Formal models of software-defined networks

Feng, H.

Chapter 5

Implementing Reo into Promela

In this chapter, we study a subclass of constraint automata with local variables. The fragment denotes an executable subset of constraint automata for which synchronization and data constraints are expressed in an imperative guarded command style, instead of a relational style as in ordinary constraint automata. To demonstrate the executability property, we provide a translation scheme from symbolic constraint automata to Promela, the language of the model checker Spin. As a proof of concept, we model in Reo a software-defined network circuit, translate it into Promela, and use the Spin model checker to verify that our model satisfies some basic temporal properties.

5.1 A short introduction to Promela

Promela is a formal language widely used to specify concurrent systems and is supported by Spin, a Linear Temporal Logic (LTL) model checker [48, 49]. Theoretically, any LTL formula φ can be converted into a Büchi automaton [40, 104], as well as the negative of φ. To verify if a system satisfies an LTL property φ, we first construct an automaton A for the system, then compute the synchronous product of the Büchi automaton $BA(\neg\varphi)$ and A. If the language of this product is empty, then we called the original system to satisfy φ, otherwise φ is not satisfied with the system, and the counter-examples will be provided, too.

A Promela program is composed of a set of processes, each processes run concurrently and interact through shared channels. Both synchronous and asynchronous communication between processes is supported by its constructs since the modeling and analysis of processes, communication channels, and synchronization primitives are granted. For synchronous communication, a channel works in a rendezvous mode with no buffer (zero capacity), for asynchronous communication, channels work as a Fifo buffer with a non-zero user-specified capacity. All of these features make Promela suitable for verifying the correctness of complex systems, including hardware and software.

One of the most powerful model checkers for Promela is SPIN, SPIN can exhaustively explore all possible system behaviors and check various properties, such as safety, liveness, and temporal logic properties automatically. In a nutshell, each Promela process is transformed by Spin into a finite state automaton, processes are then synchronized into a single system automaton. Similarly, linear temporal properties expressed in the usual LTL syntax, are transformed into finite state automata. The automata representing both the Promela program and the LTL properties are exploited by Spin to verify and assert satisfaction of the properties [48, 49].

The syntax of LTL are:

$$\top \mid \neg\varphi \mid a \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2 \mid \Diamond\varphi \mid \Box\varphi$$

The property φ could be **true**, **false**, an **atom**, **or**, **and**, **next**, **until**, **eventually**, **always**. The semantics are:

$\alpha \models \top$	iff	α is true
$\alpha \models \neg\varphi$	iff	$\alpha \not\models \varphi$
$\alpha \models a$	iff	$a \in \alpha[0]$
$\alpha \models \varphi_1 \vee \varphi_2$	iff	$\alpha \models \varphi_1$ or $\alpha \models \varphi_2$
$\alpha \models \varphi_1 \wedge \varphi_2$	iff	$\alpha \models \varphi_1$ and $\alpha \models \varphi_2$
$\alpha \models X\varphi$	iff	$\text{suffix}(\alpha, 1) \models \varphi$
$\alpha \models \varphi_1 U \varphi_2$	iff	$\exists j \geq 0$, $\text{suffix}(\alpha, j) \models \varphi_2$ and $\text{suffix}(\alpha, i) \models \varphi_1$, $\forall 0 \leq i < j$
$\alpha \models \Diamond\varphi$	iff	$\exists i \geq 0$, $\text{suffix}(\alpha, i) \models \varphi$
$\alpha \models \Box\varphi$	iff	$\forall i \geq 0$, $\text{suffix}(\alpha, i) \models \varphi$

where $\alpha = \alpha[0]\alpha[1]\alpha[2]\ldots$ is an infinite sequence of states.

5.2 Symbolic constraint automata

In this section, we give an idea of symbolic constraint automata, and show how the composition method of Reo applies to the symbolic constraint automata.

We have seen in Chapter 3 that Reo circuits are usually specified using constraint automata [13]. In that case, each transition is labeled by synchronization and relational data constraints. To enhance the expressive power, we have considered an extension of constraint automata with memories. In both cases data constraints are declarative, and serve more as a specification than an implementation of an executable Reo circuits. In this section, we introduce a subset of constraint automata with transitions labeled by symbolic guarded actions. We show that, under some simple consistency conditions, symbolic constraint automata can be implemented and systematically translated into the Promela language. In fact, most of the original basic Reo connectors can be modeled as symbolic constraint automata, with only exceptions filter channels with predicates that constraint output ports and transformer channels that update input ports.

The basic building blocks of a symbolic constraint automaton include a finite set $\mathbb{D}$ of data ranged over by d, and a set $\mathbb{V}$ of variables, ranged over by x, y and z. We use variables to denote Reo ports shared between a connector and its environment, with different read and write permissions. An input port is a variable that the environment writes to (i.e., *put* a value into it), and from which a Reo connector destructively reads (i.e., *take* a value from it). Symmetrically, an output port is a variable that a connector writes to (i.e., *put*) and from which the environment destructively reads (i.e., *take*). Local variables are internal to a channel and can be used to store data. An assignment of variables to values is a function $\sigma{:}\mathbb{V} \to \mathbb{D}$. We range over input, output, and local variables by i, o, and v, respectively.

To abstract from concrete actions, we use function symbols (ranged over by f) and predicate symbols (ranged over by P). As usual, each function symbol f comes equipped with an arity and coarity, i.e. the number of arguments it expects and it returns, respectively. Similarly, predicate symbols come with an arity. We assume the natural interpretation of such function and predicate symbols, as executable function on $\mathbb{D}^n \to \mathbb{D}^m$ and as a decidable subset of $\mathbb{D}^n$, respectively. To simplify the notation, we identify predicate and function symbols with their interpretations. Here, n is the arity and m the coarity. Syntactic substitution in a term t of every occurrence of variable x for a term t_x is denoted as usual by

$t[t_x/x]$.

Terms of symbolic constraint automata are defined by the grammar:

$$t ::= d \mid x \mid f(\bar{t})$$

Terms denote tuples of data values. Here the (local or port) variable x denotes the data value it stores, while $\bar{t}$ is a shorthand notation for a finite sequence of terms $t_1, \ldots, t_n$, and $f(\bar{t})$ represents a tuple of values resulting from the computation f when executed with input values $\bar{t}$. The size of these tuples depends on the arity and coarity of f.

A *guarded action* α consists of a predicate and an assignment,

$$P(\bar{x}) \to \bar{y} := t\,.$$

We call $P(\bar{x})$ the guard of the guarded action α, and $\bar{y} := t$ the action that is executed when the guard is true. In general, we refer to the guard of a guarded action α by $g(\alpha)$, and the assignment at the right-hand side of α by $a(\alpha)$. We implicitly assume that the size of $\bar{y}$ corresponds to the coarity of t. To avoid problems with simultaneous assignments, and without loss of generality, only different variables may occur on the left-hand side of the assignment in an action. This way, for $z \in \bar{y}$, we can denote by $a(\alpha)_z$ the z-projection of the tuple of values resulting from evaluating the term $a(\alpha)$.

Given finite subsets I, O, and V of $\mathbb{V}$ denoting some input, output, and local variables, respectively, let $Act(I, O, V)$ be the set of actions α such that all variables occurring in $g(\alpha)$ are in $I \cup V$, all variables on the left-hand side of the assignment in $a(\alpha)$ are in $O \cup V$, and all variables on the right-hand side of the assignment in $a(\alpha)$ are either in V or occurring in $g(\alpha)$. The idea is that a guard $g(\alpha)$ constrains what value input may take, based on the current value of its local variables and pending values supplied by the environment. If the guard holds then the right-hand side of $a(\alpha)$ can be satisfied using values from the local variables and values given by the environment on its inputs. The result is assigned to the variables on the left-hand side of $a(\alpha)$ changing the local store and communicating the result of the computation to the environment via the output variables. Since output ports are only used to communicate a value to the environment, we assume no occurrence of them on the guard $g(\alpha)$ and in the term on the right-hand side of the assignment $a(\alpha)$. Dually, since input ports receive values only from the environment, we assume no occurrence of them on the left-hand side

of the assignment $a(\alpha)$. We denote by $I(\alpha)$ the set of all input ports occurring in α. Similarly, we denote by $O(\alpha)$ and $V(\alpha)$ the sets of output ports and local variables, respectively occurring on the left-hand side of the assignment in α. All input ports occurring in $a(\alpha)$ must appear in the input of the guard $g(\alpha)$. The test $x = x$ denotes a guard that is true, and an assignment $x := x$ denotes the skip action. We often do not write them in a transition unless the context requires it.

For example, the guarded action $(i \leq v \to (o, v) := [i, i])$ is an action in $Act(I, O, V)$, with $I = \{i\}, O = \{o\}$ and $V = \{v\}$. But the guarded action $(i \leq v \to (o, i) := [i, o])$ is not because i and o appear both on the left-hand side and on the right-hand side of the action, respectively. Here $[-, -]$ is the pairing function, with arity 2 and coarity 2, mapping two inputs into their corresponding pair of values.

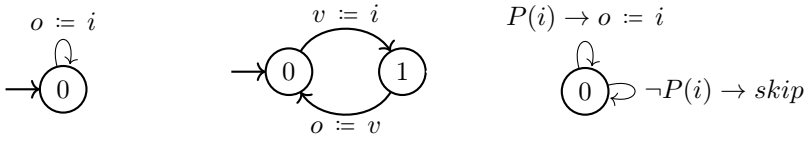
Definition 5.1. *A symbolic constraint automaton is a tuple* $(Q, q_0, I, O, V, \longrightarrow)$, *where*

- Q *is a finite set of states including the initial state* q_0,
- $I \subseteq \mathbb{V}$ *is a finite set of input ports,* $O \subseteq \mathbb{V}$ *is a finite set of output ports,* $V \subseteq \mathbb{V}$ *is a finite set of local variables such that they are mutually disjoint, i.e.,* $I \cap O = I \cap V = O \cap V = \emptyset$,
- $\longrightarrow$ *is a transition relation between states and labeled by actions in* $Act(I, O, V)$.

A transition $q \xrightarrow{\alpha} q'$ denotes the possibility of executing the action α from the state q and moving to the state q'. In order for the actual execution of α to take place, the guard of the action α must hold upon evaluation in the current state. To simplify the notation, we will not write guards of actions that are always true. Note that, differently from [57] and Chapter 3, we do not need to specify pre- and post-values of a local variable, as our actions are imperative and thus the order is implicitly given by the assignment operator.

In Figure 5.1, we show three symbolic constraint automata. The one on the left has no internal state, and the data received at the input port i is synchronously passed to the output o. This connector corresponds to the *synchronous channel* in Reo. The one in the middle has three variables: one input variable i, one output variable o, and an internal variable v. The automaton has two states: state 0 to indicate that the internal variable can be rewritten (i.e., the buffer is empty), and

state 1 to indicate that it cannot (i.e., the buffer is full). The connector assigns to v the value taken from i if it is in the empty state 0, and puts to the port o the value from v if it is in the full state 1. The two states symbolic constraint automaton is simpler than the equivalent single-state automaton (see Figure 5.2), as it avoids guards on the internal variable and the use of an extra special value $\perp$ to indicate that a variable is 'empty'. This connector corresponds to the *Fifo1 channel* in Reo. Finally, the rightmost automaton has a non-trivial guard labeling each of its two transitions. If the predicate P holds when a value is available at an input port i, then the connector behaves like a synchronous connector and passes the input value to the output port o. Otherwise, $\neg P$ holds on the value of i and the value is taken from i and lost, meaning that the component waiting for synchronization on port i is released.



(a) Synch connector (b) Fifo 1 connector (c) Filter connector

Figure 5.1: Three examples of symbolic constraint automata

An execution of a symbolic constraint automaton $(Q, q_0, I, O, V, \longrightarrow)$ is given in terms of an infinite sequence $(\sigma_i)_{i\in\mathbb{N}}$ of assignments of values to variables for which we can find an infinite sequence of states $(q_i)_{i\in\mathbb{N}}$ starting from the initial state q_0 and such that for all $n \geq 0$ there is a transition $q_n \xrightarrow{\alpha} q_{n+1}$ satisfying the following three conditions:

1. the interpretation of the guard $g(\alpha)$ holds in the assignment σ_n,

2. for all $x \in O(\alpha) \cup V(\alpha)$ the value $\sigma_{n+1}(x)$ is the x-projection of the evaluation of the variables on the left-hand side of the assignment in $a(\alpha)$ in the state σ_n, that is $\sigma_{n+1}(x) = \sigma_n(a(\alpha)_x)$;

3. for all variables not involved in the guarded action α, the value does not change, that is, $\sigma_{n+1}(y) = \sigma_n(y)$ for all $y \in (I \cup O \cup V) \setminus (I(\alpha) \cup O(\alpha) \cup V(\alpha))$.

Consecutive assignments σ_n and σ_{n+1} in a sequence represent the change of the internal and observable state of the system. The above conditions guarantee that

the guard must hold on to the values assigned to input variables in the current state before an action is taken, and that, after the execution of an action, the output variables and the local variables involved in the action change according to the action taken. The last condition guarantees that only variables that occur freely in an action get modified (for example by the environment) when executing that action. Note that after the execution of an action, input variables in $I(\alpha)$ change to a new value assigned by the environment. Dually, the value assigned to an output variable before the execution of an action is presumably taken by the environment before the new output value, assigned by the action, overwrites it. In other words, a transition in symbolic constraint automata, likewise constraint automata, assigns values to its output and local variables while being constrained on input and local variables. Also, the environment is allowed to change the values assigned to other variables not declared in the automaton, i.e., any variable in $\mathbb{V} \setminus (I \cup O \cup V)$. This represents the effect of an independent action executed by the environment in parallel with the automaton.

Local variables are used to store externally unobservable information. Only communication via input and output ports should be observable. Therefore, we define the semantics of a symbolic constraint automaton as the set of all possible executions $(\sigma_i)_{i \in \mathbb{N}}$ defined as above, but projected only on their input and output variables. As usual, two automata are then equivalent if they have the same semantics, i.e. they generate the same set of finite traces of assignments of input and output variables.

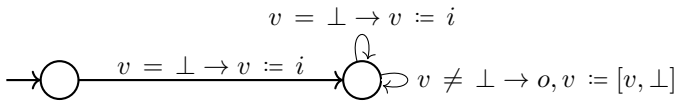


Figure 5.2: An equivalent symbolic constraint automaton for the Fifo 1 connector

Consider the symbolic constraint automaton in Figure 5.1.(b). The following is an example of a sequence of assignments recognized by that automaton:

$$[i = 1, o = 0, v = 0][i = 2, o = 0, v = 1][i = 2, o = 1, v = 1][i = 3, o = 1, v = 2] \cdots$$

Initially, the automaton is in state 0, for example, with value 1 on the input port i. The values of the two other variables do not matter at this point, and can be seen as previous values that remained stored but not accessible. By taking the transition to the state 1, the connector assigns the value of i to the internal variable v. The output port o is

blocked and cannot be changed while executing this transition, while the input port is free and here is assumed to get the value 2 from the environment. When taking the next transition the content of the variable v is put in the port o, the input is blocked and the cycle can start again. For each such a sequence we can find an equivalent sequence for the automaton in Figure 5.2, by using the extra value $\perp$ to check if the buffer is empty

$$[i = 1, o = 0, v = \perp][i = 2, o = 0, v = 1][i = 2, o = 1, v = \perp][i = 3, o = 1, v = 2] \cdots$$

Conversely, for every sequence representing the behavior of the automaton in Figure 5.2 we can find an equivalent sequence of assignments for the automaton in Figure 5.1.(b) by copying the previous value of v instead of $\perp$, and assigning an arbitrary initial value for v. In other words, the two automata are equivalent. Note that the initial state of the automaton in Figure 5.2 forces the connector to start with an empty buffer. Without this initial transition, the two automata would not be equivalent as one could start to output on port o the value stored in v.

The central operation on symbolic constraint automata is synchronization via their shared ports which are input ports for one automaton and output ports for another. Shared ports become internal local variables in the automaton resulting from the composition. No other synchronization by shared variables is allowed, as local variables are only visible within the scope of a connector. Our definition is similar in spirit to that of [57], but, in addition to that work, our symbolic constraint automata could be automatically translated to Promela. The explicit input and output variables, and the guarded command structure on the label impose some prerequisites on the product to avoid inconsistencies. In fact, we define composition only for pairs of symbolic constraint automata A_1 and A_2 such that (1) no local variables are in common, and (2) for every pair of actions α_1 and α_2 of the two automata, they synchronize only on some input ports used by one action and some output ports used by the other, but not on both input and output ports at the same time. More formally, we assume that, for all actions α_1 labeling a transition in A_1 and α_2 labeling a transition in A_2, the following holds

$$I(\alpha_1) \cap O(\alpha_2) \neq \emptyset \Rightarrow O(\alpha_1) \cap I(\alpha_2) = \emptyset$$

or, equivalently,

$$O(\alpha_1) \cap I(\alpha_2) \neq \emptyset \Rightarrow I(\alpha_1) \cap O(\alpha_2) = \emptyset$$

We call two automata with these two properties *consistent*. The intuition behind synchronizing two guarded actions α_1 and α_2 is that their data value should agree on their shared ports so that it can flow from the output of one to the input of the other actions. The above condition together with the fact that the two automata do not share local variables - and thus $V(\alpha_1) \cap V(\alpha_2) = \emptyset$ - in fact impose a causality in the execution of their actions as input is needed to update the internal state and to be passed to output

ports. Next, we define formally the synchronization of two guarded actions α_1 and α_2. Assume $I(\alpha_1) \cap O(\alpha_2) = \bar{u} \neq \emptyset$, and let

$$\alpha_1 = P_1(\bar{x}_1 \cup \bar{u}) \to \bar{y}_1 \coloneqq t_1(\bar{z}_1 \cup \bar{u}) \quad \text{and} \quad \alpha_2 = P_2(\bar{x}_2) \to \bar{y}_2 \cup \bar{u} \coloneqq t_2(\bar{z}_2)\,.$$

where $\bar{x}_1 \cup \bar{u}$ is the sequence of input and local variables occurring in P_1, $\bar{z}_1 \cup \bar{u}$ the sequence of input and local variables occurring in t_1, and $\bar{y}_2 \cup \bar{u}$ the sequence of output and local variables occurring at the left-hand side of the assignment $a(\alpha_2)$. Note that variables in $\bar{u}$ cannot occur in P_2 nor in t_2 as they are output variables for α_2. Similarly, they cannot occur in $\bar{y}_1$, as they are input variables for α_1. By definition of guarded action, they can occur in P_1. Under these circumstances, we can define the synchronization $\alpha_1 \otimes \alpha_2$ as the following guarded action

$$\alpha_1 \otimes \alpha_2 = P_1[t_{2_{\bar{u}}}/\bar{u}] \wedge P_2 \to \bar{y}_1, \bar{y_2} \cup \bar{u} \coloneqq t_1[t_{2_{\bar{u}}}/\bar{u}],\, t_2(\bar{z}_2).$$

Since the guard P_2 does not depend on output variables, we can evaluate it. If it holds we can then assign the values returned by t_2 to the output and local variables at the left-hand side of $a(\alpha_2)$. The values assigned to the shared output values $\bar{u}$ are then used in P_1 as constant replacing the input variables $\bar{u}$. If this predicate holds, then the same substitution is applied to t_1 so that we can compute the assignment. Note that $I(\alpha_1 \otimes \alpha_2) = (I(\alpha_1) \setminus O(\alpha_2)) \cup I(\alpha_2)$, $O(\alpha_1 \otimes \alpha_2) = O(\alpha_1) \cup (O(\alpha_2) \setminus I(\alpha_1))$, and $V(\alpha_1 \otimes \alpha_2) = V(\alpha_1) \cup V(\alpha_2) \cup (I(\alpha_1) \cap O(\alpha_2))$. The definition of $\alpha_1 \otimes \alpha_2$ for the symmetric case when $O(\alpha_1) \cap I(\alpha_2) \neq \emptyset$ is similar.

Definition 5.2. *Without loss of generality, let Q_1 and Q_2 be two disjoint sets of states, and V_1 and V_2 be two disjoint sets of local variables. The composition of two consistent symbolic constraint automata $A_1 = (Q_1, q_1, I_1, O_1, V_1, \longrightarrow_1)$ and $A_2 = (Q_2, q_2, I_2, O_2, V_2, \longrightarrow_2)$ is defined as the automaton $A_1 \bowtie A_2 = (Q, q, I, O, V, \longrightarrow)$ where:*

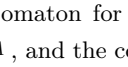
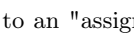
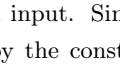
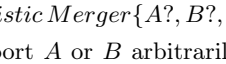
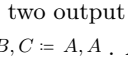
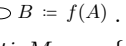
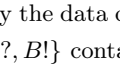
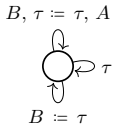
- $Q = Q_1 \times Q_2$,
- $q = \langle q_1, q_2 \rangle$,
- $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$,
- $O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1)$,
- $V = V_1 \cup V_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1)$, *and*
- $\longrightarrow$ *is defined by the following rules:*

$$\frac{q_1 \xrightarrow{\alpha_1}_1 q_1' \text{ and } q_2 \xrightarrow{\alpha_2}_2 q_2' \text{ and } IO(\alpha_1) \cap IO(A_2) = IO(\alpha_2) \cap IO(A_1)}{\langle q_1, q_2 \rangle \xrightarrow{\alpha_1 \otimes \alpha_2} \langle q_1', q_2' \rangle}$$

$$\frac{q_1 \xrightarrow{\alpha_1}_1 q_1' \text{ and } IO(\alpha_1) \cap IO(A_2) = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{\alpha_1} \langle q_1', q_2 \rangle} \qquad \frac{q_2 \xrightarrow{\alpha_2}_1 q_2' \text{ and } IO(\alpha_2) \cap IO(A_1) = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{\alpha_2} \langle q_1, q_2' \rangle}$$

where $IO(A_i) = I_i \cup O_i$ *and* $IO(\alpha_i) = I(\alpha_i) \cup O(\alpha_i)$, *for* $i = 1, 2$.

Similar to the join operation on constraint automata the above synchronization operation on symbolic constraint automata synchronizes actions on shared ports and allows independent parallel behavior for actions with no shared ports. The composition of consistent symbolic constraint automata is symmetric and, when defined, associative. Here are some other examples of symbolic constraint automata in below.

The primary distinction between symbolic constraint automata and the constraint automata defined in Chapter 3 lies in the transition label. In the former, synchronization relies on the ports used in the guarded action, whereas the latter explicitly declares which ports must synchronize. Furthermore, symbolic constraint automata are imperative and not declarative, utilizing variables as memory. For example, the constraint automaton for the synchronous connector $Sync\{A?, B!\}$ is denoted as  , and the corresponding symbolic constraint automaton removes the ports and changes the "equal to" to an "assignment"  , explicitly declaring B as an output port and A as an input. Similarly, the Synchronous Drain connector $SyncDrain\{A?, B?\}$ is modeled by the constraint automaton  , ensuring both inputs are lost synchronously when data arrives at both ports. The corresponding symbolic constraint automaton is  , where the guard of the action is always true but involves the ports A and B, while the action is just a "skip." This way, input ports A and B are forced to synchronize, but their received values are lost. The connector $Non-deterministic\,Merger\{A?, B?, C!\}$ has one output port C that receives data from either input port A or B arbitrarily. The symbolic constraint automaton is  . The connector $Replicator\{A?, B!, C!\}$ receives the data from the input port A and copies it into the two output ports B and C. The symbolic constraint automaton representing it is  . A $Transformer\{A?, B!\}$ outputs on port B the data received at input A after applying a function f. The symbolic constraint automaton is similar to the one for the synchronous channel, except for the use of f in the action of its unique transition  . The connector $PairMerger\{A?, B?, C!\}$ looks similar to $Non-deterministicMerger\{A?, B?, C!\}$, except that the output ports C receive an ordered pair formed by the data of A and B, respectively:  . Finally, the connector $Variable\{A?, B!\}$ contains an internal variable τ, that can store input data from A that is available to the output port B. Note that reading from and writing to τ can happen separately or contemporaneously:  .

5.3 From Reo to Promela

Building on the work presented in [77], we translate Reo connectors into Promela programs. We use symbolic constraint automata as a specification of Reo connectors, and use the resulting Promela process as a protocol to coordinate messages exchanged through the ports of other processes.

5.3.1 Implementing Reo ports in Promela

In Promela, a Reo port is expressed as a structure, as shown in Listing 5.1. It contains two Promela channels of capacity one: a data and a trig channel. A port in Reo is directional. We call putter the component that puts an element on the port, and getter the component that gets an element from the port. Operations on a port are blocking unless both a put and a get are performed at the same time. In which case the port *fires*, and the data is forwarded from the putter to the getter.

The data channel in the Promela implementation of a Reo port is used to forward the data message from the putter to the getter, while the trig channel is used to synchronize the putter and the getter. The reason of using two channels of size one instead of a rendezvous channel of size zero is that it is impossible in Promela to query whether a process is currently waiting on a rendezvous channel. We will later see that querying the state of a port is necessary for the protocol to coordinate the boundary processes.

Listing 5.1: definition of a Reo port in Promela

```
1 typedef port {
2     chan data = [1] of {Data};
3     chan trig = [1] of {int}; }
```

As described in Listing 5.2, two actions can be performed on a port: put and take. The function call put(q, a) atomically fills the data channel of q with the datum a, and blocks on the trig channel, waiting to synchronize with the component on the output side of q. The integer variable x is used to get a value from the trig channel, and hence to synchronize to it; the actual value communicated does not matter.

The function call take(q, a) atomically notifies, by outputting on the trig channel, that there is a component willing to take data, and blocks on the data channel, until a datum can be read and stored into the variable a. The integer value of -1 written into the synchronization channel is arbitrary, as trig is used only for signaling.

Listing 5.2: put and take functions

```
1 inline put(q,a) {
2   int x;
3   q.data!a;
4   q.trig?x }
```

```
5
6  inline take(q,a) {
7    q.trig!-1; q.data?a }
```

We describe two temporal properties in Listing 5.3 that reflect the synchronous behavior of a port. We say that a port *fires* whenever a data is exchanged between the putter and the getter. If a port does not fire, it is *silent*. In the case of an implementation of a port with two buffers, the firing property occurs whenever a port has both an input and an output request, i.e. both channels are full. We then know, due to the definition of the put and take operations, that the putter and the getter will be released from blocking on the port, and the getter will get the value from the data channel. We define some macros in Listing 5.3 to encode firing and silent property of port p as an LTL property.

Listing 5.3: Macros for firing of ports

```
1  #define p_fires (
2      !(len(p.data) == 0) && !(len(p.trig) == 0) &&
3      X((len(p.data) == 0) || (len(p.trig) == 0)))
4
5  #define p_silent    (! p_fires)
```

Ports are not typed as input or output, as that depends on the component/connector that uses them. We have seen in the previous section that within a connector, a port used in a guard must be an input port, whereas a port used on the left-hand side of the assignment of an action is an output port.

5.3.2 Implementing Reo connectors in Promela

Next we describe how to implement Reo connectors expressed as symbolic constraint automata in Promela. A symbolic constraint automaton is encoded, in Promela, as a proctype. A Promela proctype has a name, a signature, and a body. The Spin model checker executes each proctype of its main concurrently, while taking into account blocking operations on channels. As we will see, input/output variables of a symbolic constraint automaton lead to shared port channels in Promela between the protocol and the boundary processes.

Let $(Q, q_0, I, O, V, \longrightarrow)$ be a symbolic constraint automaton. Input ports I and output ports O are passed as parameters to the Promela process resulting from the translation. As expected, local variables in V are declared locally to the Promela process, i.e., within theproctype body. For each port $p \in I \cup O$, a new local variable _p is declared so as to store the value taken if p is an input or passed if p is an output. We use mytype as a generic type for input, output, and memory variables.

Each state in Q is encoded as a special value for the state variable. The state variable therefore models the control flow between the states of the automaton. For simplicity,

and without loss of generality, here we assume that $q_0 = 0$ is the initial state and $Q = \{0, 1, \ldots, n\}$.

We use the Promela non-deterministic $\mathsf{do} - \mathsf{od}$ construct to model concurrent applications of transitions of a symbolic constraint automaton. The guard (respectively, the command) of the statements in the $\mathsf{do} - \mathsf{od}$ results from the translation of the guard (respectively, the command) in the action labeling the corresponding transition. As a result, each guard in the $\mathsf{do} - \mathsf{od}$ loop contains a clause that controls that variable state has the value corresponding to the pre-state, and updates the value to the post-state in the command. The Promela language allows for non-destructive reads of channel's value: the operation A.data? < _a> assigns to the variable _a the value stored in channel A.data without actually removing the values from that channel. However, in Promela, this operation cannot be executed in a guard within a $\mathsf{do} - \mathsf{od}$ statement. We circumvent this problem by using a control variable f_i for each transition $t_i \in \longrightarrow$ of the automaton. The variable f_i can take two values. By default, f_i is 1. If the synchronization constraint of the guard of the i-th transition is satisfied but the data constraint is not, then the value of f_i is set to 0. Every transition, if taken successfully, resets all the f_i to 1.

More formally, for a symbolic constraint automaton $(Q, q, I, O, V, \longrightarrow)$ we show in Listing 5.4 its translation to Promela:

Listing 5.4: Promela code generated for a symbolic constraint automaton

```
1  proctype SCA(port P̄){
2  ∀p ∈ P, mytype _p;
3  ∀v ∈ V, mytype v;
4  ∀f ∈ {f1,...fk}. bool f = 1;
5  int state=0;
6  do
7    :: transition 1;
8    ⋮
9    :: transition k;
10 od }
```

Here $Q = \{0, \ldots, n\}$, $q = 0$, $P = I \cup O$, $\overline{\mathsf{v}} = V$, and the remaining overlined variables are just consecutive sequences of them. For every (input or output) port $\mathsf{p} \in \overline{P}$ there is a variable _p associated with it on which we store the value communicated via the port. The control variable state is used to store the current state of the automaton (thus it ranges between 0 and n), and variables $\{\mathsf{f1}, \ldots \mathsf{fk}\}$ are used when evaluating predicates on values received at input ports. Here k is the number of transitions. Let $q \xrightarrow{\alpha} q'$ be the j-th transition of the symbolic constraint automaton, with $1 \leq j \leq k$, and remember that the input ports in α are $I(\alpha) = \{\mathsf{i1}, \ldots, \mathsf{im}\}$ the output ports are $O(\alpha) = \{\mathsf{o1}, \ldots, \mathsf{ol}\}$, and the local variables occurring in α are $\{\mathsf{v1}, \ldots, \mathsf{vh}\}$. Then transition j in the listing above is given by

```
state==q && fj==1 && ⋀_{i∈I(α)} Full(i.data) && ⋀_{o∈O(α)} Full(o.trig)
  -> Atomic { i1.data?<_i1>; ...; im.data?<_im>;
     if
       :: P(i1,...,im,v1,...vh) == True -> take(i1,_i1);...; take(im,_im);
          A(_i1,...,_im,v1,...vh, _o1,...,_ol);
          put(o1,_o1);...; put(ol,_ol);
          state=q';f1=1;...; fk =1;
       :: else -> fj=0;
     fi }
```

where P is a function encoding the guard of α and A is a function encoding the assignment action of α. When the control variables are not used in a transition (for example because there is no predicate on input variables), then it is possible to simplify the generated Promela code by removing such control variables. Similarly, when there is only one state, the variable state can be removed as its value is constant.

We list several examples of translation of symbolic constraint automata to Promela. Listing 5.5 gives the Promela code resulting from the translation of the symbolic constraint automaton in Figure 5.1.(a) that models the *Sync* connector.

Listing 5.5: Promela code generated for a Sync connector

```
proctype Sync(port A; port B){
mytype _a; mytype _b;//internal values in Sync
int state = 0; //initial state
do
:: state==0 && Full(A.data) && Full(B.trig)
   -> Atomic{take(A, _a); _b =_a; put(B,_b); state = 0};
od }
```

Here the parameter port A is the input port of the Sync channel, and port B is the output port. There are no local variables except those associated with the ports, and the control variable state that we will discuss later. The condition full(A.data)&&full(B.trig) is satisfied if and only if there is an ongoing put(A, a) operation on the input port A and an ongoing take(B, x) operation on the output port B. In this case, the Sync process atomically takes the data from port A, stores it in a local variable _a, executes the action of the associated transition of the symbolic constraint automata, i.e. _a = _b, and puts the value in port B. Of course, the single state automaton of the Sync connector could have been modeled by a much simpler Promela process without such an extra variable, but, as for the other connectors, we keep it for generality.

Listing 5.6 shows the Promela code for a two-state symbolic constraint automaton of the Fifo1 Reo connector given in Figure 5.1.(b).

Listing 5.6: Promela code generated for Fifo1 connector

```
proctype Fifo1(port A; port B){
mytype _a; mytype _b; int state = 0;
```

```
3  mytype v; //v is the buffer of the Fifo 1 connector
4  // _a,_b,v are the input, output and local variables, respectively
5  do
6  :: Full(A.data) && state==0
7     -> Atomic{take(A,_a); v=_a; state=1};
8  :: Full(B.trig) && state==1
9     -> Atomic{_b=v; put(B,_b); state=0};
10 od }
```

Besides an input and an output port, we also have a local variable v for the buffer of the Fifo1 connector. The control variable state is initially set to 0, corresponding to the initial state of the automaton.

This time, the do − od loop contains two transitions, one for each transition of the symbolic constraint automaton. One statement corresponds to the transition that moves the control from state = 0 to state = 1 if there is a pending data on the input port A. The value is then taken and assigned to the local variable v. The other statement corresponds to the transition that moves the control from state = 1 to state = 0 when there is a pending request at the output port. In which case, the stored value is forwarded to the output port B.

Next, we show how to translate the Filter connector of Figure 5.1.(c) that, unlike the other two examples, contains two transitions labeled with a non-trivial predicate on the input variable. The Promela code of the Filter connector is presented in Listing 5.7.

Listing 5.7: Promela code generated for Filter connector

```
1  proctype Filter(port A; port B){
2  mytype _a; mytype _b; int state=0; bool f1=1; bool f2=1;
3  do
4  :: state==0 && f1==1 && Full(A.data) && Full(B.trig)
5     -> Atomic{ A.data ? <_a>;
6       if
7         :: P(_a)==True -> take(A,_a); _b=_a; put(B,_b); state=0; f1=1; f2=1;
8         :: else -> f1=0;
9       fi }
10 :: Full(A.data) && state==0 && f2==1
11    -> Atomic{ A.data ? <_a>;
12      if
13        :: P(_a)==False -> take(A,_a); state=0; f1=1; f2=1;
14        :: else -> f2=0;
15      fi }
16 od }
```

Initially the control variables f1 and f2 associated with the two transitions are set to true, meaning that any transition can be potentially selected. As before, the satisfiability of each guard depends on the presence of some data at the input port A and the presence of some signal at the output port B. The predicate P is evaluated only after one of the two statements of the do − od loop is chosen. If true, the action of the transition is

taken, and the two control variables f1 and f2 are reset to true. Otherwise, the value of the associated control variable fi is set to false and the control goes back to the loop statement. In this way the i-th transition associated with fn will not be selected anymore, even if all other predicates in the guard of the statement are true (for $i = 1, 2$), which removes some undesirable livelocks.

We leave the detailed description of the encoding in Promela of a composition operator mimicking that of symbolic constraint automata but just give an example below corresponding to the composition of the Promela code generated for the Fifo1 and the Filter connectors in Listings 5.6 and 5.7, respectively.

Listing 5.8: Promela code generated by composing Filter(port A;port B) with Fifo1(port B;port C)

```
1  proctype FilterFifo1(port A; port C){
2  mytype _a; mytype _c;
3  mytype v; //v is the buffer of the Fifo 1 connector
4  mytype b; //b is a shared port that becomes a local variable
5  int state=0; //there are 1 x 2 states in total
6  bool f1=1; bool f2=1; bool f3=1; //there are 3 transitions in total
7  do
8  :: state==0 && f1==1 && Full(A.data)
9     -> Atomic{ A.data ? <_a>;
10      if
11        :: P(_a)==True -> take(A,_a); _b=_a; v=_b; state=1; f1=1; f2=1; f3=1;
12        :: else -> f1=0;
13      fi }
14 :: state==0 && f2==1 && Full(A.data)
15    -> Atomic{ A.data ? <_a>;
16      if
17        :: P(_a)==False -> take(A,_a); state=0; f1=1; f2=1; f3 =1;
18        :: else -> f2=0;
19      fi }
20 :: state==1 && f3==1 && Full(C.trig)
21    -> Atomic{_c=v; put(C,_c); state=0; f1=1; f2=1; f3 =1;};
22 od }
```

5.3.3 Other Reo connectors in Promela

In this part, we show more examples of standard Reo connectors encoded as symbolic constraint automata and translated to Promela. The symbolic constraint automata of these Reo connectors are shown in the end of Section 5.2, it is important to remark that the control variables f_i do not introduce fairness or priority among the transitions, they only control the flow so that Promela will not choose the same transition again with a guard that has already been evaluated to false. Below in Listing 5.9 shows the translation of SynchronousDrain, which has two input ports A and B, without any output ports. SynchronousDrain process automatically takes two data items from A and B when

the condition full(A.data)&&full(B.data) is satisfied, there is no data item be forwarded to any ports.

Listing 5.9: Promela code generated for Synchronous Drain connector

```
proctype Syncdrain(port A; port B){
mytype _a; mytype _b;
int state = 0;
do
:: Full(A.data) && Full(B.data) -> Atomic{take(A, _a); take(B, _b); state = 0;};
od }
```

The process of Non − deterministicMerger indicates two loops in do − od, it either chooses to forward the data from port A to C when full(A.data)&&full(C.trig) is satisfied, or choose to forward the data from port B to C when full(B.data)&&full(C.trig) is satisfied. The state has not been changed and remains to be state = 0. The code is presented in Listing 5.10.

Listing 5.10: Promela code generated for Non-deterministic Merger connector

```
proctype Merger(port A; port B; port C){
mytype _a; mytype _b; mytype _c;
int state = 0;
do
:: Full(A.data) && Full(C.trig)
   -> Atomic{take(A, _a); _c = _a; put(C, _c); state = 0;};
:: Full(B.data) && Full(C.trig)
   -> Atomic{take(B, _b); _c = _b; put(C, _c); state = 0;};
od }
```

Listing 5.11 denotes the Replicator connector in Promela code, the process executes replication of the input, then forward these two value to output port B and C if full(A.data)&&full(B.trig)&&full(C.trig) is satisfied.

Listing 5.11: Promela code generated for Replicator connector

```
proctype Replicator(port A; port B; port C){
mytype _a; mytype _b; mytype _c;
int state = 0;
do
:: Full(A.data) && Full(B.trig) && Full(C.trig)
   -> Atomic{take(A, _a); _b = _a; _c = _a; put(B, _b); put(C, _c); state = 0;};
od }
```

Transformer process has a user defined inline function f which is distinct from other processes. In Listing 5.12, f has a parameter x where x here indicates the value _a after ongoing operation take(A, _a), then be forwarded to output port B when the condition full(A.data)&&full(B.trig) is satisfied.

Listing 5.12: Promela code generated for Transformer connector

```
proctype Transformer(port A; port B){
mytype _a; mytype _b;
int state = 0;
do
:: Full(A.data) && Full(B.trig)
   -> Atomic{take(A, _a); _b = f(_a); put(B, _b); state = 0;};
od
inline f(x){
% user defined
}
```

The PairMerger connector has a $\otimes$ connected to each channels, which here in Promela implemented by _c = ⟨_a, _b⟩, after taking data from input port A and B, when conditions be satisfied. It sends the result _c to the output port C at the end. The code is in Listing 5.13.

Listing 5.13: Promela code generated for PairMerger connector

```
proctype PairMerger(port A; port B; port C){
mytype _a; mytype _b; mytype _c;
int state=0;
do
:: Full(A.data) && Full(B.data) && Full(C.trig)
   -> Atomic{take(A, _a); take(B, _b); _c = <_a, _b>; put(C, _c); state = 0;};
od }
```

Listing 5.14 supports a three-choice Variable process. The variable τ could be updated by the inputs of A when there is a input in A, output port B receives τ when there is a pending request at B. However, B can only receive the not updated τ in one transition.

Listing 5.14: Promela code generated for Variable connector

```
proctype Variable(port A; port B){
mytype _a; mytype _b; mytype τ;
int state = 0;
do
:: Full(A.data) && Full(B.trig)
   -> Atomic{ take(A, _a); _b = τ; τ = _a; put(B, _b); state = 0;};
:: Full(A.data) -> Atomic{ take(A, _a); τ = _a; state = 0;};
:: Full(B.trig) -> Atomic{ _b = τ; put(B, _b); state = 0;};
od}
```

5.4 A case study: verifying a SDN

In this section, we apply our translation of symbolic constraint automata to Promela on a software defined network model, and use the Spin model checker to verify several

temporal properties. We use the model of software defined networks introduced in [33], where all components of an SDN are represented as Reo connectors, and thus as symbolic constraint automata. The model is stateful and reflects the SDN separation between switches, controllers, and network. For each part, we briefly describe the Promela code obtained from the Reo components, and how we model the basic data flow operations of an SDN: `PktIn, PktOut, FlowMod`.

5.4.1 A Promela SDN model via symbolic constraint automata

According to the model of SDN presented in Chapter 4, we got the generated Promela code for the Reo model of a switch $\mathsf{Switch}(\mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{Q0}, \mathsf{Q1}, \mathsf{Q2})$ with two input ports $\mathsf{P1}$ and $\mathsf{P2}$, two output ports $\mathsf{Q1}$ and $\mathsf{Q2}$ as shown in Figure 5.3. As explained in the last chapter, those ports form the interface of the switch with the rest of the network. Additionally, the switch interface is extended with an input port $\mathsf{P0}$ and an output port $\mathsf{Q0}$ that serve to exchange flow messages with the SDN controller that we can model directly in Promela code. Flow messages from the controller $\mathsf{Controller}(\mathsf{P}, \mathsf{Q})$ to the switch are exchanged synchronously, via the synchronous connector $\mathsf{Sync}(\mathsf{P}, \mathsf{P0})$. On the other direction, flow messages are exchanged asynchronously via a queue connector $\mathsf{Queue}(\mathsf{Q0}, \mathsf{Q})$ from the port $\mathsf{Q0}$ of the switch to the port Q of the controller.

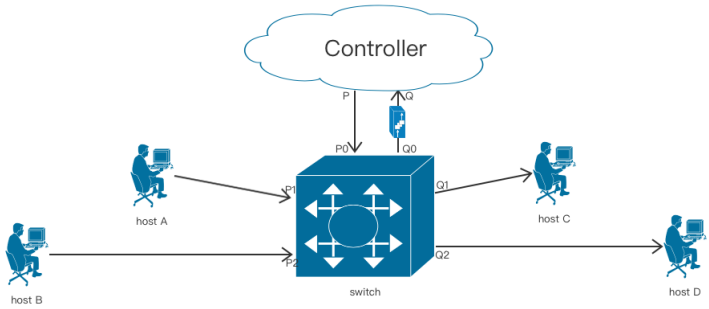


Figure 5.3: A simple example of an SDN architecture

Informally, the symbolic constraint automaton of a switch consists of a single state and few transitions labeled by the following type of guarded actions:

- α_0 is executed when a `FlowMod` message is received from the input port $\mathsf{P0}$. The action here consists in updating the flow table according to the information sent by the controller.
- α_1 is enabled when a `PktOut` message is received by the switch from port $\mathsf{P0}$. The resulting action forwards a packet to a subset (possibly empty) of output ports of

the switch. This subset is contained in the `PktOut` message from the controller.

- α_2 is enabled when one of the input port P1 or P2 of the switch receives a normal packet that is then forwarded to a subset (possibly empty) of output ports according to the current information in the flow table of the switch.

The Promela code of the switch obtained via the translation from a symbolic constraint automaton is presented (in a simplified manner for reason of space) below. The full code and the results of its verification can be found in [32].

```
proctype Switch(port P0, P1, P2, Q0, Q1, Q2){
Message m;   // either FlowMod or PktOut
Packet p;
Flowtable ft_old; ft_new
bool f1=1; f2=1; ...;fn=1;  //control variables
int state=0;    //automaton current state
do
//packet from P1 to Q1: this is a packet-forwarding α2 type of action
:: state==0 && f1==1 && Full(P1.data) && Full(Q1.trig)
    -> atomic{ take(P1,p); Match(ft_old,p,[ Q1 ]); put(Q1,p); f1=1;...fn=1}
//packet from P1 to both Q0 (PktIn message to controller) and Q2:
// this is a α2 type of action
:: state==0 && f2==1 && Full(P1.data) && Full(Q0.trig) && Full(Q1.trig)
    -> atomic{ take(P1,p); Match(ft_old,p,[Q0,Q1]); put(Q0,p);put(Q1,p);
    f1=1;...fn=1 }
//message from P0 to Q1 and Q2: this is an PktOut α1 type of action
:: state==0 && f3==1 && Full(P0.data) && Full(Q1.trig)&& Full(Q2.trig)
   -> atomic{ P0.data?<m>;
     if
      :: m==<p,[Q1,Q2]> -> take(P0,m);put(Q1,p);put(Q2,p); f1=1;...fn=1;
      :: else -> f3=0;
    fi }
//message from P0 to update flow table: this is a FlowMod α0 type of action
:: state==0 && f4==1 && Full(P0.data)
   -> atomic{ P0.data?<m>;
     if
      :: m==<p,ft_new> -> take(P0,p);update(ft_old,ft_new); f1=1;...fn=1;
      :: else -> f4=0;
    fi }
//Similar actions follows here....
...
od}
```

Each transition synchronizes some of the actors in a network (hosts, switches, controllers). Packet forwarding is done on the basis of the result of a function `Match()` of the Reo transformer channel between ports D and E in Figure 4.1. Another non-trivial function used here is `update()`, belonging to the transformer between ports F and E in Figure 4.1 and used to model a `FlowMod` operation.

In our case study as described in Figure 5.3, the network consists of a switch programmed by a controller, where hosts A and B produce packets, and hosts C and D

consume them. We abstract from the specific behaviour of the hosts and model them simply as producers and consumers of messages, respectively.

```
proctype HostA(port A){
packet p1;
atomic{p1.header = 11; p1.ipt = P1; put(A,p1)}
}
```

Host A produces a single packet that is sent to port $P1$. Here we assume a packet contains a header (with information such as the tcp/ip source or destination), and the port of the switch it is supposed to be received directly from the host.

```
proctype HostB(port B){
packet p2;
p2.ipt = P2;
do
  :: atomic{p2.header = 11; put(B,p2)};
  :: atomic{p2.header = 22; put(B,p2)}
od
}
```

The above Promela code for host B is similar to that of A, except that host B repeatedly sends packets with header 11 or 22 to port $P2$. Hosts C and D are consumers that repeatedly execute the *take* action from their ports C and D respectively. Once a packet is received, they update their own local `counter` storing the number of packets with header 11 that they receive. Below we show the Promela pseudocode of host C, that of D is similar.

```
proctype HostC(port C){
packet q1;
int counter=0;
do
  :: atomic{take(C,q1); if q1.header==11 -> counter++;}
od
}
```

Finally we give the Promela code of the controller. The controller takes a packet from port Q if available. If the packet originally passed through port $P1$ then the controller adds a rule in the flow table to forward similar messages (i.e., coming from the same address as in the header) to port $Q1$. Packets with header 22 need to be forwarded to both ports $Q1$ and $Q2$ (thus to hosts C and D). Finally, the following firewall is installed: packets with header 11 that have passed through port $P2$ must be dropped. Note that the controller will insert rules into the flow table of the switch to execute the above commands, and will apply the action itself only the first time a packet does not match any rule in the flow table, i.e., the first time the packet is forwarded to the controller.

Listing 5.15: an example of controller

```
proctype Controller(port Q, P){
Flowtable ft;
Packet p;
Message m;
do
:: Full(Q.data) && (Q.data.ipt==P1) ->
    take(Q, p);
    ft.cond = p.header; ft.action = [Q1];
    //create a rule: if match header then forward to Q1
    m=<p,ft>; put(P, m); //update flow table
    m=<p,[Q1]>; put(P, m); // Forward p to port Q1;
:: Full(Q.data) && (Q.data.header==11) && (Q.data.ipt==P2) ->
//insert a firewall rule: no message from P2
    take(Q,p);
    ft.condition = p.header; f.action = []; // drop package if comes from P2
    m=<p,ft>; put(P,m); //update flow table
:: Full(Q.data) && (Q.data.header==22) ->
    take(Q, p);
    ft.cond = p.header; ft.action = [Q1,Q2];
    //create rule: if match header then forward to Q1 and Q2
    m=<p,ft>; put(P, m); //update flow table
    m=<p,[Q1,Q2]>; put(P, m); // Forward p to port Q1 and Q2;
od
}
```

Port Q of the controller is linked to the output port of the queue connector Queue(Q0, Q) which may store at most 10 packets. This number is reasonable as we do not expect many `PktIn` messages to be forwarded to the Controller.

5.4.2 Verification and simulation

Figure 5.4 describes the scenario for which host A sends a packet with header 11 *after* host B sent a packet with header 22. Here the packet of A will arrive to C by first passing through the controller, and the packet from host B will arrive to both hosts C and D. The LTL properties associated with this scenario are

```
prop1 {[]((p1.header==11 && p1.ipt==P1)→<>(q1.header==11))}
    /* satisfied */
prop2 {[]((p2.header==22)→(<>(q1.header==22)&&<>(q2.header==22)))}
    /* satisfied */
```

Intuitively they says: The message of host A will receive B (no loop holes) and always, if B sends a 22-message, then both eventually C receives a 22-message (but not necessarily the same one), and eventually the same for D. Assuming that only B sends 22-messages then this means that every messages with header 22 received from hosts C or D is originated from host B.

Figure 5.5 describes the scenario when the packet from host A is received at the switch. As the flow table of the switch is empty, the switch forwards the packet to the

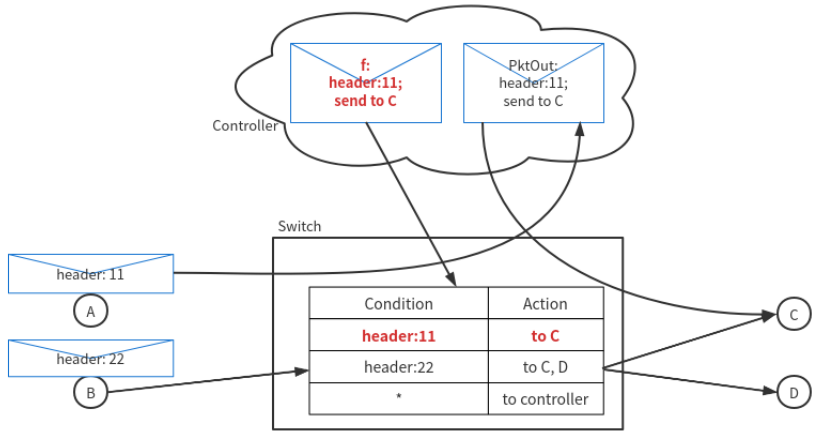


Figure 5.4: Packets from A and B arriving both to C

controller. However, the packet with the same header arriving from B is dropped. Of course, if the packet B arrives at the switch after the updates of the flow table due to the first message from host A, then the packet of B will match the flow table and be redirected to host C, violating the firewall rule. We can verify this formally in Spin via the following LTL property:

```
prop3 {<>(msg1.header==11)&&(<>[](HostC_counter==1))} /* unsatisfied */
```

This property together with the code of the three hosts A, B and C, states that there is a state in the system where a packet with header 11 is received (either from host A or B) and eventually C receive a message with header 11 and no other such a message afterwards.

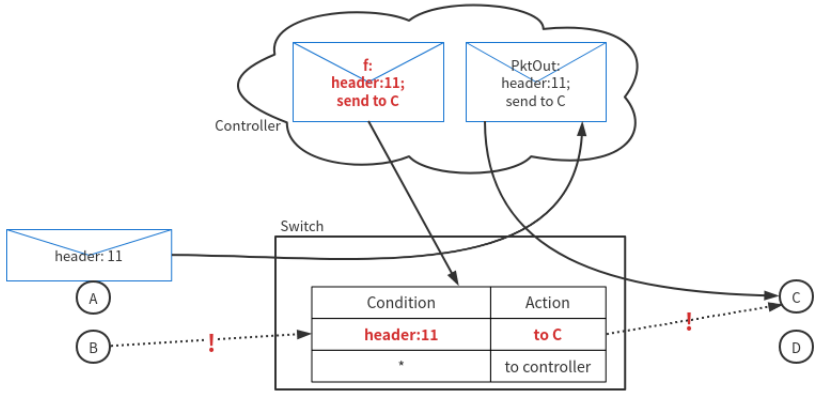


Figure 5.5: Are packets from B with header=11 always dropped?

We use the Spin model checker to verify the three LTL properties, with the following parameters: Extra Compile-Time Directives is set to 20700; the number of hash-functions in Bitstate mode to 5, and the Physical Memory Available to 102400 Mbytes. We used depth-first search with partial order reduction and Bitstate/Supertrace in order to achieve better performance [50]. The results of the verification are shown in Table 5.1.

Property	Errors found	Time usage	Depth reached	States stored	States matched	Transi -tions	State-vector
prop1	0	10.5s	479	214292	660445	1058424	2088 byte
prop2	0	8.64s	479	152057	373096	767167	2088 byte
prop3	2520	14.3s	500	183077	924560	1346688	2184 byte

Table 5.1: Verification results

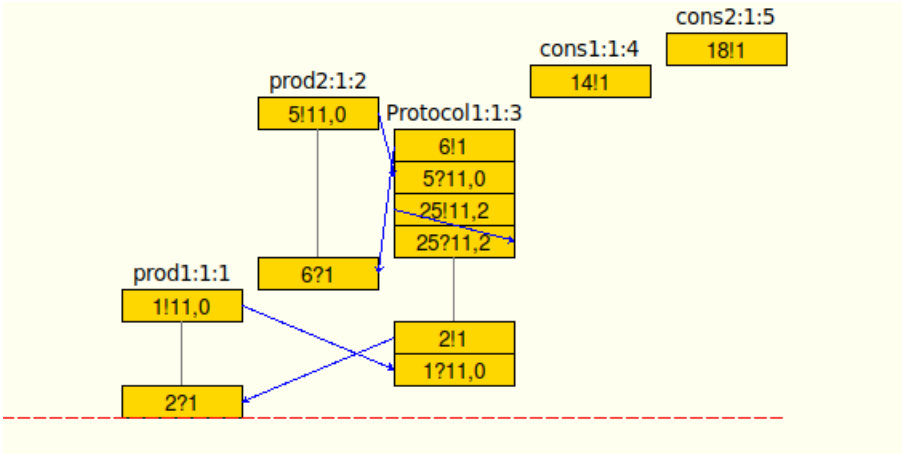


Figure 5.6: Simulation result

We simulate the results of `prop3` with the first founded error written in the ".trail" file, the result of this simulation is shown in Figure 5.6, where we see the packet forwarded by host B (here internally called prod2) to the switch (here called Protocol1). The flow table in the switch is updated successfully (see in action $25!11, 2$) but soon the packet is dropped (see in action $25?11, 2$). Before C (cons1) receives any other packets, the one sent from host A matches the new flow table (header $= 11$) and thus will be dropped (in action $1?11, 0$) instead of being forwarded to C.

5.5 Related work

The first automata based model for Reo connectors appeared in [13] where constraint automata have been introduced. The authors define a product on constraint automata that implements the Reo composition operator on timed-data streams semantics [7]. Since then, several other operational models followed, such as Büchi automata of records [54], guarded automata [19], and Reo automata [99]. Those models extend constraint automata by allowing some context-dependent reasonings. See [58] for an overview of the main operational and denotational models of Reo. Constraint automata with memory and their composition have been thoroughly studied in [57]. Our work is based on a similar model and semantics, but while we aim for finding a subset of it that can be easily implemented, the work in [57] concentrates on the efficient computation of the main composition operator.

Also, model checking of Reo connectors has been a very active area of research. Vereofy is a dedicated model checker developed explicitly to verify linear time and branching time temporal properties of Reo connectors expressed as constraint automata [12]. Vereofy, however does not allow for explicit data to be handled in the automata and properties. For this reason, in [66] the authors encode Reo connectors as communicating processes, and use mCRL2 to check some behavioral properties. The model checker mCRL2 [22], is based on the Algebra of communicating processes [11] with properties expressed as formulas in the modal μ-calculus with strong and branching bisimulation as equivalences as well as strong and weak trace equivalence. UPPAAL is a model checker [16], for linear time temporal property of networks of timed automata [3]. UPPAAL can perform reachability analysis, as well as simulation and error reports. See [28] for a recent use of UPPAAL to verify behavioral properties of real time Reo connectors. Our work differs from previous works on model checking Reo Connector in the following points. First of all, Spin is a data sensitive model checker. Contrary to Vereofy, we can verify temporal properties on connectors that involve data values and local memory. Second, our use of the Spin model checker is designed specifically to verify LTL properties of Reo connectors, contrary to the strong bisimulation equivalence used in proving equalities in the mCRL2 encoding of Reo channels in [66]. Thus is more in line with Vereofy and the semantic basis of Reo that is trace based, without branching properties. In addition, the encoding of Reo described in [66] does not encompass memory. Finally, our translation into Promela differs from the above works in the use of atomic statements in order to enforce synchrony. Of course extension of our work to real time systems and UPPAAL are imaginable, and can be pursuit in the near future. All in all our work extends the literature by providing another tool chain to compile connectors to Promela and use the Spin model checker.

In [53], the authors check network consistency properties in the model of SDN topol-

ogy by using UPPAAL with the goal to detect an inconsistency or verify a flow against real-time properties. An SDN model extended with synchronization barriers is considered in [1]. Their approach is based on encoding an SDN model into the ABS language, and use the SYCO tool to verify properties about safety policies and network loops. [35] uses Flow-LTL to specify the data flow in an extension of Petri nets. Concurrent updates and packet coherence in the network are then checked with the hardware model checker ABC.

5.6 Conclusion

In this chapter, we presented a full automatic translation from symbolic constraint automata to Promela. Symbolic constraint automata are a characterization of constraint automata with memory that can be used to compactly model almost all connectors of the coordination language Reo. In particular, we restrict ourselves to an executable subset of Reo, assuming predicate and actions to be decidable. On the one hand, symbolic constraint automata cannot characterize relational constraints involving, for example, output ports in predicates. On the other hand, our symbolic constraint automata and their Promela translation easily allow for a generalization to lossy connectors by testing the non-presence of a trigger in an output port before executing a lossy transition, as shown, for example, in the following Promela code for a lossy synchronous connector:

```
1 proctype LossySync(port A; port B){
2 mytype _a; mytype _b;//internal values in Sync
3 int state = 0; //initial state
4 do
5 :: state = 0 && Full(A.data) && Full(B.trig) ->
6     Atomic{take(A, _a); _b =_a; put(B,_b); state = 0};
7 :: state = 0 && Full(A.data) && Empty(B.trig) ->
8     Atomic{take(A, _a); state = 0};
9 od }
```

The translation to Promela is interesting in itself as we had to circumvent the problem of Promela not allowing for checking complex predicates on input ports in a guard of a statement. This is however a crucial feature for Reo connectors, that we have taken care of via delayed input and using control variables for recording transitions already taken (and thus guaranteeing not only liveness, but also fairness in the transition selection process).

As a proof of concept, we run our translation for the verification of an SDN model in Spin. The intuitive and modular Promela code is internally translated to a large transition system with more than 200.000 states and 100.000 transitions. It would be interesting to look for an abstraction mechanism at the level of symbolic constraint automata to help reduce the state explosion. An obvious candidate is the combination

of partial order reduction techniques at the symbolic level of the automaton itself.