controller with two switches.

We proceed with an example. The controller described in Figure 4.5 guarantees a flow of messages from the host connected to port P_1 to the host connected to port Q_2. It is connected to two switches through the output ports O_1 and O_2, and receives `PktIn` messages from the two switches via the input ports I. By chasing the circuit we see that the controller updates the flow table of both switches every time a new packet is received by switch 1 that does not match any condition of the table. The topmost sequencer regulates first the sending of a `FlowMod` message to O_1, then to O_2 and finally, it allows for the sending of the corresponding `PktOut` to O_1.

The second controller shown in Figure 4.6, has a similar specification: it allows for flowing a packet from P_1 to Q_2, but each time it reacts to incoming `PktIn` messages by updating the flow tables of both switches without waiting to receive a `PktIn` message from the second switch.

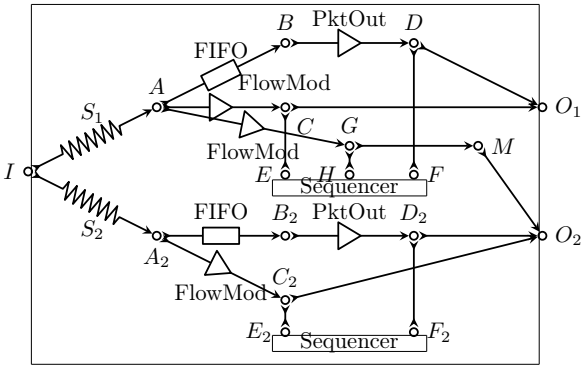


Figure 4.5: Reo circuit of controller 1

We combine the constraint automata of each controller, of the network topology, and all the switches to get a complete model of an SDN. Typically, the rate of a controller to receive messages from a switch is higher than the time needed by the controller itself to process the message and react accordingly. Therefore we use a **Queue** channel between the output ports of each switch and the input port of the controller (instead of synchronous channels $\{Q_0, I\}$ and $\{Q_0', I\}$ in Fig. 4.4). For the connections between switches and from controller to switches we use synchronous channels, but, of course, other channels with delay could be easily used instead. The Reo queue connector and its associated constraint automata with memory are described below.

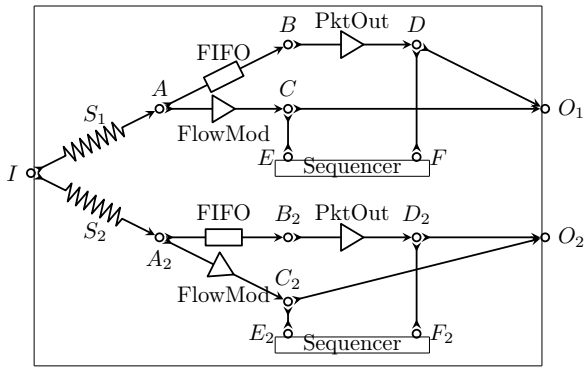
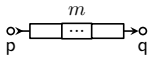
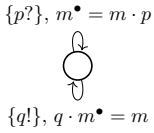


Figure 4.6: Reo circuit of controller 2



The Reo *Queue* connector behaves as a FIFO1, but it has an unbounded internal buffer m. As such, data can always be received from the input port p and stored in the buffer. If the buffer is non-empty then the first element received by p flows from the buffer to the output port q.



Both Reo circuits of the controllers described in the example above guarantee packets flowing from one host to another, but they are implemented differently and their automata are language-distinguishable. For example, when controller 1 receives a `PktIn` message, it sends a `FlowMod` message to switch one and another `FlowMod` to switch two so that a packet can pass the second switch directly without needing to wait for the table to be updated. Controller 2 however, every time receives a `PktIn` message, sends a `FlowMod` message only to the switch from which it received the message, with a consequence the updating of the flow tables of each switch happens only when a packet passes through it.

The constraint automata for controllers in Figure 4.5 and Figure 4.6 are shown in Figure 4.7 and Figure 4.8, respectively. Both automata start from the initial state 1 and always move to the state 2 when they receive a `PktIn` message from the first switch. Receiving a `PktIn` message from the second switch changes the state to 5 to the first controller and 4 to the second controller. They send either one or two `FlowMod` messages to update the table of one or two switches, and

then both send `PktOut` messages to let the packet continue its flowing to the host. After that, both automata move back to the initial state and are ready to react to new incoming messages.

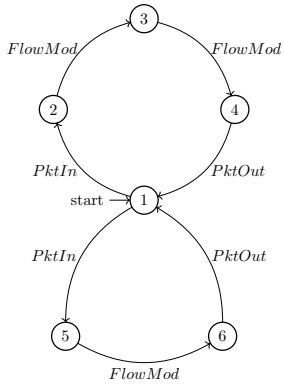


Figure 4.7: Constraint automaton for controller 1

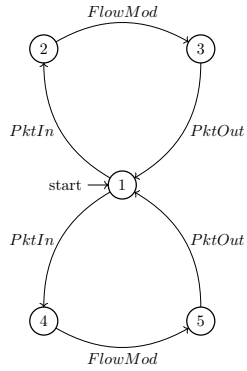


Figure 4.8: Constraint automaton for controller 2

4.3 SDN models for two controller algorithms

In this section, we combine the data plane and control plane (as in 4.1 and 4.2) together with the channel *Queue*. Since we have two different models of the controller (in Figure 4.5 and Figure 4.6), below shows two algorithms for each controller.

Network :

```
        Swith: S1, S2
        Controller: C1
        Connection: S1.Q1 -> S2.P2
        In_port: S1.P1
        Out_port: S2.Q2

Controller 1:
        Def PktIn(pkt){
        IF  pkt.inport = In_port(S1)
            Out = Out1; SwtP = Q1;
            Match = "tcp_dst = pkt.tcp_dst";
            Action1 = "Fwd(Q1)";
            Action2 = "Fwd(Q2)";
            FlowMod (<Add, Match, Action1>) to Out1;
            FlowMod (<Add, Match, Action2>) to Out2;
        ELSE
            Out = Out2; SwtP = Q2;
            Match = "tcp_dst = pkt.tcp_dst";
            Action1 = "Fwd(SwtP)";
            FlowMod (<Add, Match, Action1>) to Out;
        FI
        }

        PktIn (pkt);
        PktOut (Msg, pkt, Action1) to Out
```

The constraint automaton of the whole SDN model by compiling Controller 1 is in Figure 4.9. In the automaton, each state has five loop transitions with conditions:

1. $\{P_1\}$, packet enter into the Queue of switch 1;
2. $\{P_1\}$, packet enter into the Queue of switch 2;
3. $\{P_1\}$, packet drop in the switch 1;
4. $\{P_1\}$, packet drop in the switch 2;
5. $\{P_1, Q_2\}$, packet pass through the switch 1 and switch 2.

The initial state 1 chooses either switch 1 or switch 2 for receiving `PktIn` message, then the controller sends a `FlowMod` message to the chosen switch for installing a certain rule in it, after sending another `FlowMod` message to the alternative switch, the controller sends a `PktOut` message to the chosen switch. For

```
Network:
        Swith: S1, S2
        Controller: C1
        Connection: S1.Q1 -> S2.P2
        In_port: S1.P1
        Out_port: S2.Q2

Controller 2:
        Def PktIn(pkt){
        IF  pkt.inport = In_port(S1)
            Out = Out1; SwtP = Q1;
        ELSE
            Out = Out2; SwtP = Q2;
        FI
        }

        PktIn (pkt);
        Match = "tcp_dst = pkt.tcp_dst";
        Action = "Fwd(SwtP)";
        FlowMod (<Add, Match, Action>) to Out;
        PktOut (Msg, pkt, Action) to Out
```

example, if the first `PktIn` message comes from switch 1, the specific conditions of this automaton are:

1. `PktIn`: $q_1 = q_1^{\bullet} \cdot PktIn$;
2. `FlowMod`: $\tau_1^{\bullet} = Upd(\langle \tau_1, \langle t, b, Act_1 \rangle \rangle), t \neq Msg$;
3. `FlowMod`: $\tau_2^{\bullet} = Upd(\langle \tau_2, \langle t, b, Act_2 \rangle \rangle), t \neq Msg$;
4. `PktOut`: $Q_2 = \pi \wedge PktOut = \langle Msg, \pi, Q_2 \rangle$;

To note q_1 means the memory of Queue in switch 1, τ_1 means the flow table in switch 1, τ_2 means the flow table in switch 2, q_2 means the memory of Queue in switch 2. If the first `PktIn` message comes from switch 2, the specific conditions of this automata are:

1. `PktIn`: $q_2 = q_2^{\bullet} \cdot PktIn$;
2. `FlowMod`: $\tau_2^{\bullet} = Upd(\langle \tau_2, \langle t, b, Act_2 \rangle \rangle), t \neq Msg$;
3. `PktOut`: $Q_2 = \pi \wedge PktOut = \langle Msg, \pi, Q_2 \rangle$.

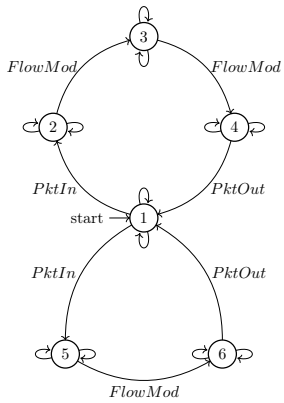


Figure 4.9: The automatons for the SDN with controller 1

The constraint automaton of the whole SDN model by applying controller 2 is in Figure 4.10.

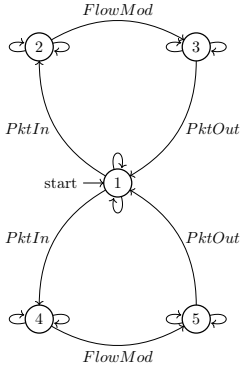


Figure 4.10: The automatons for the SDN with controller 2

Similar to the first algorithm, each state in the Fig 4.8 has five loop transitions, the conditions of these loops are the same, the difference is controller only installs one rule in the flow table of the switch each time when it receives a `PktIn` message. If the first `PktIn` message comes from switch 1, the specific conditions of this automaton are:

1. `PktIn`: $q_1 = q_1^{\bullet} \cdot PktIn$;
2. `FlowMod`: $\tau_1^{\bullet} = Upd(\langle \tau_1, \langle t, b, Act_1 \rangle \rangle), t \neq Msg$;
3. `PktOut`: $PktOut = \langle Msg, \pi, SwtP \rangle$;

If the first `PktIn` message comes from switch 2, the specific conditions of this automaton are:

1. `PktIn`: $q_2 = q_2^\bullet \cdot PktIn$;
2. `FlowMod`: $\tau_2^\bullet = Upd(\langle \tau_2, \langle t, b, Act_2 \rangle \rangle), t \neq Msg$;
3. `PktOut`: $Q_2 = \pi \wedge PktOut = \langle Msg, \pi, Q_2 \rangle$.

It is easy to see these two algorithms are very similar, both in program and automata. However they are distinguishable in the language, e.g., for the loop above in these two automata, algorithm 1 has two $FlowMod$s in sequence, but algorithm 2 has only one $FlowMod$ between $PktIn$ and $PktOut$.

4.4 Related work

The recent interest in the application of formal methods to software-defined networks started with VeriCon [14], an interactive verification system based on first-order logic to model admissible network topologies and network invariants. Similar to our model is a finite state machine model of SDN introduced in [113]. In this work model checking is possible via a translation to binary decision diagrams, under a similar assumption to ours: controllers are described as finite-state machines. Our approach however is based on a declarative description of both controllers, switches, and network topology as a Reo circuit, that is automatically, and compositionally, translated into a finite automaton.

Different than our declarative approach, [2] proposes an actor-based modeling to verify concurrent features of SDN via the ABS tool suite. The use of automata in our work instead of actors makes it easier to specify real-time and other quantitative properties of SDN. We do not explore this direction in this paper, but we leave it for future work. Variation of regular expressions has been very successful in modeling network programming languages [90, 100, 4]. In particular, NetKAT offers a sound and complete algebraic reasoning system with an interesting coalgebraic decision procedure. However, NetKAT only models a stateless snapshot of the data plane traversed by a single packet. There is no update of flow tables and no multiple packages are possible. Also, TLA+ [72] has been used to model the behavior of SDN but in a very restrictive manner allowing only a single switch [61].

Formal models are used not only to verify properties of an SDN such as consistency of flow tables, violation of safety policies, or forwarding loops, but also for

finding flaws in security protocols using CSP and the model checker PAT [110].

4.5 Conclusion

In this chapter, we presented a Reo model of SDN, based on a novel semantics for constraint automata with memory, recently studied in [57]. The difference is in a neater treatment of the values in the memory before and after the execution of a transition. The model is stateful and allows concurrency at the level of controllers but also at the level of the packets. The model can immediately be used for verification of quantitative and qualitative properties of SDN, such as consistency of flow tables, violation of safety policies, or forwarding loops. In the next chapter, we show how this model can be used for verification through a translation of Reo into the language Promela of the model checker Spin. In the future, we plan to verify these properties by using tools like ReoLive [29], or mCRL2 [65], which are part of the Reo framework [89]. Another line of research easily supported by our model is the development of simulation and visualization tools for packets flowing into the network.