An algebra for interaction of cyber-physical components
Lion, B.

Chapter 3

Reo as an algebra of order sensitive components

In this Chapter, we instantiate the algebra of components introduced in Chapter 2 to compositionally design, compile, and analyse order sensitive components. Order sensitive components are components for which only the fact that an observation *happens before* another observation matters, but not the absolute time at which the observation occurs nor the interval of time between two observations. Many real world applications fall in the class of interacting order sensitive components, and dedicated design languages, such as the Reo coordination language [6, 3, 48], focus on formally specifying the interaction occurring among such components.

Reo protocols coordinate components by means of elementary and composite connectors. The behavior of a whole system based on these components is mainly defined by its Reo protocol, which makes the reliability of the coordination protocol a central part of the verification. We list in Table 3.1 few properties of interest to study the temporal behavior of connectors. We note a port p in italic, together with some key words such as *silent* and *always*. Reo semantics is at the level of transactional components: ports fire within transactions. An executable of a Reo specification (e.g., an implementation) would however use sequential (yet concurrent) primitives, and falls into the category of linear components. This Chapter discusses a possible implementation of the linearization introduced in Section 2.3. The property of *synchrony* captures that two ports a and b fire together. In a sequential setting, we allow, as shown in Section 2.3, silent steps between firing of a and firing of b: we call this relation a *then*

b.

Table 3.1: Properties on firing of ports and their meaning

Properties	Relations
a fires	exchange of data at *a*
silent	no firings
a before *b*	*a* fires then eventually *b* fires
a then *b*	*a* fires then *silent* until *b* fires
synchrony (sync)	*always a* fires iff *b* fires
asynchrony (async)	*always a* before *b* or *b* before *a*

We show that the property of synchrony on transactional components is then reflected by the property *a* then *b* on a sequential setting. To experiment and verify temporal properties, we give a translation from Reo to Promela. Promela, the specification language used for the LTL model checker SPIN, is sequential and relies on message passing through channels. Some tools currently exist to translate a Reo circuit into a language used by a model checker [55]. However, no general mechanism is described to deal with the translation of synchronous properties on Reo circuits to asynchronous properties on the target language. This section extends a prior work[63].

We express, in Section 3.1, the formal semantics of Reo as an algebra of order sensitive components. We show that this algebra can express Reo connectors using the primitive notion of a port component, and a set of interaction signatures. We use the results of Chapter 2 to show algebraic properties of the Reo composition operator, and discuss some connector equivalences. Section 3.2 introduces a logical specification of Reo connector as guarded commands. The semantics of such connector is linked to the Reo semantics by considering sequences of observations that satisfy the connector's constraint. Section 3.3 is motivated by the extension [60] of our Reo compiler to generate verifiable specification using the SPIN model checker [41]. We implement a translation from a logical specification of Reo connectors to Promela, an executable language, and show that properties of synchrony are preserved during translation. We use the SPIN model checker to investigate some temporal properties of basic Reo connectors, and give a general framework and domain specific language for verifying temporal properties of Reo protocols.

3.1   Reo

In Chapter 2, we introduced an algebra of components and their parametrized products. We also presented properties of components, such as being order sensitive. The order sensitivity property manifests the fact that a component behavior is independent to the concrete time value of its observations, and only the order of observations matters. For instance, the time at which a packet is sent on a network is usually irrelevant, but the order at which each packet arrives at its destination is important.

We show in this section how to define Reo connectors (and Reo primitives) using our algebra of components. More precisely, we give a description of Reo primitives (like a merger, a replicator, a fifo1, etc.) as product of port components under some interaction signatures. This way, we open new reasoning about circuit equivalences based on the underlying algebraic laws of such operators (e.g., associativity, commutativity, and distributivity).

A *component* in Reo has an *interface* consisting of a set of ports, and a *behavior* specifying the sequence of data flowing at each port. Components interact with other components via shared ports by *agreeing* on the data flowing at that port.

Current work on Reo focuses particularly on a specific class called *connector*. A connector specifies the exchange or transformation of data only, but not its creation nor its deletion. Therefore, a connector has open ended input and outputs ports, to which consumer and producers are eventually placed. Typically, for a connector to have observable behaviors, components plugged at the input ports must *feed* data into that port, while components at the output ports must *consume* data through that port. As an example, the *sync* connector in Reo has an input and output open ended points, and synchronously forwards input data to its output. If connected to two components, one at each endpoint, a sync connector essentially models a simultaneous send and receive operation between those two components. Connectors may be connected to each others, forming of more complex input/output relations [6].

Formally, each port denotes a set of Timed-Data Streams (TDS), which are infinite sequences of a datum paired with a time stamp. Intuitively, a Timed-Data Stream tells what data flows at what time through a port, and faithfully transcribes the observable behavior of that port. A component denotes a set of tuples of TDSs, where each port from the component's interface is uniquely represented by a TDS in each tuple. A component therefore specifies which of the TDS tuples are accepted or rejected. We study properties of Reo connectors expressed as a fragment of the component framework of Chapter 2.

The first model for Reo was introduced in [3], and made use of timed-data streams where each port labels its firing with a time stamp. The representation of time as a real value is motivated by the necessity to allow arbitrarily many finite interleavings between two observations. Most of Reo connectors, however, are order sensitive, which implies that the precise value of the time stamp of an observation does not matter, only the order of observations matters.

Port as a component In Reo, the most primitive form of computations take place at a port. A port denotes events that occur over time at a unique location. Often, a port does not restrict which sequence of events may occur, and captures all possible such sequences.

We therefore model a port as a unary component from the model introduced in Chapter 2. We use $P_a(D) = (E_a(D), L_a)$ to denote the port a with domain D where $E_a(D) = \{(a, d) \mid d \in D\}$ that contains all events for port a, and its behavior $L_a \subseteq TES(E_a(D))$ contains all TESs with singleton or empty observations, i.e., such that $\sigma \in L_a$ implies that for all $i \in \mathbb{N}$, $|\sigma(i)| \leq 1$. When the context is clear, we drop the domain of a port and simply write $P_a = (E_a, L_a)$. Note that a port is an order sensitive component, as only the order between occurrences of events matter, but not the exact time nor the time interval between two observations.

Example 38 (Alternating port)**.** *We define* $P_m = (E_m, L_m)$ *where* $E_m = \{(m, 0), (m, 1)\}$ *and* $\sigma \in L_m \subseteq TES(E_m)$ *if and only if, for all* $i \in \mathbb{N}$, $\sigma(2i) = (\{(m, 0)\}, t_i)$ *and* $\sigma(2i + 1) = (\{(m, 1)\}, t_{i+1})$, *which consists of a stream of alternating bits at port* m.

Interaction signature Following Example 11, we define three main interaction signatures that are used to form binary and n-ary Reo components.

The *synchronous* signature $\Sigma^{sync}_{(a,b)} = ([\kappa^{sync}_{(a,b)}], [\cup])$ enforces events at port a to occur at the same time as the same event at port b, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa^{sync}_{(a,b)}(E_a, E_b)$ if and only if

$$t_1 < t_2 \implies O_1 \cap E_b = \emptyset \wedge$$
$$t_2 < t_1 \implies O_2 \cap E_a = \emptyset \wedge$$
$$t_2 = t_1 \implies \forall d.((a, d) \in O_1 \iff (b, d) \in O_2)$$

Note that κ^{sync} from Example 11, when restricted to observations occurring at ports, is the symmetric and reflexive relation that synchronizes the occurrences of events at every shared port, i.e., κ^{sync} is the union of all $\kappa^{sync}_{(x,x)}$ with x a port name.

The *asynchronous* signature $\Sigma_{(a,b)}^{async} = ([\kappa_{(a,b)}^{async}], [\cup])$ prevents events from port a and b to occur at the same time, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa_{(a,b)}^{async}(E_a, E_b)$ if and only if

$$\forall d_1, d_2.((a, d_1) \in O_1 \wedge (b, d_2) \in O_2) \implies t_1 \neq t_2$$

The *relational* signature $\Sigma_{(a,b,\sqcap)}^{rel} = ([\kappa_{(a,b,\sqcap)}^{rel}], [\cup])$, for $\sqcap \subseteq E_a \times E_b$ a relation, relates an event (a, d_1) from port a to a simultaneous event (b, d_2) at port b such that $((a, d_1), (b, d_2)) \in \sqcap$, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa_{(a,b,f)}^{rel}(E_a, E_b)$ if and only if

$$
\begin{aligned}
t_1 < t_2 &\implies O_1 \cap dom(\sqcap) = \emptyset \wedge \\
t_2 < t_1 &\implies O_2 \cap codom(\sqcap) = \emptyset \wedge \\
t_2 = t_1 &\implies (\forall d_1.(a, d_1) \in O_1 \cap dom(\sqcap) \\
&\implies (\forall d_2.(b, d_2) \in O_2 \cap codom(\sqcap) \\
&\implies ((a, d_1), (b, d_2)) \in \sqcap)
\end{aligned}
$$

where $dom(\sqcap) \subseteq E_a$ and $codom(\sqcap) \subseteq E_b$ are the sets of events in the domain and co-domain of the relation $\sqcap$. Every event that is not in the related by $\sqcap$ can therefore freely occur at anytime. Note that if $\sqcap$ is the identity relation on the data element, i.e., $(a, d_1), (b, d_2) \in \sqcap$ implies $d_1 = d_2$, then $\Sigma_{(a,b,\sqcap)}^{rel}$ is equal to the signature $\Sigma_{(a,b)}^{sync}$.

Example 39 (Binary component)**.** *Let a be a port, and m an alternating port. Let f_0 be the functional relation such that $f_0(a, v) = (m, 0)$ for all $(a, v) \in E_a$, the product $P_a \times_{\Sigma_{(a,m,f_0)}^{rel}} P_m$ represents the component that synchronizes all values at port a with the value 0 at port m. Reciprocally, fixing the functional relation f_1 to be such that $f_1(b, v) = (m, 1)$ for all $(b, v) \in E_b$, the product $P_b \times_{\Sigma_{(b,m,f_1)}^{rel}} P_m$ represents the component that synchronizes all values at port b with the value 1 at port m.*

The *delay* signature $\Sigma_{(a,b,\sqcap)}^{delay} = ([\kappa_{(a,b,\sqcap)}^{delay}], [\cup])$, for port a, b, and a relation $\sqcap \subseteq E_a \times E_b$, restricts every occurrence of data at port a to be related to a later observable at port b, i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa_{(a,b,\sqcap)}^{delay}$ if and only if

$$t_1 < t_2 \implies (\forall d_1.(a, d_1) \in O_1 \implies (\forall d_2.(b, d_2) \in O_2 \implies ((a, d_1), (b, d_2)) \in \sqcap))$$

When $\sqcap$ is the identity relation on the data element, i.e., $((a, d_1), (b, d_2)) \in \sqcap$ implies $d_1 = d_2$, we simplify the notation to write $\Sigma_{(a,b)}^{delay}$.

Reo syntax Reo has a graphical syntax that visualizes composition of components. Figure 3.1 shows three kinds of components: *synchronous channels*, *asynchronous channels* and *nodes*. In this section, we describe the semantics of these kinds only intuitively. Typically, we call a binary component a channel and certain kinds of other components nodes.

First, we consider two synchronous channels. The *sync* channel in Figure 3.1 represents a synchronous transfer of data from port a to port b. The *syncdrain* in Figure 3.1 models a synchronous firing of port a and b without necessarily equating data at those ports. There are also asynchronous channels. A *fifo1* channel (depicted in Figure 3.1) has an internal buffer with the capacity to hold one data item. This buffer is initially empty. When its buffer is empty, a *fifo1* channel accepts a data item through its input port a, places it in its buffer, which then becomes full. When its buffer is full, a *fifo1* channel no longer accepts any intput. When the buffer of a *fifo1* channel is full, the channel delivers the content of its buffer to a get operation performed by the environment on its output port b, and its buffer becomes empty. A get on the output port of a *fifo1* channel with an empty buffer blocks until after its buffer becomes full. Thus, the get and put operations on the ports of a *fifo1* channel succeed only asynchronously: never together. The *asyncdrain* channel in Figure 3.1 never allows a pair of put operations on its boundary ports to succeed synchronously. Finally, Figure 3.1 has two ternary components: a *merger* and a replicator. A merger synchronizes data transfer through at most one of its input boundary ports with data transfer through its output port. If data is available at both input ports, a *merger* non-deterministically chooses one to synchronize with its output port. A *replicator* forwards the data from its input port to both of its output ports. All ports must be ready for the replicator to proceed. A *filter* forwards the data from its input to its output if the predicate ϕ labeling the filter holds on the input data. In the case where the predicate ϕ does not hold on the data at its input, the data is lost. Oppositely, the blocking *filter*, written *bfilter*, blocks when the data at its input violates the constraint ϕ. See Section 4.3 for the use of Reo primitives to construct connectors.

Reo semantics Typically, the composition operator is fixed in Reo to be $\times^{sync}$ that joins behaviors of components on shared port names. We use the notation $\bowtie$ to denote such operation. Reo fixes the semantics of a node [6], whereas all channels are user defined. There is, however, a commonly useful set of channels (see Figure 3.1) that we use in this section. We define algebraically some common channels out of the port component introduced earlier and few interaction signatures that we listed.
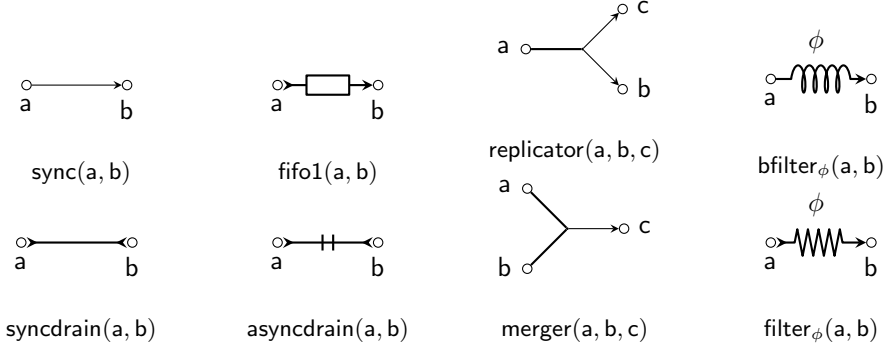
Figure 3.1: Graphical syntax for some primitives.

More generally, we define Reo components as a fragment of our component algebra:

$$C := C \times_\Sigma C \mid P_x$$

where $\Sigma \in \{\Sigma^{sync}, \Sigma^{sync}_{(a,b)}, \Sigma^{async}_{(a,b)}, \Sigma^{delay}_{(a,b,\sqcap)}, \Sigma^{rel}_{(a,b,\sqcap)}\}$ for some port names a, b, x, for some relation on events $\sqcap$.

We define the following Reo channels and nodes:

$$sync(a,b) = P_a \times_{\Sigma^{sync}_{(a,b)}} P_b$$

$$syncdrain(a,b) = P_a \times_{\Sigma^{rel}_{(a,b,\sqcap)}} P_b \; with \; \sqcap = (E_a \times E_b)$$

$$filter(a,b,f) = P_a \times_{\Sigma^{rel}_{(a,b,\sqcap_f)}} P_b \; with \; \sqcap_f = \{((a,d),(b,f(d)) \mid d \in dom(f)\}$$

$$fifo1(a,b,M) = (P_a \times_{\Sigma^{rel}_{(a,m,f_0)}} P_m) \times_{\Sigma^{delay}_{(a,b)} \cup \Sigma^{sync}_{(m,m)}} (P_b \times_{\Sigma^{rel}_{(b,m,f_1)}} P_m)$$

$$merger(a,b,c) = (P_a \times_{\Sigma^{excl}_{(a,b)}} P_b) \times_{\Sigma^{sync}_{(a,c)} \cup \Sigma^{sync}_{(b,c)}} P_c$$

with $\Sigma^{delay}_{(a,b)} \cup \Sigma^{sync}_{(m,m)} = ([\kappa^{delay}_{(a,b)}] \cup [\kappa^{sync}_{(m,m)}], [\cup])$ and $\Sigma^{sync}_{(a,c)} \cup \Sigma^{sync}_{(b,c)} = ([\kappa^{sync}_{(a,c)}] \cup [\kappa^{sync}_{(b,c)}], [\cup])$ and $\sqcap_f = \{(\{(a,d_1)\}, \{(b,d_2)\}) \mid d_2 = f(d_1) \; or \; \}$. The $sync(a,b)$ component is such that the data observed at port a and b are equal and synchronous, i.e., occurs at the same time. The $syncdrain(a,b)$ component ensures that both the data of a and b are observed at the same time, but does not restric their data to be equal. The component $fifo1(a,b,M)$ synchronizes the observation of a data at a with the change of the memory state M, and then outputs the same data at b. As defined here, the $fifo1(a,b,M)$ component is infinitely productive, i.e., always eventually has an input at a and an output at b. The component $filter(a,b,f)$ synchronizes events from a with

event from b related by the function f. Any unrelated event at a or b can freely happen. Note that, in the case that f is the identity, we recover the Reo filter behavior where the condition $d \in dom(f)$ denotes some predicate ϕ. The blocking filter behavior can be encoded by composition of a non-blocking filter and other Reo primitives. The $merger(a, b, c)$ component either synchronizes a with c or b with c but never all ports together.

A strength of Reo is its compositional nature: protocols are built out of primitives. We use the join operation defined in Example 7 (see Theorem 1) for the proof of associativity and commutativity of $\bowtie$) to define two Reo connectors:

$$alternator(a, b, c) = sync(a, c_1) \bowtie fifo1(x, c_2) \bowtie syncdrain(a, b) \bowtie$$
$$sync(b, x) \bowtie merger(c_1, c_2, c)$$
$$fifo2(a, b) = fifo1(a, x, M_1) \bowtie fifo1(x, b, M_2)$$

3.2 Logical specification of connector components

Components defined in Section 3.1 are order sensitive components: the order of the sets of events in their behavior matters, but not the exact time of occurrence of that set. Yet, no finite specification of component behavior is given.

In this section, we give a logical specification of components as a predicate in guarded command form. Intuitively, the guards of the perdicates are conditions for the commands to be executed. Consecutive satisfactions of the guarded command predicate form the behavior of the corresponding component. The resulting logical specification can be translated to an output language for execution or verification.

This work relates to existing work on verification of temporal on Reo components. In addition, this section presents a powerful intermediate representation in Section 3.2.1, that minimizes the size of the conjunction of guarded command predicates. Moreover, we introduce a structure for a port at runtime that facilitates that specification of properties in Table 3.1 as temporal properties.

Language of constraints We formally characterize the behavior of a component as a predicate relating the data flow through its ports. Note that we first study the behavior of a component in its ideal environment, i.e. all input and output sequences are possible. The resulting behavior that a component describes is a set of tuples of data streams representing the synchronous flowing of data through the ports of its interface.

Data elements that flow through ports and get stored in memory belong to a domain that we call D. In this work, we use a unique domain for all port and memory, since we do not use any algebraic operations on data. Note that Reo allows more structured data elements exchanged through ports and memories. As we will later see in the characterization of components' behavior, the need of talking about the case where no data is observed at a port or memory is of importance. The item $*$ is added to the data domain D, and D_* denotes the resulting data domain.

Ports and memories appear as variables in the logical characterization. Port and memory variables take values in the domain D_*. The set of port variables is denoted as P, and M denotes the set of memory variables. While ports do not have any memory (as mentioned in the previous paragraph), memories always store the previous data item. For each memory variable $m \in M$, there is a memory variable $m' \in M'$. Their interpretation becomes clear in the next paragraph.

User defined components are characterized by a user supplied predicate or function among the ports of its interface. The sets Q and F respectively denote the set of n-ary predicate symbols and n-ary function symbols.

A *term* is either a variable $p \in P$, $m \in M$, or $m' \in M'$, an n-ary function application $f(t_1, ..., t_n)$ where $f \in F$ is an n-ary function symbol, or a constant $d \in C$.

A *formula* is built inductively by:

$$\phi \ ::= t_1 = t_2 \mid B(t_1, ..., t_n) \mid \phi_1 \wedge \phi_2 \mid \neg\phi$$

Where $B \in Q$ is a predicate symbol. The set of formula expressions is denoted by $\mathcal{F}$. We use the shorthand notation $t_1 \neq t_2$ for $\neg(t_1 = t_2)$, $\perp$ for $t_1 \neq t_1$, and we get $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$ as well as $\phi \implies \psi = \neg\phi \vee \psi$.

We allow existential quantifier at the outermost left position of a formula. Quantifiers range over port variables only. A port occurring in ϕ but not existentially quantified is called a *free* port variable. We call *atomic* a formula that is either an equality, an inequality, or a predicate. We call V_ϕ the set of free variables occurring in ϕ, and similarly $P_\phi \subseteq V_\phi$ and $M_\phi \subseteq V_\phi$ for the set of free port and memory variables. When ϕ is clear from the context, we drop the subscript notation.

Example 40. *A Reo component, as introduced earlier, constrains the flow of data through its boundary ports. We call constraint of the component the logical formula used to relate the data flowing through the ports. We usually write* comp(a, b) *for the formula of the component comp having as free port variables a and b. We give below two examples for components sync and fifo1.*

Given a port $p \in P$, the proposition of whether or not p fires is encoded as an equality between p and elements of D_*. We say that p fires if $p \neq *$. On the other hand, p does not fire if $p = *$. Taking $a, b \in P$, we can now express that a and b fire synchronously with the same datum, as the formula $a = b$.

$$\mathsf{sync}(a, b) = a = b \tag{3.1}$$

$$\mathsf{fifo1}(a, b) = (m' = a \wedge a \neq * \wedge b = * \wedge m = b) \vee$$
$$(m' = a \wedge a = * \wedge b \neq * \wedge m = b) \vee$$
$$(m' = m \wedge a = * \wedge b = *) \tag{3.2}$$

The constraint for a fifo1 channel has three clauses. The first one corresponds to filling the buffer with the data item observed at port a; the second one empties the buffer through port b; and the last one corresponds to the case where no port fires, in which case the value in the buffer must remain unchanged.

As hinted previously, protocols can be built by composing primitives. In the case of a composite component, the resulting constraint is defined as the conjunction of the constraints of the underlying components.

The logic is agnostic regarding the direction of data flow and merely represents the constraint on the data observed at each port.

Solution of constraints Every constant element of D_* get mapped to an element of the homonym domain D_*. Let γ be the map for every n-ary function symbol $f \in F$ to an element of $D_*^n \to D$. Let $\mathcal{I}$ be the map for every n-ary predicate symbol $B \in Q$ to an element of $D_*^n \to 2$. Let $\Gamma : V \to D_*$ be the interpretation function for variable symbols where $V = P \cup M \cup M'$. The interpretation of a term t, noted $[\![t]\!]_{(\Gamma, \gamma)}$, is the standard inductive interpretation providing the signature (Γ, γ)

A *solution* to a formula ϕ is an assignment Γ such that Γ *satisfies* ϕ, written as $\Gamma \models \phi$, defined inductively on ϕ as:

$\Gamma \models \top$ always

$\Gamma \models t_1 = t_2$ iff $[\![t_1]\!]_{(\Gamma, \gamma)} = [\![t_2]\!]_{(\Gamma, \gamma)}$

$\Gamma \models \phi_1 \wedge \phi_2$ iff $\Gamma \models \phi_1$ and $\Gamma \models \phi_2$

$\Gamma \models \neg\phi$ iff $\Gamma \not\models \phi$

$\Gamma \models \exists p\phi$ iff there exists $d \in D_*$ such that $\Gamma \models \phi[d/p]$

$\Gamma \models B(t_1, ..., t_n)$ iff $([\![t_1]\!]_{(\Gamma, \gamma)}, ..., [\![t_n]\!]_{(\Gamma, \gamma)}) \in \mathcal{I}(B)$

We extend the domain definition of an n-ary function from D^n to D^n_* by defining
$f(t_1,...,t_n) = * \iff t_1 = * \lor ... \lor t_n = *$.

Example 41. *Following the Example 40, the set of solutions for a sync and a fifo1 channel corresponds to the assignments for all free variables in the formula $\mathsf{sync}(a,b)$ and $\mathsf{fifo1}(a,b)$ that satisfy the formula. We use the data domain $D = \{0,1,*\}$, and for clarity, we write $(0,1)$ as the assignment that maps $a \mapsto 0$ and $b \mapsto 1$. In this context, the assignments $(0,0)$, $(1,1)$, and $(*,*)$ are the only assignments that satisfy the constraint $\mathsf{sync}(a,b)$. We write $(0,1,0,1)$ the assignment that maps a to the first element, b to the second element, m to the third, and m' to the last element. Thus, considering the constraint $\mathsf{fifo1}(a,b)$, the assignment $(0,*,*,0)$ and $(1,*,*,1)$ both satisfy the constraint. In the case of the latter, the next value of the memory should now be equal to 1, and only assignment mapping the memory to 1 would be allowed. We explain in the next paragraph the behavior of a component as all the infinite sequences of assignments that satisfy its internal constraint.*

3.2.1   Connector as guarded commands: an intermediate form

A component, if it has some open ports, is not in isolation but must co-evolve with its environment. The solution of its internal constraints must conform with the data provided or requested by the environment.

Guarded commands   We propose a method based on guards and command to implement and simulate transactional behavior. The guard is a predicate on the state of the ports and the memory. If the guard is true, the update describes a constraint that both the component and its environment must satisfy. In an implementation language, updates are themselves sequential, which can introduce some interleaving in a concurrent setting. We later show, by making use of temporal properties, that the translation to an sequential (yet concurrent) implementation preserves the atomicity property.

We first refine the language of constraint and impose some requirement on ϕ to be a guarded command. We show some properties if ϕ satisfied those requirements.

We denote by v^{in} an input variable and v^{out} an output variable. An input term t^{in} refers to either a data element $d \in D_*$, an n-ary function $f(t^{in}_1,...,t^{in}_n)$ whose arguments are input terms $t^{in}_1,...,t^{in}_n$.

A guards g is a conjunction of literals l defined as follows:

$$l ::= \ B(t_1^{in}, ..., t_n^{in}) \mid t_1^{in} \sim t_2^{in} \mid v^{out} \sim d$$

where $\sim \in \{=, \neq\}$.

A commands c is a conjunction of equalities of the kind $v^{out} = t^{in}$. We say that a formula ϕ is in *guarded command form* if

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$$

where g_i are formulas in the language of guards and c_i are formulas in the language of commands.

Given ϕ in guarded command form, we call an implication of the type $g \implies c$ in ϕ, a guarded command of ϕ. We call the number of guarded commands in such a formula, the size of that formula. We denote the set of guards by $\mathcal{G}$, and the set of commands by $\mathcal{C}$. Note that guarded commands are quantifier free formulas.

We say that a quantifier free formula $\phi = \phi_1 \vee ... \vee \phi_n$ in disjunctive normal form and expressed in the language of constraints is *deterministic* if ϕ can be written with the grammar of guarded commands such that $\phi = g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n$, where $\wedge$ has precedence over $\vee$, g_i are guards, c_i are commands, and $g_i \wedge g_j \equiv \bot$ for all i, j where $i \neq j$.

We assume that if a v^{out} is involved in an equality, then the other term is either $*$, or an input term t^{in}. In other words, if an equality $v_1^{out} = v_2^{out}$ appears in the constraint ϕ, there must exist an input term t^{in} such that $v_1^{out} = t^{in}$ and $v_2^{out} = t^{in}$ in order for ϕ to be written in the guarded command form. Moreover,

Proposition 1. *A deterministic formula $\phi = \bigvee_i (g_i \wedge c_i)$ can be written as a guarded command where:*

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i)$$

where i ranges over the number of disjuncts of ϕ.

Proof. By induction on the structure of ϕ. We assume ϕ is in disjunctive normal form, and negation is pushed to the literals. We can write $\phi = \phi_1 \vee ... \vee \phi_n$ where ϕ_i are conjunctions of equalities, inequalities, or predicates.

<u>Step 1</u> (identification): We syntactically partition each ϕ_i with its corresponding guard g_i and command c_i such that $\phi_i = g_i \wedge c_i$. We get the resulting formula $\phi = g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n$. We syntactically substitute $g_i \wedge c_i$ by $(g_i \implies c_i) \wedge g_i$. We then get

$\phi = (g_1 \implies c_1) \wedge g_1 \vee ... \vee (g_n \implies c_n) \wedge g_n$.

<u>Step 2</u> (factorization): Because the formula ϕ is deterministic, for all $i \neq j$ we have $g_i \wedge g_j \equiv \bot$, which equivalently gives $g_i \wedge \neg g_j \equiv g_i$. We can then rewrite $(g_1 \implies c_1) \wedge g_1$ as $(g_1 \implies c_1) \wedge (g_2 \implies c_2) \wedge g_1$, since

$$
\begin{aligned}
(g_1 \implies c_1) \wedge (g_2 \implies c_2) \wedge g_1 &= g_1 \wedge c_1 \wedge (\neg g_2 \vee c_2) \\
&= g_1 \wedge c_1 \wedge \neg g_2 \vee g_1 \wedge c_1 \wedge c_2 \\
&\equiv g_1 \wedge c_1 \vee g_1 \wedge c_1 \wedge c_2 \\
&\equiv g_1 \wedge c_1 \\
&\equiv (g_1 \implies c_1) \wedge g_1
\end{aligned}
$$

By induction on the number of implications, we conclude that $(g_j \implies c_j) \wedge g_j$ is equivalent to $\bigwedge_i (g_i \implies c_i) \wedge g_j$ for all j, and thus:

$$
\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i)
$$

$\square$

Given two guarded commands $\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$ and $\psi = \bigwedge_i (g_i' \implies c_i') \wedge (\bigvee_j g_j')$, we write the composition $\phi \wedge \psi$ as the formula:

$$
\phi \wedge \psi = \bigwedge_i (g_i \implies c_i) \bigwedge_i (g_i' \implies c_i') \wedge (\bigvee_{i,j} g_i' \wedge g_j)
$$

Proposition 2. *Given two deterministic formulas ϕ and ψ, their product $\phi \wedge \psi$ is also deterministic.*

Proof. Since ϕ and ψ are deterministic, we can write $\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$ and $\psi = \bigwedge_i (g_i' \implies c_i') \wedge (\bigvee_j g_j')$ where for $i \neq j$, $g_i \wedge g_j \equiv \bot$ and $g_i' \wedge g_j' \equiv \bot$. The product $\phi \wedge \psi$ can be seen as:

$$
\begin{aligned}
\phi \wedge \psi &= (g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n) \wedge (g_1' \wedge c_1' \vee ... \vee g_n' \wedge c_n') \\
&= (g_1 \wedge g_1' \wedge c_1 \wedge c_1' \vee ... \vee g_n \wedge g_n' \wedge c_n \wedge c_n')
\end{aligned}
$$

The new formula $\phi \wedge \psi$ inherits from $\phi \wedge \psi$ that for any i, j, k and l such that $i \neq j \vee k \neq l$, we have $g_i \wedge g_k' \wedge g_j \wedge g_l' \equiv \bot$. Therefore, $\phi \wedge \psi$ is also deterministic and

can be written as a guarded command:

$$\phi \wedge \psi = \bigwedge_{i,j}((g_i \wedge g'_j) \implies (c_i \wedge c'_j)) \wedge \bigvee_{i,j}(g_i \wedge g'_j)$$

$\square$

As the construction in the proof of Proposition 2 shows, writing the composition of two deterministic formulas as a deterministic formula may increase the size of the resulting formula.

We present some optimizations that can be applied to formulas before forming their product. We use the formula $\phi = \bigwedge_i(g_i \implies c_i) \wedge (\bigvee_i g_i)$ and $\psi = \bigwedge_j(g'_j \implies c'_j) \wedge (\bigvee_j g'_j)$ to illustrate these transformations, where i and j range over a finite set of natural numbers.

In general, the disjunction of guards $\bigvee_i g_i$ constituting the last part of the guarded command can imply a relation on the literals of the guards, and simplify the guards themselves. We consider the example of the sync channel:

$$\begin{aligned}
\phi_{sync} =& (a = * \wedge b = * \wedge a = b) \vee (a \neq * \wedge b \neq * \wedge a = b) \\
=& ((a \neq * \wedge b \neq *) \implies a = b) \wedge \\
& ((a = * \wedge b \neq *) \implies a = b) \wedge \\
& ((a = * \wedge b = *) \vee (a \neq * \wedge b \neq *)) \\
=& (a \neq * \implies a = b) \wedge (a \neq * \iff b \neq *)
\end{aligned}$$

In this example, the guarded command form of the formula for *sync* induces a relation on a fires and b fires that simplifies the formula. It also means that in subsequent compositions, we can consider literals $a \neq *$ and $b \neq *$ as interchangeable. We consider as future work the exploitation of such relations that emerge from the disjunction of guards.

Given the product $\phi \wedge \psi$, if we find g_k and g'_l such that $g_k \iff g'_l$, we can equivalently consider $\phi \wedge \psi$ as the following formula:

$$\begin{aligned}
\phi \wedge \psi =& ((g_k \wedge g'_l) \implies (c_k \wedge c'_l)) \wedge \\
& \bigwedge_{i \neq k}(g_i \implies c_i) \bigwedge_{i \neq l}(g'_i \implies c'_i) \wedge (\bigvee_{i,j} g_i \wedge g'_j)
\end{aligned}$$

In other words, if ϕ is of size N and ψ of size M, we decrease the size of $\phi \wedge \psi$ by 1 by identifying two guards, i.e., the size of $\phi \wedge \psi$ is $M + N - 1$. If we compare the resulting size of the disjunctive normal form, since $g_k \wedge g_l'$ is exclusive of all other guards g_i and g_j' for all $i \neq k$ and $j \neq l$, identifying two guards remove $M + N$ clauses from $M \times N$.

Example 42. *We now consider an example of guarded commands for the fifo1 primitive. The fifo1 primitive has a permanent constraint written in Fig. 3.2, with the io designation of $io(a) = 1$ and $io(b) = 0$, i.e., a is an input port and b is an output port. The formula of a fifo1 is deterministic, and with the result of Proposition 1, we can write its permanent constraint as a guarded command:*

$$\phi_{fifo1} = \ (g_1 \implies c_1) \wedge (g_2 \implies c_1) \wedge (g_3 \implies c_2) \wedge$$
$$(g_1 \vee g_2 \vee g_3)$$

where we have the following guards $g_1 := (a \neq * \wedge b = * \wedge m = *)$, $g_2 := (a = * \wedge b \neq * \wedge m \neq *)$, and $g_3 := (a = * \wedge b = *)$; and the following commands $c_1 := (m' = a \wedge m = b)$, and $c_2 := m' = m$.

The formula for a fifo1 channel has three different guards. The first guard g_1 checks whether its source port a is active, in which case the command c_1 fills the buffer with the data observed at port a; the second guard g_2 checks whether the channel's sink port b is active, in which case its command c_1 empties the buffer through port b. The last guard g_3 checks if the two ports a and b are inactive, and if true, triggers the command c_2 where memories are copied over.

Proposition 3. *Given ϕ a deterministic formula written in guarded command form, Γ is a solution of ϕ if and only if there exists a unique guarded command $g \implies c$ of ϕ such that:*

$$\Gamma \models g \wedge c$$

Proof. We assume $\phi = \bigwedge_i (g_i \implies c_i)$ of size $n \in \mathbb{N}$ where g_i are guards and c_i are commands for all $1 \leq i \leq n$.

$$\Gamma \models \phi \text{ iff } \Gamma \models \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_i g_i)$$
$$\text{iff } \forall \ 1 \leq i \leq n, \ \Gamma \models g_i \implies c_i \text{ and } \Gamma \models \bigvee_{j \leq n} g_j$$

Because guards are exclusive to each others (ϕ is a deterministic formula), if there

exists $1 \leq i \leq n$ such that $\Gamma \models g_i$, then $\Gamma \not\models g_j$ for all $j \neq i$. Therefore, there exists a unique $1 \leq j \leq n$ such that $\Gamma \models g_j$.

$$\Gamma \models \phi \text{ iff } \text{ for all } 1 \leq i \leq n, \ \Gamma \models g_i \implies c_i \text{ and}$$
$$\text{there exists a unique } 1 \leq j \leq n, \ \Gamma \models g_j$$

Since there exists a unique guard that Γ can satisfy, for all $1 \leq i \leq n$ such that $i \neq j$, $\Gamma \models g_i \implies c_i$ since $\Gamma \models \neg g_i$. Thus:

$$\Gamma \models \phi \text{ iff } \text{ there exists a unique } 1 \leq j \leq n, \ \Gamma \models g_j$$
$$\text{and } \Gamma \models g_j \implies c_j$$

Which is equivalent to:

$$\Gamma \models \phi \text{ iff } \text{ there exists a unique } 1 \leq j \leq n, \ \Gamma \models g_j$$
$$\text{and } \Gamma \models c_j$$

$\square$

3.2.2 Behavior of connectors

Operational semantics. We present in this section the operational semantics of constraint ϕ specifying a protocol as a labeled transition system, where we consider memory assignment as labels for *states*, and port assignments as labels for *transitions* between states. We call Γ_{V_ϕ} the set of assignment functions $\Gamma : P \cup M \cup M' \to D_*$ that satisfy the constraint ϕ. Given an assignment $\Gamma \in \Gamma_{V_\phi}$, we denote by Γ_P, Γ_M, and $\Gamma_{M'}$ the restrictions of Γ to respectively the port variables, un-primed memory variables, and primed memory variables assignment.

The operational semantics of a connector characterized by an internal constraint ϕ, where $V_\phi = P \cup M \cup M'$, is defined in terms of a labeled transition system $(S, L, s_0, \Gamma_{V_\phi}, \to)$, where:

- S the set of states

- s_0 is the initial state

- $L : S \to (M \to D_*)$ is a labeling function

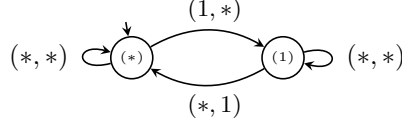- $\Gamma_{V_\phi} = \{\Gamma \mid \Gamma \models \phi\}$ is the set of solutions of ϕ

Figure 3.2: LTS of a *fifo1* channel with domain $D = \{*, 1\}$

- $\to \subseteq S \times (P \to D_*) \times S$ the transition relation s.t.: $(s_0, \Gamma_P, s_1) \in \to$ iff there exists $\Delta \in \Gamma_{V_\phi}$ such that $\Delta_P = \Gamma_P$, $\Delta_M = L(s_0)$, and $\Delta_{M'} = L(s_1)$.

According to this definition, each state in $s_i \in S$ represents a data assignment for memories of a component (free variables in ϕ) at an instant i.

In the case where $M = \emptyset$, which means no memories are defined in the protocol, then the set of states S is composed only of one state s_0, such that $L(s_0) = \emptyset$.

By Proposition 3, the label transition system of a deterministic constraint is deterministic with respect to its label: for any state $s \in S$ and pair of distinct states $s', s'' \in S$, if $(s, \Gamma, s') \in \to$ and $(s, \Gamma', s'') \in \to$, then $\Gamma \neq \Gamma'$. Indeed, the two solutions Γ and Γ' must satisfy two different guarded commands, and since ϕ is deterministic, the two guards cannot be satisfied at the same time. Then, Γ and Γ' must differ in at least one port assignment. It makes sense therefore to use such a label transition system to define the operational semantics of a connector specified by a deterministic constraint, as the resulting LTS is deterministic.

Example 43. *This example prolongs the Example 41, and use the data domain $D = \{1, *\}$. States are labeled by memory assignment. A fifo1 has only one memory, therefore a state labeled by the element (d) represents the assignment of value d to the memory. Transitions are labeled by port assignment. A fifo1 has two ports in its interface, therefore we express the assignment a maps to d_a and b maps to d_b as the tuple (d_a, d_b). Initially, the fifo1 start with an empty memory.*

The set Γ_{V_ϕ} for the fifo1 channel corresponds to all solution of the constraint fifo1(a, b), *that could be listed in the same manner as explained in Example 41. The resulting LTS for a fifo1 channel is shown in Figure 3.2.*

Execution path We define an infinite execution path σ of a transition system LTS as a sequence of transitions, i.e. $\sigma = s_0, \Gamma_{P0}, s_1, \Gamma_{P1}, s_2, ..., s_i, \Gamma_{Pi}, s_{i+1}, ...$, where $(s_i, \Gamma_{Pi}, s_{i+1}) \in \to$.

We denote by w_σ the word induced by the path σ consisting of the sequence of the assignments Γ, i.e. $w_\sigma = (\Gamma_i)_{i \in \mathbb{N}}$ where $\Gamma_i(p) = \Gamma_{Pi}(p)$ and $\Gamma_i(m) = L(s_i)(m)$ for

all $p \in P$, and $m \in M$ where L is the labeling function of the LTS. We write $\mathcal{T}$ for the set of infinite words, and $w \in \mathcal{T}$ is accepted if there exists an infinite execution path σ of LTS such that $w = w_\sigma$. Given $w \in \mathcal{T}$, the element $w(i)$ designates the i-th port and memory assignment in w, and $w(i)(v)$ gives the specific assignment for the port variable if $v \in P$ or memory variable if $v \in M$. The n-th derivation of a word w is noted as $w^{(n)}$ and defined such that $w^{(n)}(i) = w(n + i)$ for all $i \in \mathbb{N}$. Based on this definition, we have $w^{(0)}(i) = w(i)$. For example, in the first table (from the left) in Table 3.2, we have: $w^{(0)}(0) = (*, *, *)$, $w^{(0)}(1) = (1, *, *)$, and $w^{(1)}(0) = (1, *, *)$, $w^{(1)}(1) = (*, 1, *)$, where the values of the first, the second, and the third elements in the tuples are associated respectively to the ports and memory, a, m, and b.

Behavior of components   We define the behavior of a component whose specification is given by a formula ϕ as the set of infinite words accepted by the LTS, i.e.:

$$L_\phi = \bigcup_{w \in \mathcal{T}} \{w \mid w \text{ is an accepted word }\}$$

We refer to the behavior of a port as the restriction of the behavior of a component to a single variable. We note $w_{|a}$ the restriction of the word w to the port variable $a \in P$. The restriction $w_{|a}$ thus denotes the stream of values observed at port a, and is such that $w_{|a}(i) = w(i)(a)$ for all $i \in \mathbb{N}$.

Example 44. *The example of an execution of the protocol corresponding to the connector fifo1 is represented by Table 3.2.*

a	m	b	a	m	b	a	m	b
*	*	*	1	*	*	1	*	*
1	*	*	*	1	*	*	1	1
*	1	*	*	1	*	1	*	*
*	1	1	*	1	1	*	1	1
...	...	...	...	...	...	...	...	...

Table 3.2: Three words in the behavior set of a *fifo1* channel with domain $D = \{1, *\}$

We use the same convention as in Section 2.3, namely that a behavior of a order sensitive component is described without time labels. Moreover, using the set notation for an assignment Γ of ports to values, a word labeling an LTS is a sequence of sets of assignments, which denotes the representative of a set of equivalent behaviors under stretching. Thus, the semantics of a constraint ϕ is given as a component (E_ϕ, L_ϕ)

where E_ϕ contains all possible assignments for all free ports and memories occurring in ϕ.

3.3 Verification of temporal properties on connectors

To specify the properties of the executions of Reo protocols, we define in t this section the LTL formulas semantics on Reo connector behaviors.

Let C be a Reo protocol specified with the formula ϕ, such that its operational semantics is specified with LTS. Let σ be an execution path of LTS. We refer by w_σ a word accepted by LTS over σ, and denote $\mathcal{T}$ the set of accepted words.

An LTL formula is expressed using the following syntax:

$$\varphi ::= \ v = d \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{X}\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1\mathsf{U}\varphi_2$$

where v is a port or memory variable in $P \cup M$ and d is an element of the data domain D_*.

We interpret the LTL formulas over the accepted word w_σ and define the satisfaction relation $\models$ such as,

- $w_\sigma \models v = d$ iff $w_\sigma^{(0)}(0)(v) = d$

- $w_\sigma \models \varphi_1 \wedge \varphi_2$ if and only if $w_\sigma \models \varphi_1$ and $w_\sigma \models \varphi_2$

- $w_\sigma \models \neg\varphi$ if and only if $w_\sigma \not\models \varphi$

- $w_\sigma \models \mathsf{X}\phi$ if and only if $w_\sigma^{(1)} \models \phi$

- $w_\sigma \models \Box\varphi$ if and only if $w_\sigma^{(i)} \models \varphi$ for all $i \in \mathbb{N}$

- $w_\sigma \models \Diamond\varphi$ if and only if $w_\sigma^{(i)} \models \varphi$ for some $i \in \mathbb{N}$

- $w_\sigma \models \varphi_1\mathsf{U}\varphi_2$ if and only if there exists $j \in \mathbb{N}$ such that $w_\sigma^{(j)} \models \varphi_2$, and $w_\sigma^{(i)} \models \varphi_1$ for all $0 \le i < j$.

We introduce several properties of interest on Reo circuit. Given p, m, and m' respectively port, memory, and next memory variables, we denote by **p** and **m** the atomic propositions $p \ne *$ and $m' \ne m$ which represent that p is firing and m is changing value. In both cases, we say that p or m fires.

Example 45. *As an example of LTL formula for a sync channel, we have* $\Box(\mathbf{a} \iff \mathbf{b})$ *where a and b are port variables. For a fifo1 channel, we have* $\Box(\mathbf{a} \implies \mathsf{X}(\neg\mathbf{a}\mathsf{U}\mathbf{b}))$ *and also* $\Box(\mathbf{b} \implies \mathsf{X}(\neg\mathbf{b}\mathsf{U}\mathbf{a}))$.

Properties	Temporal formulas
a fires	$\mathbf{a}$
a silent	$\neg\mathbf{a}$
a before b	$\mathbf{a} \implies \Diamond\mathbf{b}$
a then b	$\mathbf{a} \implies \mathsf{X}(\textit{ silent } \mathsf{U} \ \mathbf{b})$
a sync b	$\Box(\mathbf{a} \iff \mathbf{b})$
a async b	$\Box((\mathbf{a} \implies \Diamond\mathbf{b}) \vee (\mathbf{b} \implies \Diamond\mathbf{a}))$

We now look into the implementation and verification of a protocol described by a formula in guarded command form. As Proposition 3 shows each solution is described by a unique guarded command. Therefore a program implementing a protocol checks the set of guards that are satisfied by the state of the environment and the internal state, and nondeterministically satisfies one and only one corresponding command. We develop in the next section the steps leading from a protocol specification to a program written in Promela.

3.3.1 From synchronous protocol to asynchronous implementation

In the previous section, we detailed the requirement and the procedure to write a protocol as a formula in the guarded command form. The implication of having such a form for a protocol makes it possible and easier to translate it into a program. In this section, we define a translation from a formula written as a guarded command to a Promela program. We show the correctness of the translation, by comparing the semantic of a Reo specification, and that of its target program in Promela.

Translation of Reo to Promela Throughout this section, we assume a generic data type denoted as Data for data flowing in the protocol, since data type is specific in the application that employs the protocol. We go through the main constructs in Reo, being ports, components, and connectors.

Ports In Reo, as defined in the previous section, a port is a location where two components synchronize and exchange data. In Promela, we implement a Reo port as a pair of two Promela channels, each with a buffer size of one. We show that our Promela implementation of a port simulates Reo's synchronized message passing between components.

```
typedef port {
        chan data = [1] of {Data};
```

```
      chan trig = [1] of {int}; }
```

Listing 3.1: definition of a Reo port in Promela

As expressed in Listing 3.1, a port has a data channel and a synchronization channel. The data channel is responsible of the data flow between input and output ends of a port. The Promela synch channel ensures synchronous exchange between these two ends.

As described in Listing 3.2, two actions can be performed on such a constructed port: put and take.

```
inline put(q,a) {                    inline take(q,a) {
  int x;                               q.trig!-1; q.data?a }
  q.data!a;
  q.trig?x }
```

Listing 3.2: put and take functions

The action put has two arguments: a port q and a datum a. 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 x is used to empty the trig channel, but its value does not matter.

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

We describe two temporal properties that reflect the synchronous behavior of a port in an asynchronous implementation. We say that a port fires whenever a data is exchanged between the input party and the output party. 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: both buffers are full. We say the a buffer (or a memory) is full whenever it contains a data, and is empty otherwise. We define some macros in Listing 3.3 for firing and silent property of port p, and for full and empty buffer.

Listing 3.3: Macros for firing of ports

```
#define p_fires (
        !(len(p.data) == 0) && !(len(p.trig) == 0) &&
```

```
        X((len(p.data) == 0) || (len(p.trig) == 0)))

#define p_silent          (! p_fires)



#define m_full            (!(len(m)==0))
#define m_empty           ((len(m)==0))
```

Components   We call an external component that interacts with the main protocol, an *agent*. For each port q in the protocol's interface, we assume an *agent* connected to that port. More precisely, if q is used as an input port by the protocol, the *agent* connected to q must use q as an output port, and vice versa. We give in Listing 3.4 an example of a definition of an agent with two ports as its arguments, one input and one output.

```
proctype agent(port p1; port p2){
        /*      p1: input, p2: output    */
        do
          :: /* action */
        od }
```
Listing 3.4: A generic structure for an agent

Each agent is defined as a proctype in Promela, and runs concurrently with the main protocol. We represent the generic behavior of an agent as an infinite sequence of non deterministic actions (with the do − od loop), but the definition of the precise behavior of an agent is left for the user. Since agents and protocol share ports, it is possible that an agent blocks until a datum is delivered at its port.

We assume a set of agents given by the user. Our compiler generates only the skeleton of an agent including the set of ports in its interface, with the direction of each port (either input or output). As an extension, we intend to make the input/output restriction of ports direction more strict, using the Promela assertion xr and xs to prevent misuse of port directionality. We later specify some properties of the desired observable behavior.

Connectors   As introduced previously, the main difference between a component and a connector is the ability for the component to block on some put or get operations on its port. We showed in the previous section a guarded command transformation for a connector, where the guard plays the role of a safety check on the state of the input

and output ports, and the command gives the values exchanged at the output ports in terms of the value taken at the input ports.

A connector is a process running concurrently to the components. We define a connector as a proctype, taking as argument all ports in the interface of the connectors. The internal operation in the connector proctype are defined based on the definition of its internal constraint. We call P the set of free port variables used in the connector's constraint, and M the set of memory variables.

We first instantiate every variable occurring free in the connector's internal constraint as a channel structure in Promela. For every port variable $p \in P$, we define a global instance of the port structure defined in previous paragraph, with the same name as the variable. Since variable have unique names, the structure is also unique for every ports. The structure for a port is global, and will later be used for checking some temporal properties. For every memory variable $m \in M$, we define a local channel of size 1. Constant symbols are mapped to their corresponding domain. Promela only supports few data types, we assume that the constants get an interpretation in one of those data type (integer, boolean, float, characters). Function symbols are mapped to inline procedures in Promela. For each function used in Reo, we assume that an inline procedure with the same name will be provided in Promela. We use, for every port or memory variables, an additional variable in Promela that will temporary store the data occurring at a port or memory. This additional variable is typically used when multiple output variables take the value of a single input variable: in this case, the value of the input variable is temporary stored, and replicated to every outputs.

Based on the result of Proposition 1 we take the connector in its guarded command form. We use $\mathcal{GC}$ for the set of guarded commands. A guarded command $g \to c \in \mathcal{GC}$ has a natural interpretation in Promela as a conditional update. The guard g is translated as a condition on the status of the port and memory channels, while the command c is an update of the port and memory status. The most common statements in the guards are `full(p.data)`, `full(p.trig)` and `full(m)`, which respectively checks whether a port `p` has an incoming data request, an outgoing data request, or if the memory `m` is full. The negation of those statement are also commonly used in the guards. In the command, we proceed for the update of ports and memories, which corresponds to statements of the kind `take(p,d)`, `put(p,d)`, `m?d`, or `m!d`, where `d` is a local variable.

The generic structure of a Promela program obtained from a deterministic formula with n guarded commands is shown in Listing 3.5.

```
proctype Protocol(port p1;...){
  /*    p1:      input , ...      */
  /* Memory declaration */
  chan m = [1] of {int}; ...
  /* Initial state */
  m!0; ...
  /* Local variables */
  int _m;  int _p1 ;  ...
  /* Guarded commands */
  do
  :: (guard_1) ->  command_1
  :: ...
  :: (guard_n) ->  command_n
  od }
```

Listing 3.5: a generic structure of a protocol

Note that the statements in the command are executed sequentially. We show in the next section that we can express some synchronous patterns as an LTL property on the state of the ports and memories.

LTL properties   We give a translation to an LTL property on the Promela translation, such that the properties of synchrony are preserved. Therefore, we show that if the LTL property should hold on the Reo circuit, then the corresponding asynchronous LTL property should hold on the Promela program generated from the Reo circuit.

The property `p_silent` is defined as the negation of the firing property, and represents all the non firing states of the port p. The property `silent` denotes the conjunction of all silent properties for all ports. Note that internal memory updates are still allowed.

The two properties `p1_before_p2` and `p1_then_p2` express some asynchronous firing for ports p_1 and p_2. The latter property is stricter, since the `silent` property requires that between the firing of ports p_1 and p_2, no other ports fire: it is true that `p1_then_p2` implies `p1_before_p2`.

We define the synchronous property in Promela as a binary relation between two ports p_1 and p_2. We say that the two ports are synchronous if it is always the case that p_1 fires and then p_2 fires (or the opposite), and all steps in between the firings satisfy the `silent` property. Synchronous property is a stricter form of asynchronous property: it is true that `sync_p1_p2` implies `async_p1_p2`.

102

Properties	Temporal formulas
p_fires	`len(p.data) != 0 && len(p.trig)!=0 &&` `X( len(p.data)==0 || len(p.trig)==0)`
p_silent	`!(p_fires)`
m_full	`len(m.data)!=0`
m_empty	`len(m.data)==0`
m_fires	`m_full && X(m_empty)) ||` `m_empty && X(m_full)`
m_silent	`!(m_fires)`
p1_before_p2	`(p1_fires -> <> (p2_fires))`
p1_then_p2	`(p1_fires -> X(silent U p2_fires))`
sync_p1_p2	`[] (p1_then_p2 \/ p2_then_p1)`
async_p1_p2	`[] (p1_before_p2 \/ p2_before_p1)`

Arguments for correctness In this section, we show that the sequence of messages exchanged between the generated Promela processes can be related to the sequence of data exchanges at a port of a Reo circuit. We first establish, based on the structure of the generated code resembling the guarded command form of the connector, that every data exchanged between Promela processes can be related to an assignment that satisfies the connector. As a consequence, every sequences of data exchanges between Promela processes can be related to a sequence of assignments in the behavior of a connector. We then show that there exists, for any sequences in the behavior of a Reo connector in its ideal environment, a set of processes in Promela such that the message exchanges between Promela processes correspond to the sequence in the behavior of the Reo connector. For simplicity and clarity, we consider a binary data domain for ports. The arguments for a larger domain are similar.

We use the semantic of Promela defined in [88] to show the correctness of our translation. The operational semantics of a Promela program $\mathcal{P}$ composed of processes P_i is defined as a graph $T = (Q, \rightarrow, q_0)$ where Q is a set of states and $\rightarrow$ is a binary relation on states. A state $q \in Q$ is a tuple $q = (l_0, ..., l_m, lv1, ..., lvm, gv)$ where each l_i is a location in process P_i, lvi is the vector of local variable values in process P_i, and gv is the vector of global variables in P. The state $q_0 \in Q$ denotes the initial state. We write $l_i \xrightarrow{st} l_i'$ if and only if there is a statement st from l_i to l_i'. The variables after executing st are $st(lv)$ and $st(gv)$. In our case, st is an assignment, `skip`, or

conditional statement; or it is an asynchronous send (resp. receive) from a non-full (resp. non-empty) channel. The initial state of a Promela program $T = (Q, \rightarrow, q_0)$ is $q_0 = (l^0, lv^{init}, gv^{init})$ and a *path* of T is a sequence of state $q_0 q_1...$ such that $(q_i, q_{i+1}) \in \rightarrow$.

A Promela program produced by a Reo compiler consists of a set of N processes running concurrently, where $N-1$ processes are agents interacting through the protocol process. Agents share ports with the protocol, such that they can *put* values on the input port of the protocol, and *take* values from the output ports of the protocol.

By construction, the connector in Promela can only change its internal variables (memories) and the global vector of variables (boundary ports) if one of its guard is true and the command is performed. Guards are boolean statements checking whether the `trig` and `data` channels of a port are full (or empty), or if the memories channels are full (or empty). Given a connector with n ports at its interface, and k internal memories, there are 2^{2n+k} possible states in the graph of the Promela process.

We show soundness of our implementation with the following arguments:

1. In the initial state, the vector gv is set to its initial value, together with all memory channels.

2. Given a vector gv of global variable, and a vector lv of local variables, the process for the connector, if scheduled, evaluate its guards. From the set of guards satisfied, one is selected non-deterministically, and the corresponding command is performed. The guard ensure that the `take` and `put` statements in the command will not block. The value exchanged at the ports, the current and next values for the memory constitute an assignment that satisfies a unique guarded command in the deterministic formula of the connector (Proposition 3).

3. If no guards are satisfied, processes for agents are scheduled, modifying the global vector of variables gv.

All sequences of assignments allowed by the connector process are included in the sequences of assignments satisfying the formula of the connector.

Completeness can be derived by showing that every solution of the guarded command corresponds to an implementation where the set of agents simulate the environment of the solution. By taking an implementation that unifies the agents, and using the non-deterministic properties of Promela, we can simulate any solution of the formula as a statement and a gv vector in Promela. Elaboration of this part remains as future work.

3.3.2   Case Study

In this section, we present an application of our approach. We show the Promela specification of a *fifo* channel, and study its LTL properties in two contexts: ideal and constrained environment. A second example is available in [63], in which we analyse a composite connector representing the protocol involved in a railway system. We show the Promela program compiled from the corresponding Reo circuit, and verify some properties of interest using SPIN.

We refer to an on-line repository [60] for reproduction of the results presented in this section. The compiler that generates the Promal code is accessible at [64]. The compiler takes as input a Treo file (Textual Reo [31]) and give an intermediate representation of the circuit as a guarded command formula described in this section.

Study on the *fifo* channel   We consider the Reo specification of *fifo* described in Example 40, where ports a and b are renamed to ports $p1$ and $p2$. We connect the channel with two components: one that produces data at the input port $p1$ and one that consumes that at the output port $p2$.

We study the properties of a *fifo* channel in two different environments:

- The ideal environment: the two components connected by the *fifo* channel, producer and consumer, behave in the ideal way. The producer is willing to produce messages infinitely often, and the consumer is willing to consume messages infinitely often.

- The constrained environment: the producer is willing to produce messages infinitely often, but the consumer consumes a finite number of messages only. We show in this case that some properties of the *fifo* channel that were satisfied in the ideal environment may now be violated due to some non ideal behavior of boundary components.

Before detailing the verification of LTL properties in these two cases, we present the Promela implementation of *fifo* channel, described in Listing 3.6. The Reo protocol is implemented by the process `Protocol`, and the producer and consumer are implemented respectively by the processes `Prod` and `Cons`.

```
proctype Protocol(port p1;port p2 ){
bit _m = 0 ;
do
:: (empty(m) && full(p1.data)) ->  take(p1,_m); m!_m
:: (full(m)  && full(p2.trig)) ->  m?_m; put(p2,_m)
```

```
od
}

init{
run Prod(p1); run Cons(p2); run Protocol(p1,p2); }
```
<div align="center">**Listing 3.6:** Promela implementation of a *fifo* channel</div>

The Promela implementations of the components producer and consumer are described in Listing 3.7.

```
proctype Prod(port a){
  do
  :: atomic{put(a,1)}
  od
}
proctype Cons(port a){
bit y;
  do
  :: atomic{take(a,y)}
  od
}
```
<div align="center">**Listing 3.7:** Promela implementation of a consumer and producer</div>

Verification in the ideal environment.   We present three properties to verify on *fifo1* connector:

- $prop_1 \equiv \Box(\textbf{p1} \implies \Diamond(\textbf{p2}))$, which states that always if the port $p1$ fires then eventually the port $p2$ fires. This property is verified. It is implemented in Promela as follows:

    ```
    ltl prop1 {[] ( p1_fires -> <> p2_fires ) }
    ```

- $prop_2 \equiv \Box(m \neq * \implies \Diamond(\textbf{p2}))$, which states that always if the buffer m is full then eventually the port $p2$ fires. This property is verified. It is implemented in Promela as follows:

    ```
    ltl prop2 {[] ( m_full -> <> p2_fires ) }
    ```

- $prop_3 \equiv \Box(m \neq * \implies \textsf{X}(\textbf{p2}))$, which states that always if m is full then in the next state $p2$ fires. This property is not verified because the port $p2$ becomes silent in the next state. It is implemented in Promela as follows:

    ```
    ltl prop3 {[] ( m_full -> X p2_fires ) }
    ```

- $prop_4 \equiv \Box(\textbf{p1} \implies \mathsf{X}(\neg\textbf{p1} \cup \textbf{p2}))$, which states that always if $p1$ fires then in the next state $p1$ becomes silent until $p2$ fires. This property is verified. It is implemented in Promela as follows:

```
ltl prop4 {[](p1_fires -> X( p1_silent U p2_fires))}
```

Remark 11. *The result of the property verification, described above, shows that the implementation of the fifo1 connector is correct. Indeed, based on the behavior of fifo1, this result is the one we expected.*

Verification in the constrained environment: In this case, the producer interacts with a finite consumer, so the number of messages that could be consumed by the is limited to 5. In Listing 3.8, is presented the Promela implementation of a finite consumer, specified by the process $consFinite()$.

We verified the properties described above, $prop1$, $prop2$, $prop3$, and $prop4$, and as expected, all these properties are not satisfied. Indeed, the properties $prop1$, $prop2$, and $prop4$ are not satisfied because of the behavior of the component consumer that prevents firing the port $p2$ after consuming the first 5 messages. Therefore their violation is due to the protocol environment. However the property $prop3$, remains not satisfied as in the case of the ideal environment.

```
proctype consFinite(port a){
bit y;
int i = 5;
do
:: i>0; atomic{take(a,y)}; i = i-1
:: break
od
}
```
Listing 3.8: a generic structure of a protocol

3.4   Related work and future work

We should mention the existing works on Reo semantics [47] and pioneer work on Reo compiler [46]. Our work expands existing work on the Reo coordination language by giving a new algebraic semantics for Reo, whose primitive components are not channels, but ports. Basic Reo channels (such as a sync or fifo channel) can be described as an algebraic product of their ports, parametrized by the proper interaction

signature. Moreover, the algebraic properties of the interaction signature provides new ways to reason about equivalent Reo expressions.

Model checker   Vereofy [14, 15, 13] is a model checking tool developed at the University of Dresden to analyze and verify Reo connectors. Vereofy has two input languages: the Reo Scripting Language (RSL), used to specify the coordination protocol, and a guarded command language called Constraint Automata Reactive Module Language (CARML), a textual version of constraint automata used to specify the behavior of components. Vereofy allows the verification of temporal properties expressed in LTL and CTL-like logics.

Our work differs from Vereofy, since we use the Treo [31] (Textual Reo language) to describe both the protocol and the boundary components. In Treo, the description of the behavior of primitive channels and components is parametric: the user has to define a semantic domain, and the Reo composition operation in that semantics. We make use of the rule based semantics [32] for channels and give the description of boundary components directly as Promela processes. Our work extended the set of backend for the compiler so that the Textual Reo description (i.e., Treo input file) compiles to a Promela program that can be used by the Spin model checker.

Denotational semantics for Reo   In [3], the authors give a co-inductive semantic model for Reo connectors, based on timed-data streams (TDS). We shall briefly highlight the main differences between such model and ours. First, the TES model for component behavior explicitly captures atomic set of events in an observation, while the TDS model implicitly represent atomicity as all firing of ports that occur at the same time. The time stamp of a TES can therefore be dropped while preserving the information of atomicity: this construction simplifies reasoning about atomicity.

Another difference between TES and TDS is that TESs are not restricted to communication ports, but can model arbitrary events. Moreover, the implicit semantics in TDS that a port fires infinitely often (which is used as an argument for fairness in [3]) is no longer assumed in the TES model, but can still be recovered if needed.

Operational semantics for Reo   The constraint automata semantics for Reo was also considered in [17] for defining and verifying bisimulation and language equivalence between Reo connectors. In [6, 7], the authors considered time constraints, and proposed a timed version of constraint automaton to verify by model checking timed CTL properties. In [52, 51], the authors use timed constraint automata and present a SAT-based approach for bounded model checking of real-time component connec-

tors. The authors proposed a framework for the verification of Reo circuits using the mCRL2 toolset (developed at the TU of Eindhoven). Their tool automatically generates mCRL2 specifications from Reo graphical models. The translation from Reo to mCRL2 uses the constraint automata semantics of Reo.

In [42], the authors provide a semantics for Reo circuits using Büchi automata, and verify temporal properties of Reo circuit. Our work differs in that our logic is first order (and not monadic second order as for Büchi automata) and we give an internal representation (as guarded commands) that minimizes the size of the resulting composition. The internal representation of a Reo circuit is then translated either to a model checker for temporal verification, or to an imperative language for execution. The tool on which this section is based has shown state of the art results [32].