



A Reo Model of Software Defined Networks

Hui Feng^{1(✉)}, Farhad Arbab^{1,2}, and Marcello Bonsangue^{1,2}

¹ LIACS, Leiden University, Leiden, The Netherlands

² CWI, Amsterdam, The Netherlands

{h.feng,f.abbab,m.m.bonsangue}@liacs.leidenuniv.nl

Abstract. Reo is a compositional coordination language for component connectors with a formal semantics based on automata. In this paper, we propose a formal model of software defined networks (SDNs) based on Reo where declarative constructs comprising of basic Reo primitives compose to specify descriptive models of both data and control planes of SDNs. We first describe the model of an SDN switch which can be compactly represented as a single state constraint automaton with a memory storing its flow table. A full network can then be compositionally constructed by composing the switches with basic communication channels. The reactive and proactive behaviour of the controllers in the control plane of an SDN can also be modelled by Reo connectors, which can compose the connectors representing data plane. The resulting model is suitable for testing, simulation, visualization, verification, and ultimately compilation into SDN switch code using the standard tools already available for Reo.

Keywords: Formal model · Software defined networks · Reo · Constraint automata · Component composition · Coordination

1 Introduction

Since the concept of software defined network (SDN) was introduced in 2006 [9] it has become increasingly popular in both academia and industry as a new architecture for operating and managing computer networks via the OpenFlow protocol [19]. In traditional networks, the control plane (where the packet forwarding strategy is set up) is tightly coupled with the data plane (where the actual packet forwarding happens) and distributed in a multitude of hardware devices. Because no entity has a global view of the network, and the size and complexity of today's networks are very large, it has become extremely complicated to program network-wide decisions for end-to-end policies and to verify their compliance with global objectives.

Different from traditional network, SDN offers a network architecture that decouples data from its routing control, and places network intelligence and

This research is supported by China Scholarship Council.

© Springer Nature Switzerland AG 2019

Y. Ait-Ameur and S. Qin (Eds.): ICFEM 2019, LNCS 11852, pp. 69–85, 2019.

https://doi.org/10.1007/978-3-030-32409-4_5

states in a logically centralized routing control entity, the so called controllers. Controllers operate independently from network switches which contain programmable forwarding tables that are set up and managed by the controllers. Since controllers can be programmed, SDN enables the application of formal methods to prove the correctness of computer networks. In the recent years several formal models of SDN (e.g. [2, 15, 16]) have been proposed in order to test or check that a network behave correctly.

In this paper we present a formal model of SDN based on Reo [3], a graphical language for compositional construction of interaction protocols, manifested as connectors. A connector consist of several typed channel and nodes, arranged in a graph of edges and vertices. Every edge in this graph represents a channel of a specific type and every vertex represents a node. The type of a channel determines its data-flow behaviour. Nodes regulate data-flow by non-deterministically selecting data items available through their incoming channel ends and replicating them through their outgoing channel ends. Nodes with both incoming and outgoing channel ends are called mixed nodes. Nodes with no incoming channel end are called source nodes, and those with no outgoing channel end are called sink nodes. Source and sink nodes collectively comprise the boundary nodes of a connector, forming the interface that regulates its communication with the environment. Every connector can be described by functional constraints that relate the timing and the contents of the data-flows at its interface [7]. Reo was originally introduced as a coordination language. Since its introduction, however, Reo has become a domain-specific language for compositional specification of protocols based on an interaction-centric model of concurrency [4, 14].

Using Reo we regard components in an SDN as constraints imposed on the interactions of parties engaged in the processing of network packets. Starting with a small set of simple constraints, we obtain a declarative descriptions of switches in the data plane as well as controllers in the control plane. Composition of these components is supported through other simpler connectors which give a global description of the topology of the network.

The formal semantics of Reo is based on automata [7] and as such it supports formal analysis, testing and verification as well as distributed automatic code generation [14]. For a more compact representation and for enabling constraints depending on stored data we consider basic channels with memory, and as such we present a variation of the original semantics of Reo to support constraints on stored and to be stored data. The result is a compact finite state model for SDN particularly suited for formal verification using techniques as in [17]. While we are only considering functional modelling in this paper, extensions for capturing the notions of time, quality of service, resources, as well as probabilistic behaviour can be captured by similar extension of the underlying Reo model [6].

In order 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. The resulting model is independent from the possibly infinite sequences of packets traversing the network.

Recent interest in the application of formal methods to software defined networks started with VeriCon [8], 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 [25]. 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. Another relevant work on automated verification is [22]. Our approach however is based on a declarative descriptions of controllers, switches, and network topology as a Reo circuits, whose automatic composition yields a finite automaton.

Different from our declarative approach, [1] proposes an actor-based modelling to verify concurrent features of SDN via the ABS toolsuite. The use of automata in our work instead of actors make it easier to specify real time and other quantitative properties of SDN. We do not explore this direction in this paper, leaving it for future work. Variation of regular expressions have been very successful in modelling network programming languages [2, 21, 23]. In particular NetKAT offers a sound and complete algebraic reasoning systems with an interesting coalgebraic decision procedure. However NetKAT models only a stateless snapshot of the data plane traversed by a single packet. It does not support update of flow tables nor routing of multiple packets. TLA+ [18] has also been used to model the behaviour of SDN but in a very restrictive manner, allowing only a single switch [16]. 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 [24].

This paper proceed as follows. In Sect. 2 we give a brief introduction to the main concepts of software defined networks, while in Sect. 3 we introduce Reo and give a new automata based semantics using memory cells for storing data. This model is used in Sect. 4 where we present a Reo circuit for the data plane and the control plane of an SDN. We conclude with an example showing the semantic difference between two controllers.

2 A Primer on Software Defined Networks

Network management includes many different tasks that, traditionally, have been realized through manufacturer-specific low-level languages for the configuration of hardware network devices, e.g., switches and routers. The primary function of a network management task is to ensure transport of packets, and entails two planes: the control plane for making routing decisions and the data plane concerned with packet forwarding. In traditional networks, the control plane is coupled with the data plane on each hardware device. As such the control plane is highly distributed, with no global view of the network, making it impossible to program network-wide decisions and verify their compliance with global specifications.

SDN offers a network architecture that simplifies the design and deployment of network management tasks: the control plane is a logically centralized controller that gathers information from the data plane and provides a global view to applications running on top of the controller. These applications make packet routing decisions based on the global view and distribute the decisions to the data plane via the controller using the OpenFlow protocol [19].

Each switch in the data plane consists of a number of ports where packets are received or forwarded. Further, each switch is connected to at least one controller, from which it may receive or to which it may send messages. The basic messages forwarded from switch to switch are packets. A packet consists of a finite set of fields, grouped in header information and pure data, as the two packets in the example below show, where the header of each packet contains the information about the `tcp` and `ethernet` destination address of the packet:

```
tcp_dst:22, eth_dst:11 | data: d1      tcp_dst:23, eth_dst:11 | data: d2
```

Forwarding of packets is implemented in each switch through a flow table, a memory store consisting of an ordered set of pairs (b, a) . Here b is a Boolean condition on the packet fields (the so called matching criteria) and a is the corresponding action to be executed on the matching packet. The order of the matching-action pairs gives a priority on the application of the matching condition. There are basically three types of actions: *forwarding* a packet to one or more ports of the switch, *dropping* a packet, and *updating* a field of a packet with some value. For example, the leftmost packet above matches the first rule of table below and it is forwarded to the output ports 3 and 4. The rightmost packet however matches only the last rule and it is forwarded to port 1 after its field `tcp_dst` is updated to 22.

Matching condition	Action
<code>tcp_dst:22</code>	<code>Forward[3, 4]</code>
<code>tcp_dst:23, eth_dst:12</code>	<code>drop</code>
<code>true</code>	<code>tcp_dst := 22; Forward[1]</code>

Controllers and switches communicate through messages. A `PktIn` message is a packet sent from a switch to a controller, typically to be processed there or to trigger an update of the flow tables. A `PktOut` message sent from the controller to a switch consists of a packet together with a flow table action to be executed by the switch. This way a packet need not pass through the flow table but is, for example, immediately forwarded to other switches.

The flow table of a switch is updated by `FlowMod` messages, another type of message from a controller to a switch. Each `FlowMod` message consists of a `ModType` t (`Add`, `Remove`, `Modify`), a matching condition b and an action a . If $t = \text{Add}$ then the pair (b, a) is added on top of the table (higher priority), while if $t = \text{Modify}$ then the first pair in the flow table (b', a') with b implying b' is

substituted with the pair (b, a) . In remaining case when with $t = \text{Remove}$ the first pair in the flow table (b', a') with b implying b' is removed from the table. In this case the action a does not play any role and therefore can be considered empty. Those three types of messages plus dedicated packets to communicate data allow controllers to gather information about the network and manage it.

3 Reo and Constraint Automata

Reo is a coordination language for compositional construction of component connectors [3]. The emphasis in Reo is on connectors, their behaviour and composition out of simple channels. Reo can also be used to define an interaction protocol as a connector, a graph-like structure that enables (a)synchronous data flow along its edges. Each edge is called a channel and it specifies constraints on the flows of data at its ends. A channel end is either a source end through which the channel accepts data, or a sink end through which the channel offers data. Multiple channel ends coincident at a vertex of the connector together form a node. Nodes have predefined ‘merge-replicate’ behaviour: a node repeatedly accepts a datum from one of its coincident sink ends, chosen non-deterministically, and offers that datum through all of its coincident source ends.

3.1 Constraint Automata

Constraint automata are a formalism to describe the “behaviour” of Reo channels and their composition as connectors [7]. Constraint automata can be thought of as conceptual generalizations of finite state automata where data constraints influence applicable state transitions.

We assume a finite set \mathbb{D} of data ranged over by d , a finite set \mathbb{P} of ports ranged over by p, q (note that ports in Reo are distinct from ports in SDN switches), and a finite set \mathbb{M} of memory cells ranged over by m . Further, let \mathcal{F} be a set of function symbols and \mathcal{P} a set of predicate symbols. Each predicate symbol and each function symbol comes with an arity, the number of arguments it expects. A term is defined as follows:

$$t ::= d \mid p \mid m \mid m^\bullet \mid f(t, \dots, t)$$

Terms are used in constraints defined by the following predicate formulas:

$$\phi ::= \top \mid p = t \mid m = t \mid m^\bullet = t \mid P(t, \dots, t) \mid \phi \wedge \phi \mid \neg\phi$$

The constraint $p = t$ denotes the equality between the value passing through the port p , and the value obtained by evaluating the term t ; $m = t$ is the equality between the value stored in the memory m before evaluating the constraint and the value denoted by t ; $m^\bullet = t$ is equality between the value stored in the memory m immediately after the evaluation of the constraint and the value denoted by t . The others are just the usual constraints.

In order to define the satisfaction of constraints, we assume the existence of a function $\hat{f} : \mathbb{D}^n \rightarrow \mathbb{D}$ for each $f \in \mathcal{F}$ of arity n , and a subset $\hat{P} \subseteq \mathbb{D}^m$ for each predicate symbol $P \in \mathcal{P}$ of arity m . For fixed sets of input ports I , output ports O and hidden ports H , the evaluation of constraint is defined by using the function $\alpha : I \cup O \cup \mathbb{M} \rightarrow \mathbb{D}_\perp$, and an environment $\eta : H \rightarrow \mathbb{D}_\perp$ assigning values to hidden ports. α is used for the visible components of a Reo connector. Here $\alpha(A)$ represents the value passing through the port A unless $\alpha(A) = \perp$ that denotes the absence of flow of data through port A . Similarly $\alpha(m)$ denotes the value stored in the memory cell m .

We denote by At the set of all atoms α . Note that m^\bullet is not a part of an atom, because it refers to the value of m after the evaluation of a transition. Therefore we need pairs of atoms, one for the current values stored in memory cells, and another for storing the side effect of evaluation, i.e., the value of a memory cell after the evaluation. Evaluations of guards is defined inductively as follows:

$$\begin{array}{ll}
\alpha_1 \alpha_2 \models_\eta \top & \\
\alpha_1 \alpha_2 \models_\eta p = t & \text{iff } \alpha_1(p) = \llbracket t \rrbracket_{\alpha_1 \alpha_2}^\eta \\
\alpha_1 \alpha_2 \models_\eta m = t & \text{iff } \alpha_1(m) = \llbracket t \rrbracket_{\alpha_1 \alpha_2}^\eta \\
\alpha_1 \alpha_2 \models_\eta m^\bullet = t & \text{iff } \alpha_2(m) = \llbracket t \rrbracket_{\alpha_1 \alpha_2}^\eta \\
\alpha_1 \alpha_2 \models_\eta P(t_1, \dots, t_n) & \text{iff } \langle \llbracket t_1 \rrbracket_{\alpha_1 \alpha_2}^\eta, \dots, \llbracket t_n \rrbracket_{\alpha_1 \alpha_2}^\eta \rangle \in \hat{P} \\
\alpha_1 \alpha_2 \models_\eta \phi_1 \wedge \phi_2 & \text{iff } \alpha_1 \alpha_2 \models_\eta \phi_1 \text{ and } \alpha_1 \alpha_2 \models_\eta \phi_2 \\
\alpha_1 \alpha_2 \models_\eta \neg \phi & \text{iff } \alpha_1 \alpha_2 \not\models_\eta \phi
\end{array}$$

Finally, we define the evaluation of a guard without hidden ports as follows:

$$\alpha_1 \alpha_2 \models \phi \text{ if and only if there is } \eta \text{ such that } \alpha_1 \alpha_2 \models_\eta \phi.$$

Here $\llbracket t \rrbracket_{\alpha_1 \alpha_2}^\eta$ denotes the value of the term t and is defined inductively by:

$$\begin{aligned}
\llbracket d \rrbracket_{\alpha_1 \alpha_2}^\eta &= d \\
\llbracket p \rrbracket_{\alpha_1 \alpha_2}^\eta &= \begin{cases} \alpha_1(p), & \text{if } p \in I \cup O \\ \eta(p), & \text{if } p \in H \end{cases} \\
\llbracket m \rrbracket_{\alpha_1 \alpha_2}^\eta &= \alpha_1(m) \\
\llbracket m^\bullet \rrbracket_{\alpha_1 \alpha_2}^\eta &= \alpha_2(m) \\
\llbracket f(t_1, \dots, t_n) \rrbracket_{\alpha_1 \alpha_2}^\eta &= \hat{P}(\llbracket t_1 \rrbracket_{\alpha_1 \alpha_2}^\eta, \dots, \llbracket t_n \rrbracket_{\alpha_1 \alpha_2}^\eta)
\end{aligned}$$

We are now ready for the definition of constraint automata with memory cells describing operationally the behaviour of a Reo connector.

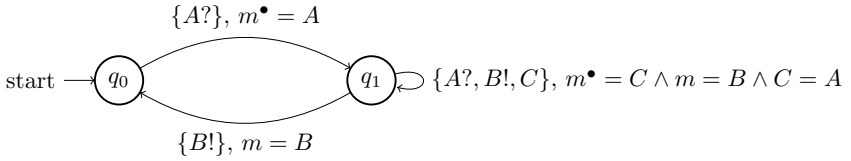
Definition 1. A constraint automaton is a tuple $(Q, I, O, H, M, \longrightarrow, q_0)$ where Q is a finite set of states with $q_0 \in Q$ the initial state, $I, O, H \subseteq \mathbb{P}$ are sets of ports known by the automaton, $M \subseteq \mathbb{M}$ is the set of memory cells, and \longrightarrow is a transition relation with $q \xrightarrow{N, \phi} q'$ denoting a transition from q to q' synchronizing a set of ports $N \subseteq I \cup O \cup H$ under the data constraint ϕ . We assume that the ports appearing in ϕ are a subset of N and the memory cells occurring in ϕ are a subset of M .

An *execution* of a constraint automaton is described by means of infinite strings [12] in At^ω . An infinite string $\alpha \cdot w$ is an execution from the state q , denoted by $\alpha \cdot w \in E(q)$ if and only if there is a transition $q \xrightarrow{N, \phi} q'$ such that the following three conditions hold:

1. $\forall p \in I \cup O, p \notin N$ iff $\alpha(p) = \perp$;
2. $w = \alpha' \cdot w'$ and $\alpha\alpha' \models \phi$;
3. $w \in E(q')$

By the above definition a constraint of a transition $q \xrightarrow{N, \phi} q'$ is evaluated in an execution $\alpha \cdot w$ starting from q with respect to its first two atoms. Furthermore, only the ports in N fire, meaning that a value passes through them as recorded by α , and the rest of the string w is an execution of the target state q' .

Consider the following constraint automaton:

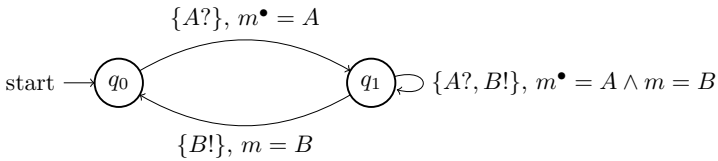


Here “?” and “!” are syntactic means for indicating which ports belong to I and O , respectively. The unmarked ports belong to H . An example of an execution of the above automaton starting from q_0 is the infinite string:

$$[A = 1, B = \perp, m = 22] \cdot [A = 3, B = 1, m = 1] \cdot [A = 5, B = 3, m = 3] \cdot [A = \perp, B = 5, m = 5] \cdot [A = 7, B = \perp, m = 33] \cdot \dots$$

Note that the value of the memory of the second element of the string is equal to the value at port A of the first element, and the value of port B of the second element. Similarly for the value of A in the second element and the value of B and the memory m in the third element.

The above automaton has the same executions from the initial state as the following automaton without hidden ports.



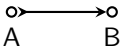

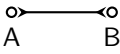

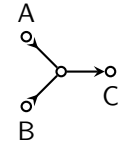

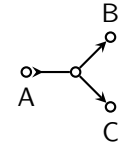

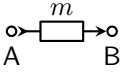
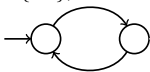
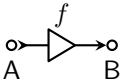

While in general it is not always possible to remove all hidden ports without modifying the set of executions, for simplicity and when there is no problem, in the sequel we will simplify a constraint automaton by removing hidden ports obtaining an automaton with the same structure (states and transitions) and the same executions from its initial state.

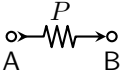
The language of a constraint automaton consists of the projection with respect to the ports of all executions starting from the initial state. A language represents the behaviour of the automaton as visible from the environment. Therefore, only input and output ports are visible, but not hidden ports or memory cells. For example, the language accepted by the above two constraint automata is the same and it includes the following infinite string:

$$[A = 1, B = \perp] \cdot [A = 3, B = 1] \cdot [A = 5, B = 3] \cdot [A = \perp, B = 5] \cdot \dots$$

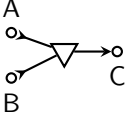
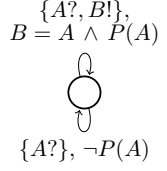
3.2 Basic Channels and More Complex Connectors

Next, we briefly introduce the constraint automata and their graphical representation for all basic Reo channels [3, 17] we use in this paper.

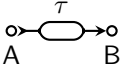
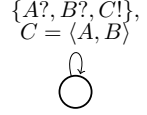
	<p>The <i>synchronous channel</i> accepts data from its input port A, and it passes them synchronously to its output port B.</p>	$\{A?, B!\},$ $A = B$ 
	<p>The <i>synchronous drain</i> has two input ports A and B, from which it accepts any data, but only when the two ports can be synchronized. The data received as input is not important, only the ports' synchronization matters.</p>	$\{A?, B?\}$ 
	<p>The <i>non-deterministic merger</i> receives data from either A or B and sends them to the sink node C synchronously. If data is available from both A and B at the same time, one of them is chosen non-deterministically.</p>	$\{A?, C!\},$ $C = A$  $\{B?, C!\},$ $C = B$
	<p>The <i>replicator</i> receives data from A and replicates them to both sink nodes B and C.</p>	$\{A?, B!, C!\},$ $B = A \wedge$ $C = A$ 
	<p>The <i>FIFO1</i> channel receives data from the input port A if the internal buffer m is empty. The data is stored in the buffer, which can only contain at most one data item. When m is full its content flows to the output port B and it becomes empty. The behavior of a similar channel with dot inside is represented by the automaton with the other state as the starting state.</p>	$\{A?\}, A = m \bullet$  $\{B!\}, B = m$
	<p>The <i>transformer</i> channel applies a user-defined function f to a data item consumed from its source end A, and synchronously offers $f(A)$ through its channel end B.</p>	$\{A?, B!\},$ $B = f(A)$ 



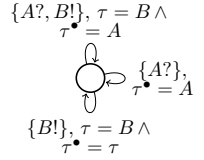
The pattern of *filter* channel $P \subseteq \text{Data}$ specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted through its source end if its sink end can simultaneously dispense d ; all data items $d \notin P$ are always accepted through the source end but are immediately lost.



The *PairMerger* accepts two data items d_1 and d_2 through the source ends A and B , merges them and synchronously offers the pair $\langle d_1, d_2 \rangle$ through its sink end C .



The *variable* can accept a data item d through source end A , update its memory τ , synchronously offer the data stored in τ through sink end B if B fires; also it can directly synchronously offer τ through B if B fires but A doesn't fire, τ remains in the buffer.



Note that the PairMerger uses a binary function symbol $\langle -, - \rangle$ interpreted as the usual pairing. In all automata in the table, we assume that the ports known by each automaton are those used in the channels.

A Reo circuit is built out of some basic channels via the join operation which is performed by joining common ports of the channels. On the automata level, the join operation is realized by the following product construction.

Definition 2. *The product of the two constraint automata $A_1 = (Q_1, I_1, O_1, H_1, M_1, \longrightarrow_1, q_1)$ and $A_2 = (Q_2, I_2, O_2, H_2, M_2, \longrightarrow_2, q_2)$ with disjoint sets of states Q_1 and Q_2 , and disjoint sets of memory cells M_1 and M_2 is:*

$$A_1 \bowtie A_2 = (Q, I, O, H, M_1 \cup M_2, \longrightarrow, \langle q_1, q_2 \rangle)$$

where $Q = Q_1 \times Q_2$, $I = (I_1 - O_2) \cup (I_2 - O_1)$, $O = (O_1 - I_2) \cup (O_2 - I_1)$, $H = (I_1 \cap O_2) \cup (I_2 \cap O_1) \cup H_1 \cup H_2$, and \longrightarrow is defined by the following rules:

$$\frac{q_1 \xrightarrow{N_1, \phi_1} q'_1 \text{ and } q_2 \xrightarrow{N_2, \phi_2} q'_2 \text{ and } \text{Prt}_1 \cap N_2 = \text{Prt}_2 \cap N_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, \phi_1 \wedge \phi_2} \langle q'_1, q'_2 \rangle}$$

Here $\text{Prt}_1 = I_1 \cup O_1 \cup H_1$, and $\text{Prt}_2 = I_2 \cup O_2 \cup H_2$.

Figure 1 shows an example of composition of a non-deterministic merger (on the left) on ports $\{A?, B?, C!\}$ with a synchronous channel (second automata from the left) acting on port $\{C?, D!\}$. The result is a new automaton with C as hidden port (third automaton from the left), which however is language equivalent to the automaton of a non-deterministic merger (the rightmost one) on ports $\{A?, B?, D!\}$.

Note that the port C is a hidden port in the resulting automaton because it is an output port of one channel and input of the other. It is not hard to see that the join operation is associative and commutative.

As another example, in Fig. 2 we introduce the circuit of a three-port sequencer and its corresponding constraint automaton [11]. This three-port-sequencer regulates the flow of data from ports A, B and C, in a sequential order. Similar sequencers can be defined for any number of ports.

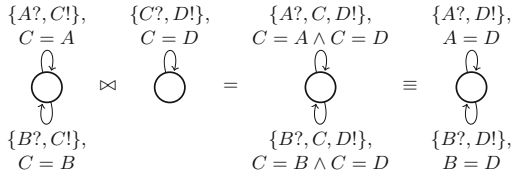


Fig. 1. The example of automata conjunction

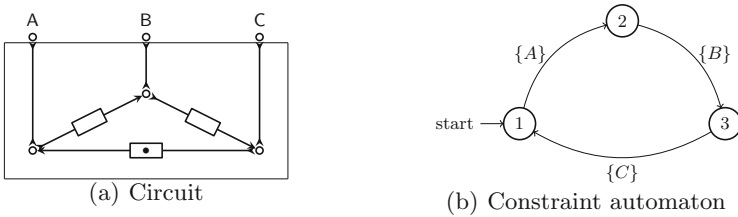


Fig. 2. A three-port sequencer

4 A Reo Model of Software Defined Networks

In this section, we present an SDN model based on the Reo language. First, we describe the switches of the data plane as Reo circuits, and we translate it into its corresponding constraint automaton. Afterwards, 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.

4.1 Data Plane

The basic data type we use is that of a packet. We see a packet as a record $\pi : Fields \rightarrow Data$ assigning fields from a finite set of $Fields$ to data in $Data$. We denote a packet by $\pi = [f_0 = d_0, f_1 = d_1, \dots, f_n = d_n]$, and use the notation $\pi.f$ to denote the value of the field f of the packet π . The set $Fields$ is assumed to include a field IPt for storing the identity of the input port of the switch

where the packet is received, OPt for the output port of the switch where the packet is forwarded.

Figure 3 introduces the Reo circuit representing a switch with an interface consisting of input ports $\{P_0, P_1, \dots, P_n\}$ and output ports $\{Q_0, Q_1, \dots, Q_m\}$. 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 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.

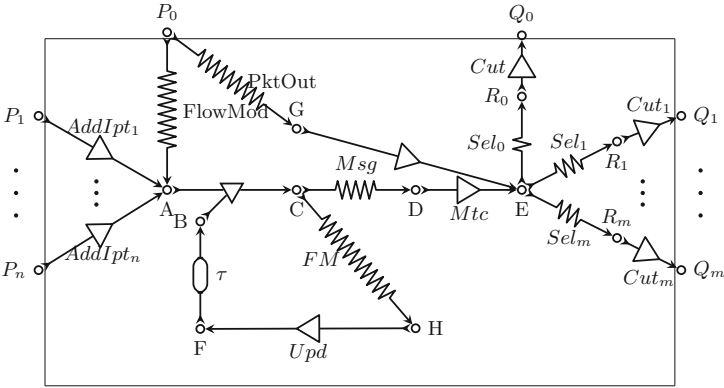


Fig. 3. Reo circuit of one switch

We can describe the behaviour of the circuit representing a switch by means of 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 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 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 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 π 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 is 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 consist 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 $t = \textit{remove}$, 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 .

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).

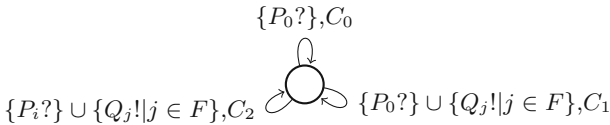


Fig. 4. Constraint automaton of a switch

The conditions C_0 , C_1 and C_2 are:

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

Condition C_0 specifies when a *FlowMod* message is received by a switch so that the flow table is updated. Transitions labelled by condition C_1 or C_2 are dependent on the subset of output ports F received as input from P_0 or assigned after a matching action. This means that 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 Fig. 5 consists of one state, two memory cells (one for each switch flow table) and 26 transitions, which can be generated using automated tools [5].

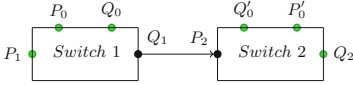


Fig. 5. Data plane

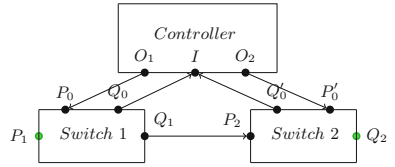


Fig. 6. A simple example

4.2 Control Plane and the Whole SDN Model

The SDN control plane contains a set of controllers. Each controller behaves as a reactive system, responding to *PktIn* messages received from switches by sending either *PktOut* or *FlowMod* messages. We assume controllers to be specified as Reo circuits, and thus with a behaviour described by means of constraint automata. Input ports and output ports represent the connection of a controller with the switches under its control. Figure 6 shows a simple example of a controller with two switches. A controller need not know the operational details of any of the switches that it controls (e.g., their automata); its concern consists of deciding *when* to update the flow table of a switch, and *what* modification constitutes that update. For instance, it may decide to modify the flow table of a switch

in reaction to the switch receiving (and escalating) a packet for which it has no matching condition.

For example, the controller described in Fig. 7 guarantees a flow of messages from the host connected to port P_1 to the host connected to port Q_2 . It updates the flow table of a switch every time a new packet is received that does not match any condition of the table. In the second controller shows in Fig. 8, we see a similar specification of a controller flowing a packet from P_1 to Q_2 , but each time it updates switches apart.

We combine constraint automata of controllers and switches together to get a complete model of an SDN. Because the rate of forwarding by a switch is different from the rate of processing by a controller, we put a **Queue** channel between output ports of each switch and input ports of the controller (like channels $\{Q_0, I\}$ and $\{Q'_0, I\}$ in Fig. 6), a synchronous channel between input ports of each switch and output ports of the controller (like channels $\{O_1, P_0\}$ and $\{O_2, P'_0\}$). Here are the description of Queue.

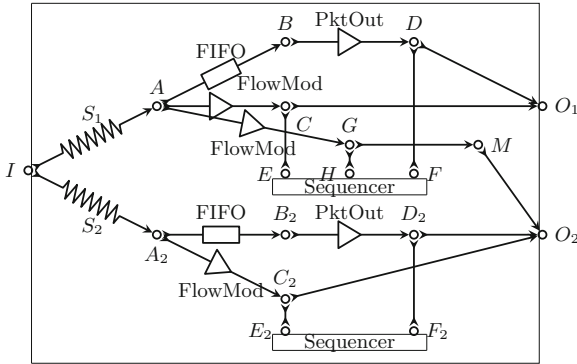


Fig. 7. Reo circuit of controller 1

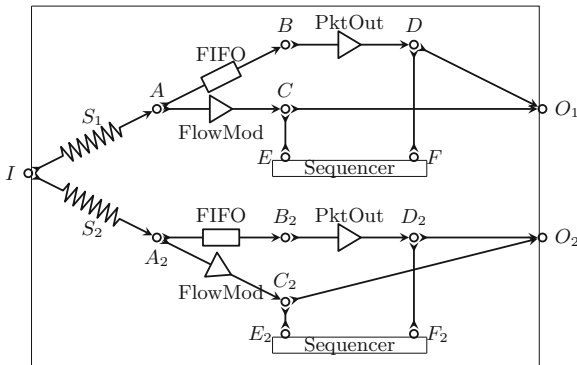
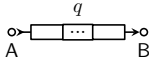
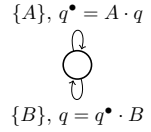


Fig. 8. Reo circuit of controller 2



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



While the two models guarantee packets to flow from one host to another, they have different semantics and therefore they are language distinguishable. The two cases have different behaviours because in the first case when the controller receive a `PktIn` message, it sends a `FlowMod` message to switch one and another `FlowMod` to switch two, so that the packet π can pass the two switches directly. But in the second case, every time the controller receives a `PktIn` message, it just sends a `FlowMod` message to the current switch, so π can only pass the current switch.

5 Conclusion

In this paper we presented a Reo model of SDN, based on a novel semantics for constraint automata with memory, recently studied in [13]. 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 future, we plan to verify these properties by using tools like ReoLive [10], or mCRL2 [17], which are part of the Reo framework [20] and can directly generate executable code for the switches. Another line of research easily supported by our model is the development of simulation and visualization tools for packets flowing into the network.

References

1. Albert, E., Gómez-Zamalloa, M., Rubio, A., Sammartino, M., Silva, A.: SDN-actors: modeling and verification of SDN programs. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 550–567. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_33
2. Anderson, C.J., et al.: NetKAT: semantic foundations for networks. ACM Sigplan Not. **49**(1), 113–126 (2014)
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
4. Arbab, F.: Proper protocol. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. LNCS, vol. 9660, pp. 65–87. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30734-3_7

5. Arbab, F., Koehler, C., Maraïkar, Z., Moon, Y.J., Proença, J.: Modeling, testing and executing Reo connectors with the eclipse coordination tools. Presented at the 5th International Workshop on Formal Aspects of Component Systems (2008)
6. Arbab, F., Meng, S., Moon, Y.J., Kwiatkowska, M., Qu, H.: Reo2MC: a tool chain for performance analysis of coordination models. In: van Vliet, H., Issarny, V. (eds.) Proceedings of the of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 287–288. ACM (2009)
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.* **61**(2), 75–113 (2006)
8. Ball, T., et al.: Vericon: towards verifying controller programs in software-defined networks. *SIGPLAN Not.* **49**(6), 282–293 (2014)
9. Casado, M., et al.: SANE: a protection architecture for enterprise networks. In: Keromytis, A.D. (ed.) *USENIX Security Symposium*, p. 50. USENIX Association (2006)
10. Cruz, R., Proença, J.: ReoLive: analysing connectors in your browser. In: Mazzara, M., Ober, I., Salaün, G. (eds.) *STAF 2018. LNCS*, vol. 11176, pp. 336–350. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_25
11. Ghassemi, F., Tasharofi, S., Sirjani, M.: Automated mapping of Reo circuits to constraint automata. *Electron. Notes Theor. Comput. Sci.* **159**, 99–115 (2006)
12. Izadi, M., Bonsangue, M.M.: Recasting constraint automata into Büchi automata. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) *ICTAC 2008. LNCS*, vol. 5160, pp. 156–170. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85762-4_11
13. Jongmans, S.S., Kappé, T., Arbab, F.: Constraint automata with memory cells and their composition. *Sci. Comput. Program.* **146**, 50–86 (2017)
14. Jongmans, S.S.T.Q.: Automata-theoretic protocol programming. Ph.D. thesis, Leiden University (2016)
15. Kang, M., et al.: Formal modeling and verification of SDN-openflow. In: 6th International Conference on Software Testing, Verification and Validation, pp. 481–482. IEEE (2013)
16. Kim, Y.M., Kang, M., Choi, J.Y.: Formal specification and verification of firewall using TLA+. In: Daimi, K., Arabnia, H.R. (eds.) *Proceedings of the International Conference on Security and Management (SAM)*, pp. 247–251 (2017)
17. Kokash, N., Krause, C., de Vink, E.P.: Data-aware design and verification of service compositions with Reo and mCRL2. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C. (eds.) *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2406–2413. ACM (2010)
18. Lamport, L.: *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley (2002)
19. McKeown, N., et al.: Openflow: enabling innovation in campus networks. *Comput. Commun. Rev.* **38**(2), 69–74 (2008)
20. Proença, J., Clarke, D., De Vink, E., Arbab, F.: Dreams: a framework for distributed synchronous coordination. In: Ossowski, S., Lecca, P. (eds.) *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pp. 1510–1515. ACM (2012)
21. Reitblatt, M., Canini, M., Guha, A., Foster, N.: FatTire: declarative fault tolerance for software-defined networks. In: Foster, N., Sherwood, R. (eds.) *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot topics in Software Defined Networking*, pp. 109–114. ACM (2013)

22. Schnepf, N., Badonnel, R., Lahmadi, A., Merz, S.: Automated verification of security chains in software-defined networks with synaptic. In: 2017 IEEE Conference on Network Softwarization (NetSoft), pp. 1–9. IEEE (2017)
23. Soulé, R., et al.: Merlin: a language for provisioning network resources. In: Seneviratne, A., Diot, C., Kurose, J., Chaintreau, A., Rizzo, L. (eds.) Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies, pp. 213–226. ACM (2014)
24. Xiang, S., Zhu, H., Xiao, L., Xie, W.: Modeling and verifying TopoGuard in OpenFlow-based software defined networks. In: Pang, J., Zhang, C., He, J., Weng, J. (eds.) 2018 International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 84–91. IEEE Computer Society (2018)
25. Zakharov, V.A., Smelyansky, R.L., Chemeritsky, E.V.: A formal model and verification problems for software defined networks. *Autom. Control. Comput. Sci.* **48**(7), 398–406 (2014)