# Reactive Systems with Unbounded Event Memorization

Franck Cassez Frédéric Herbreteau Olivier Roux

IRCyN (U.M.R. 6597 CNRS, ECN, Univ. Nantes, EMN)*

**Résumé**

*Dans cet article, nous étudions une machine d'états infinie pour modéliser le comportement des systèmes réactifs dans lesquels on prend en compte les mémorisations d'occurrences de certains événements. Nous montrons comment, à partir d'un système de transitions fini qui modélise la partie contrôle d'un système réactif spécifié dans le langage* Electre*, nous construisons un FIFFO-système de transitions*[a] *qui permet de modéliser la mémorisation et le traitement des occurrences enregistrées d'événements. Les différentes étapes de cette construction sont illustrées au travers du problème de synchronisation bien connu des lecteurs-écrivains.*

[a]FIFFO : First In First Fireable Out.

**Keywords**

Reactive systems, Specification languages and tools, Semantics, Unbounded Memorization.

## 1 Introduction

Hard Real-Time systems may be characterized by the following two features (possibly among other features): [1] they are subject to externally defined timing constraints, and [2] their behaviors are associated with crucial safety properties. The timing constraints are indeed specific safety properties.

For the sake of efficiency, as well as to provide simulation or validation tools in order to check real-time properties, several proposals and methods based on strong mathematical principles have appeared.

*Reactive languages*[1] are among those approaches. They are intended to describe the behaviors of the tasks of the application through the possible executions of the actions with respect to the event occurrences. An important feature of reactive languages is that time is revealed through occurrences of events.

Reactive languages may be split into two classes. The first class is that of the *synchronous* reactive languages [2] which are based on the so-called *strong synchrony hypothesis* where actions are supposed to have a null duration (Esterel [3], Lustre [4], Signal [5], Statecharts [6] are examples of such imperative or data-flow or graphical synchronous languages). The second class is that of the *asynchronous* reactive languages [7], which deal with lasting actions, non-simultaneous event occurrences and memorizations of event occurrences (which are to be processed later on). In this paper, we are mostly interested in the asynchronous model.

In many applications, the synchronous approach is too restrictive since, although it makes it possible to describe the preemptions of an action, the way this action may be later restarted or resumed from the point of interruption cannot be written in the program [8].

Nevertheless, the main asynchronous feature which is the point of this paper lays in the fact that it is often difficult to memorize the multiple occurrences of an event, which are to be taken into account later on (although, of course, memorization can be programmed in a specific module).

*1, rue de la noë, B.P. 92101 — 44321 Nantes Cedex 3, FRANCE — roux@ircyn.ec-nantes.fr

In order to take into account more easily the asynchronous applications (where actions have to be preempted and later resumed, and events have to be memorized), we have built another system: Electre [7, 9, 10]. We will see that the model for the behavior of Electre programs (namely transition systems) fits well for the description of real processes to be specified: the states of the model are isomorphic to the states of the controlled process. The main features of the ELECTRE language on which we will focus are the following ones:

- an ELECTRE program can be compiled into a *control* transition system [7] where the states stand for the lasting executions of some of the actions and the transitions stand for the event occurrences. The compilation is achieved through a calculus of conditional program rewriting rules which are attributes of the language grammar. We will introduce the control transition system in section 2;
- this control transition system has been proved complete and finite; in fact it does not deal with the memorization of events occurrences but only with their processing [7]. To upgrade the model, a list of stored occurrences of events is added in order to deal with the ordered and multiple memorizations of the events occurrences. Thus, we obtain a FIFFO-transition system (a stored occurrence of an event is processed as soon as possible, priority is given to the oldest stored occurrence, hence the name First-In/First Fireable Out) which can be related to FIFO-transition systems [11, 12]; this machine models the memorization and the processing of the events. This will be described in section 3;
- the FIFFO-transition system is the deterministic model for the behavior of the system specified by an Electre program and can be used to run the programs, as well as to simulate the possible executions in order to observe the resulting behaviors.

# 2 Control Transition Systems of Electre Programs

## 2.1 The Electre Reactive Language

### 2.1.1 Overview of the Language

Electre is a reactive asynchronous language aimed at specifying and programming real-time applications. An Electre program describing the behavior of a process is made of three types of components:

**modules:** which are tasks of the process without blocking points: each instance of a module is a piece of executable code which can be either active, preempted or idle,

**events:** which can be software or hardware originated: each occurrence of an event is a signal which can be either memorized or not (see section 3),

**operators:** combining the two previous components (for instance parallelism, sequence, preemption or launching (of a module by an event), repetition, and so on).

The term reactive means that the *system* controlling the process is to react *instantaneously* to any event occurring in the environment. Since we take into account the events occurrences one after the other (in the order in which they are received by the hardware system), two events cannot be seen to have occurred at the same instant. This is consistent with our asynchrony assumption where actions are lasting and thus launching and completion of an action are not simultaneous.

### 2.1.2 The Readers/Writers Problem

The *readers/writers* problem was originally stated and solved in [13]. There are several variations on this problem, all involving priorities. We specify our readers/writers problem here, with the following requirements: [1] several readers can read the book simultaneously, and [2] when a writer writes the book, no other process (reader or writer) can access the book.

To specify the problem in the Electre language with two readers and two writers, we proceed as follows:

- the processes readers and writers are what we call *modules*
  - $READ_1$ (respectively $READ_2$) refers to the module for reader 1 (respectively reader 2) to read the book,
  - $WRITE$ refers to the module for both writer 1 and writer 2 to write the book,
- a request to read or write the book is an *event*
  - $r_1$ (respectively $r_2$) is a request for reading the book made by reader 1 (respectively reader 2),
  - $w_1$ (respectively $w_2$) is a request for writing the book made by writer 1 (respectively writer 2).

An Electre program that specifies the behavior of the system is:

```
PROGRAM Readers&Writers ;
     loop
        await
             { @r1 : READ1 || @r2 : READ2 }
           or
             w1  : WRITE
           or
             w2  : WRITE
     end loop ;
END Readers&Writers ;
```

*Fig. 1 : Readers/writers with no memorization for reading requests.*

We shall not go into details about the syntax of the Electre language; the meaning of the above written program can be summed up as follows:

1. a request for reading is a *fleeting* event (qualifier `@`); this means that if the request can not be satisfied at the time the event occurs then the request is lost; on the contrary if the request can be satisfied, the corresponding module ($READ_1$ for request $r_1$) is launched (this is the meaning of the symbol ":"),
2. the activities of readers 1 and 2 may be run simultaneously (symbol `||`) and when $READ_1$ is being run request $r_2$ can be taken into account (the converse holds),
3. the writing activity $WRITE$ and the parallel activity $READ_1$ and $READ_2$ are in mutual exclusion (symbol `or`),
4. the requests for writing (events $w_1$ and $w_2$) can be satisfied one at a time (symbol `or`)
5. the program consists in an infinite cycle (structure `loop--end loop`) of waiting until one of the events $r_1$, $r_2$, $w_1$ or $w_2$ occurs (structure "`await`"),
6. event $w_1$ and $w_2$ are *standard* events (no qualifier before them), which means that in the case they cannot be taken into account at the moment they occur, they must be memorized for later processing.

## 2.2 Control Transition System Associated with an Electre Program

### 2.2.1 Operational Semantics [14, 7]

A state of the system is a pair $\langle B, C\rangle$ where: $\boxed{1}$ $B$ is a behavior (an Electre program), $\boxed{2}$ $C$ is the context and stands for the states of the modules. For any event $e$, we can determine the pair $\langle B', C'\rangle$ which is the state of the system after *taking into account* the occurrence of event $e$ [14, 7]. More precisely, for all states $\langle B, C\rangle$ of the system and any event $e$ that can be processed (the events that must be stored will be dealt with later), we can calculate the set of *actions* $\mathcal{A}$ that

must be performed on modules to take them from the state they were in $C$ to the new state they are in $C'$ [7]. We denote a *reaction* of the system in state $\langle B, C\rangle$ to event $e$ by:

$$\langle B, C\rangle \xrightarrow{e/\mathcal{A}} \langle B', C'\rangle$$

Since the Electre language is an asynchronous reactive language, a transition is labeled with a **single** event's name and the corresponding set of actions. The event is unique since we consider that there is no actual perception of simultaneity.

The behavioral model for Electre programs is thus a labeled transition system, transitions of which are reactions as defined above. The actions will be introduced in the model as properties of the transitions yielding a parameterized transition system.

We denote by $\mathcal{C} = (Q_{\mathcal{C}}, s^0_{\mathcal{C}}, E, \rightarrow)$ the control transition system associated with an Electre program $p$ where: $Q_{\mathcal{C}}$ is the set of states (a state is a pair $\langle B, C\rangle$); $s^0_{\mathcal{C}}$ is the initial state (the initial program $p$ and an initial state for each module); $E$ is the set of input events (events that appear in the Electre program $p$); $\rightarrow \subseteq Q_{\mathcal{C}} \times E \times Q_{\mathcal{C}}$ is the transition relation.

This transition system is enhanced with properties on states and transitions yielding a *parameterized transition system* [15]: this model contains all the information of a reaction.

### 2.2.2 The Readers/Writers Problem (Semantics)

The transition system modelling the behavior of the readers/writers program of Fig. 1 is depicted on Fig. 2. At the initial instant, the state of all the modules is *idle*. The events are also in state *non-existent* (in the sequel, when the state of an event is not mentioned, it is to be taken as non-existent).

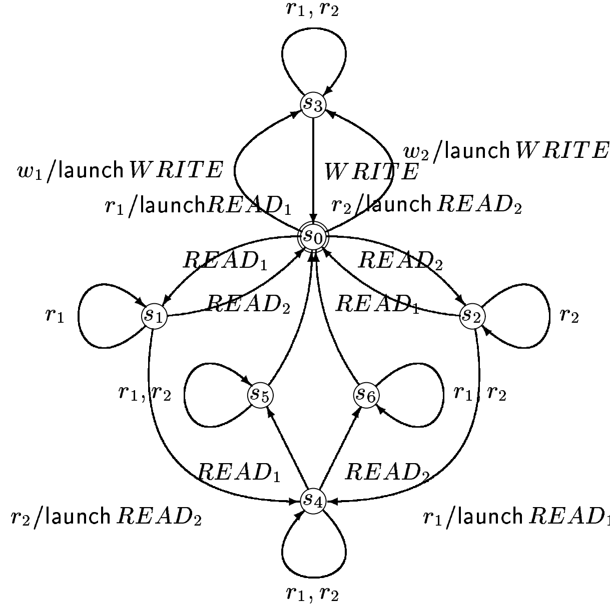


*Fig. 2 : Control transition system for our readers/writers specification of Fig. 1.*

We call this transition system the *control* transition system as it does not deal with the memorization of occurrences of events but only with the reactions to events that can be taken into account.

The important result about the compilation procedure is that for any Electre programs, the associated *control* transition system is finite with respect to the number of states and transitions [7].

So far, we have built up a model for the behavior of an Electre program, in term of reactions to occurrences of events that can be taken into account. In the next section, we will show how to improve the model so as to be able to handle occurrences of events that must be stored.

# 3 FIFFO-Transition System and Events Memorization

## 3.1 The readers/writers problem (memorization)

In the Electre program that we wrote for the specification of our readers/writers problem (Fig. 1), the two events $w_1$ and $w_2$ are *standard* events: this is different from other kinds of events, for instance, *fleeting* events like $@r_1$ and $@r_2$[1]. The *standard* event in the Electre language (no symbol before the event as in $w_1$), has *standard* properties. One of these properties is that an occurrence of such an event *must* be memorized if it can not be taken into account immediately. Moreover, only a unique occurrence of a *standard event* may be memorized. Lastly, there are *multiple-storage* events for which all the occurrences are to be memorized, in order to be processed later.

Once an occurrence of an event has been memorized, the intuitive semantics of the Electre language states that it must be processed *as soon as possible* and *in the order in which they have occurred.*

## 3.2 Relation to Control Transition System

In term of transition system or state machine that models the behavior of our system, the storage and processing of the first occurrence of event $w_1$ requires two steps:

**memorization:** from state $s_0$, as the occurrence of $r_1$ and $r_2$ can be taken into account at the time they occur (see the control transition system depicted on Fig. 2) the control transition system goes in state $s_4$; in this state, an occurrence of event $w_1$ cannot be processed (no transition labeled $w_1$ from state $s_4$) and this event is a standard event: it is memorized,

**processing:** when both $READ_1$ and $READ_2$ have completed, the control transition system goes back to state $s_0$, where the memorized occurrence of event $w_1$ can be processed: module $WRITE$ is launched; the effect of the memorized occurrence is then the same as if the occurrence had occurred on the instant it could be taken into account.

In the above example, only a single occurrence of an event may be stored: one could think of including the memorized occurrences in the states of the control transition system, by adding new states. Although it would be a simple solution, it cannot be put into practice: the Electre language enables us to use events, *all* the occurrences of which are to be processed: the *multiple-storage* event. In such a case, many occurrences of the event may be memorized, and the number of these occurrences is unbound. This is why we use a more powerful machine (as far as memorization is concerned) to model the behavior of a system specified in the Electre language.

## 3.3 Building the FIFFO-Transition System

In this section, we use a model we call FIFFO-transition system, derived from FIFO-automata [11, 12] and Pushdown-automata [17]. A FIFFO-transition system is a transition system with a memory capability managed as a FIFFO-list, and on this model we use the following notation:

- the events in the Electre program we consider belong to a finite set $E = \{e_1, e_2, \ldots, e_n\}$, and memorizable events are given by a subset $E_M \subseteq E$,
- the general form of a transition in our FIFFO-transition system is $\delta(q, e, a) = [q', b]$, where $e \in E$ and $a, b \in E_M$, which means that in state $q$, processing $e$ when $a$ is in the FIFFO-list takes us to state $q'$, $a$ is removed from the FIFFO-list and $b$ is added to the FIFFO-list.

Since we do not allow simultaneous processing of events (asynchronous assumption), transitions in our model for memorization and processing will be in the following forms:

[1] We remind the reader that an occurrence of a fleeting event is lost if it cannot be taken into account immediately; the events in the Esterel language[16] belong to this category.

**memorization:** $\delta(q, e, \xi) = [q', e]$ means that whatever the FIFFO-list contains ($\xi$), the occurrence of $e$ takes the FIFFO-transition system, to state $q'$ and $e$ is added at the end of the FIFFO-list,

**processing:** $\delta(q, \xi, e) = [q', \epsilon]$ means that whatever the current occurrence of event is ($\xi$), if $e$ is processed from the FIFFO-list, the FIFFO-transition system goes to state $q'$, $e$ is removed from the FIFFO-list and nothing is added ($\epsilon$) to the FIFFO-list.

### 3.3.1 Memorization: When?

The fact that an occurrence of an event is to be stored is an information given by the **state** of the control transition system [7]. Obviously, the problem is to deal with the memorizable events (set $E_M$). If $m$ is such an event, two cases may arise from a state $s$ of the control transition system: $\boxed{1}$ $s$ is the source of a transition labeled $m$, then the occurrence of $m$ can be taken into account immediately: there is no memorization; $\boxed{2}$ there is no transition labeled $m$ from state $s$: in this case the occurrence of $m$ must be memorized in order to be processed later.

The first steps towards the FIFFO-transition system to model the behavior of an Electre program are:

1. take the control transition system as a base: every transition $s \xrightarrow{e} s'$ yields a transition $\delta(s, e, \xi) = [s', \epsilon]$ in the FIFFO-transition system,
2. for every state $s$ of the control transition system,
   - let $Source(s)$ be the set of labels (event's name) of transitions starting from $s$,
   - let $Memo(s) = E_M \setminus Source(s)$,
   - $\forall e \in Memo(s)$, we create in the FIFFO-transition system a transition $\delta(s, e, \xi) = [s, e]$.

### 3.3.2 Processing of Memorized Occurrences: When?

Memorized occurrences of events must be processed as soon as possible and with priority to the oldest memorized occurrences. The priority will be dealt with in the sequel. In the FIFFO-transition system we only create transitions indicating that a memorized occurrence can be processed as well as the result in the state and the FIFFO-list. A memorized occurrence of an event can be taken into account in every state where an occurrence of the same event can be taken into account. Moreover, the effect of taking into account a memorized occurrence of an event is the same as taking into account an occurrence of the same event. Consequently, for all the transitions in the control transition system

$$s \xrightarrow{e} s', \ \ e \in E_M$$

we add a "clone" transition $\delta(s, \xi, e) = [s', \epsilon]$ in the FIFFO-transition system.

## 3.4 A Comprehensive Model for the Behavior of Electre Programs

In this section, we formally define a model for the behavior of Electre programs taking into account the memorizations of events. This model is a *FIFFO-transition system.*

Let $\mathcal{C} = (Q_\mathcal{C}, s^0_\mathcal{C}, E, \rightarrow)$ be a parameterized (control) transition system of type ($\mathcal{X} = \{X_1, \dots, X_n\}, \mathcal{Y} = \{Y_1, \dots, Y_m\}$) associated with an Electre program and $E_M = \{e_1, \dots, e_k\} \subseteq E$ the set of memorizable events of this program. With this program we associate a parameterized FIFFO-transition system $\mathcal{F} = (Q_\mathcal{F}, s^0_\mathcal{F}, E, E_M, \delta)$ of type ($\mathcal{X}_\mathcal{F} = \{X'_1, \dots, X'_n\}, \mathcal{Y}_\mathcal{F} = \{Y'_1, \dots, Y'_m\} \cup \{M_{e_1}, \dots, M_{e_k}\} \cup \{B_{e_1}, \dots, B_{e_k}\}$) where $Q_\mathcal{F} = Q_\mathcal{C}$ is the set of states, $s^0_\mathcal{F} = s^0_\mathcal{C}$ is the initial state, $E$ is the input alphabet, $E_M$ is the FIFFO-list alphabet, $\delta \subseteq Q_\mathcal{F} \times E \times E_M \times Q_\mathcal{F} \times E_M$ is the transition relation (a transition from $\delta$ is written $\delta(q, e, e') = [q', e'']$) defined by:

- immediate processing: $\dfrac{(q \xrightarrow{e} q') \in \ \rightarrow}{\delta(q, e, \xi) = [q', \epsilon]}$,

- batch processing: $\dfrac{(q \xrightarrow{e} q') \in \ \rightarrow \ \textit{and}\ e \in E_M}{\delta(q,\xi,e) = [q',\epsilon]}$,

- memorization: $\dfrac{(q \xrightarrow{e} q') \notin \ \rightarrow \ \textit{and}\ e \in E_M}{\delta(q,e,\xi) = [q,e]}$,

$\forall X_i' \in \mathcal{X}_\mathcal{F},\ s \in Q_{X_i'} \iff s \in Q_{X_i},\ \forall Y_i' \in \mathcal{Y}_\mathcal{F},\ \delta(q,e,\xi) = [q',\epsilon] \in T_{Y_i'}$ *and* $\delta(q,\xi,e) = [q',\epsilon] \in T_{Y_i'} \iff (q \xrightarrow{e} q') \in T_{Y_i},\ \forall M_{e_k} \in \mathcal{Y}_\mathcal{F},\ \delta(q,e_k,\xi) = [q,e_k] \in T_{M_{e_k}}$ (this subset characterizes the transitions on which $e_k$ is memorized), $\forall B_{e_k} \in \mathcal{Y}_\mathcal{F},\ \delta(q,\xi,e_k) = [q',\epsilon] \in T_{B_{e_k}}$ (this subset characterizes the transitions on which $e_k$ is batch processed).

### 3.5 Runs of the FIFFO-Transition System

The FIFFO-transition system we have built is a *static* model for the behavior specified by Electre programs in the sense that it contains information that permits to control an industrial process specified in the Electre language. To complete the definition of the behavior of the system that controls the application, we must specify how to interpret the FIFFO-transition system.

DEFINITION 3.1 (CONFIGURATION.) *A* configuration *of the FIFFO-transition system is a pair* $(s,\varphi)$ *where* $s \in Q_\mathcal{F}, \varphi \in (E_M)^*$. *A configuration* $(s,\varphi)$ *is* stable *iff no memorized occurrences of events in* $\varphi$ *can be processed from* $s$, *i.e.* $\forall e \in \varphi,\ \delta(s,\xi,e)$ *is not defined (we note this* $\delta(s,\xi,e) = \bot$*).*

For a FIFFO-list $f$ and $e, e' \in f$ we denote $e < e'$ if the oldest occurrence of $e$ in $f$ is older than the oldest occurrence of $e'$ in $f$; $f \oplus e$ is equal to $f$ with $e$ added as the newer element; $f \ominus e$ is equal to $f$ with the oldest occurrence of $e$ removed.

An *execution step* $(s,\varphi) \xrightarrow{l} (s',\varphi')$ of the FIFFO-transition system corresponds either to the processing of an occurrence of event, or to the memorization of an occurrence or to the batch-processing of a stored occurrence.

DEFINITION 3.2 (SEMANTICS OF FIFFO-TRANSITION SYSTEM.) *The semantics of the FIFFO-transition system* $\mathcal{F} = (Q_\mathcal{F}, s^0_\mathcal{F}, E, E_M, \delta)$ *is a transition system* $\mathcal{S} = (Q_\mathcal{S}, s^0_\mathcal{S}, E_\mathcal{S}, \rightarrow)$ *where:*

- $Q_\mathcal{S} = Q_\mathcal{F} \times (E_M)^*$ *is the set of states,*
- $E_\mathcal{S} = E \cup \{?e, !e, e \in E_M\}$ *is the set of labels,*
- $s^0_\mathcal{S} = (s^0_\mathcal{F}, \emptyset)$ *is the initial state,*
- $\rightarrow \subseteq Q_\mathcal{S} \times E_\mathcal{S} \times Q_\mathcal{S}$ *is the transition relation defined as follows:*
  - *when* $(s,\varphi)$ *is stable:* $\dfrac{\forall e' \in \varphi, \delta(s,\xi,e') = \bot \ \textit{and}\ \delta(s,e,\xi) = [s',\epsilon]}{(s,\varphi) \xrightarrow{e} (s',\varphi)}$, $\dfrac{\forall e' \in \varphi, \delta(s,\xi,e') = \bot \ \textit{and}\ \delta(s,e,\xi) = [s,e]}{(s,\varphi) \xrightarrow{!e} (s,\varphi \oplus e)}$,
  - *when* $(s,\varphi)$ *is unstable:* $\dfrac{\exists e \in \varphi, \delta(s,\xi,e) = [s',\epsilon] \ \textit{and}\ \forall e' < e, \delta(s,\xi,e') = \bot}{(s,\varphi) \xrightarrow{?e} (s,\varphi \ominus e)}$.

# 4 Conclusion

In this paper, we have shown that: assuming the compilation of Electre programs with no memorization of events gives a finite-state transition system; every Electre program with event memorization can be modeled by a FIFFO-transition system.

The FIFFO model can be used for simulations or real executions (tools have been developed namely SILEX for simulations, EXILE for executions). A specific real-time executive based on the

FIFFO-transition system model is run in EXILE and provides an efficient execution. Of course, the arbitrary number of events memorization is actually efficient during run time, as it is an important feature of the language.

Moreover, the framework developped for Electre programs can be extended to reactive systems with memorizations.

# References

[1] Amir Pnueli. Specification and development of reactive systems. In *Information Processing.* Elsevier Science Publishers B.V. (North Holland), 1986.

[2] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, september 1991.

[3] Frédéric Boussinot and Robert De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, september 1991.

[4] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow language LUSTRE. *Proceedings of the IEEE*, 79(9):1304–1320, september 1991.

[5] Paul Le Guernic, Michel Le Borgne, Thierry Gautier, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, september 1991.

[6] David Harel and Amir Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13, pages 477–498. NATO ASI Series, (Springer Verlag, New-York), 1985.

[7] Franck Cassez and Olivier Roux. Compilation of the ELECTRE reactive language into finite transition systems. *Theoretical Computer Science B*, 146(1–2):109–143, July 1995.

[8] G. Berry. Preemption and concurrency. In *Proc. FSTTCS 93*, Lecture Notes in Computer Science 761, pages 72–93. Springer-Verlag, 1993.

[9] Jean Perraud, Olivier Roux, and Marc Huou. Operational semantics of a kernel of the language ELECTRE. *Theoretical Computer Science*, 97(1):83–104, april 1992.

[10] Olivier Roux, Denis Creusot, Franck Cassez, and Jean-Pierre Elloy. Le langage réactif asynchrone ELECTRE. *Technique et Science Informatiques*, 11(5):35–66, 1992.

[11] Bernard Vauquelin and Paul Franchi-Zanettacci. Automates à files. *Theoretical Computer Science*, 11:221–225, 1980.

[12] Alessandra Cherubini, Claudio Citrini, Stephano Crespi Reghizzi, and Dino Mandroli. Quasi-real-time fifo automata, breadth-first grammars and their relations. *Theoretical Computer science*, 85:171–203, 1991.

[13] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[14] Jean Perraud, Olivier Roux, and Marc Huou. Operational semantics of a kernel of the language ELECTRE. *Theoretical Computer Science*, 97(1):83–104, april 1992.

[15] André Arnold. *Finite transition systems.* Prentice-Hall, 1994.

[16] Gérard Berry and Georges Gonthier. The ESTEREL synchronous language: design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[17] Thomas Sudkamp. *Languages and Machines.* Addison-Wesley, 1988.