ELSEVIER

Theoretical Computer Science 146 (1995) 109–143

Theoretical Computer Science

Compilation of the *ELECTRE* reactive language into finite transition systems☆

Franck Cassez, Olivier Roux*

LAN (U.A. CNRS 823), Ecole Centrale de Nantes, Université de Nantes, 1 rue de la noë, 44072 Nantes Cedex 03, France

Received September 1993; revised May 1994
Communicated by M. Nivat

Abstract

We present in this paper an operational semantics for the *ELECTRE* reactive language (Roux et al., 1992). This language is based on an asynchronous approach to real-time systems. First basic concepts and intuitive semantics are introduced. Then we give rules to model dynamic semantics of *ELECTRE* programs: this constitutes an operational semantics for the *ELECTRE* language. This operational semantics is used to define a model of execution for *ELECTRE* programs: transition system. In addition, we prove, using structural induction on the operational semantics, that this transition system is a *finite* state transition system. Eventually, we extend the previous transition system so as to handle multiple-storage events: it is important since the asynchronous *ELECTRE* language deals with multiple memorized occurrences of the events.

This result gives a means of compiling the *ELECTRE* language into a finite-state machine.

1. Introduction

Many reactive programming languages have appeared recently [15], most of them are based on a synchronous assumption (LUSTRE [10], SIGNAL [17], ESTEREL [4], STATECHARTS [14]). Synchronism seems to be well suited for the description of some real-time processes. But it suffers from a lack of realism. Indeed, the whole world is asynchronous, particularly when dealing with real-time processes which are to react to events arriving at unpredictable instants [18]. The assumption of asynchronism is that the perception of simultaneousness is not possible. In other words, two events cannot occur simultaneously. This is the reason why *ELECTRE* is founded on the asynchronous assumption. Furthermore, this is a nonprocedural language which

* Corresponding author. E-mail: {cassez|roux}@lan.ec-nantes.fr.

☆ Work supported by the PRC C^3 and C^2A of the CNRS.

SSDI 0304-3975(94)00136-7

deals with the possible executions of tasks like those in the paths expressions [9,16].

Real-time programming languages are also designed to be used as tools for describing real-time processes. When the description is completed, one would like to ensure that the program meets some specifications [13,19]. Reliability turns out to be the most important feature in safety critical industrial processes. Finite-state machines like automata are now well known and many tools are available to check various properties [2,29,22]. This accounts for the worldwide use of automata to model real-time processes.

In the *ELECTRE* system, we have choosen to model the execution of programs with transition systems. The first step towards the building of this model is to define a dynamic semantics: this is achieved by the means of rewriting rules which stand for the operational semantics (based on the one described in [20]) of the language. Then we can build up a transition system according to the operational semantics. This transition system defines the behaviour of a system as the set of the possible paths which are sequences of transitions from one state to another. The states stand for the current configuration of the system.

A similar methodology has been used for constructing language acceptors and has proved to be efficient for the modellization of the behaviours of the synchronous programs [5]. Indeed, the synchronous assumption makes it possible to match the ESTEREL programs with regular expressions (for it considers signals and actions as a unique concept which constitutes the alphabet). On the other hand, the duration of the actions and the memorization of the event occurrences (which are the main features of the *ELECTRE* language and which grants it with its asynchronous nature) make it difficult considering such a method (as it will be discussed in Section 3 of this paper). For these reasons, we cannot easily amount to Mealy machines, and consequently applying Brzozowski's algorithm [7]: such differences between language acceptors and models for concurrency were already stated by Plotkin [21].

The final step in the building of an execution model for *ELECTRE* programs is to extend the previous one into a FIFO-transition system [11,26], the transitions of which are calculated using the operational semantics. This model (namely the FIFO-transition system) was adopted in order to take into account the unbounded number of memorizations of event occurrences. Thus, the finiteness of our transition system is an important result since it extends the result of the synchronous reactive languages to the asynchronous *ELECTRE* language.

The work presented here consists of describing the *ELECTRE* compiling phase. Only the main features of the language will be given in this paper: comparisons between synchronous and asynchronous approaches are not considered here and can be found in [23]. In Section 2, we introduce the syntax of the language and its intuitive semantics. In Section 3, we show briefly how to build a transition system (via the intuitive semantics) representing the execution of an *ELECTRE* program. Section 4 is devoted to the operational semantics formalizing the ideas of Section 2 and it is illustrated with an example. Two important facts about this semantics are

- it may lead to an inductive definition of an execution model for *ELECTRE* programs, namely a transition system,
- nothing can be said about the finiteness of the model.

The finiteness of this transition system is proved in Section 5, where we follow the approach of Plotkin [21]: to prove facts (properties) in operational semantics, structural induction is a standard means; moreover, we agree with the idea that although the finiteness may be thought obvious, it is essential to give a formal proof of it. This latest result is fairly important since it provides a means of compiling an *ELECTRE* program into a FIFO-transition system (presented in Section 6) which is deterministic, and can be checked for various real-time properties.

2. *ELECTRE* syntax and intuitive semantics

The *ELECTRE* language is a formal tool for describing the scheduling of *modules* (tasks) with regard to *events*. In this section, the *ELECTRE* syntax is gradually introduced using a "rewriting" approach. We explain informally the main constructs of the language and their behaviours by means of a rewriting rules. A *reaction* is the response to a modification of the environment (the history of the events that have occurred until now) in order to reach a stable state of the system. In this way, the *ELECTRE* language is said to be *reactive*, and the rewriting of the program implies changes on the modules states, some are to be run, others suspended or stopped. Roughly, upon a reaction, an *ELECTRE* program "rewrites" in another program which is the part of the program left to be executed.

2.1. *A foretaste of ELECTRE*

One of the entities handled by the language are modules. Modules are defined as elementary tasks with no blocking points (for instance a loop waiting for an event to occur). Modules stand for actions whose duration is finite but not necessarily null.[1] Nothing else is assumed about the source code of these modules, and they can be written in any sequential programming language.

Modules can be modelled by automata whose states are the different states a module can be in, which are illustrated in Fig. 1. The different states are defined as follows.

- State *idle* means either that the module has never been activated, or that it terminated naturally.[2]
- State *ready* means that the module can be run by the real-time executive, it will be actually run as soon as the processor accepts it (since nothing is assumed about the scheduling, there is no point in differentiating the state "running" from the state "ready").

[1] This assumption makes *ELECTRE* different from the synchronous languages we quoted in Section 1.
[2] Natural termination means that the module has completed.

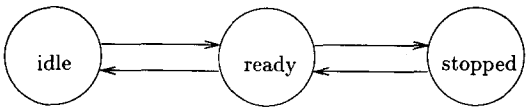

Fig. 1. The possible states of a module.

- State *stopped* means that since it has been started, something preempted it. It is neither ready to run nor completed either.

Given these definitions, we can introduce three basic constructs of the *ELECTRE* language (which are commonly present in most of the real-time programming languages):

1. sequentiality,
2. concurrency,
3. repetitive construct.

2.1.1. Sequantiality

A widespread construct in classical computer programming is the sequential one. Sequentiality between two actions means that they are to be run one after the other. The *ELECTRE* program A B. is said to be the sequential since this syntax means that module *B* is to be activated after *A* has ended (the period tells us that the end of the program has been reached). Note that program A B. implicitly means that module *A* is being run even when *B* has not yet been launched.

Assuming that c_A is an event corresponding to the completion of module *A* (in the sense of natural completion), we can guess what will happen when event c_A occurs. Module *A* has just completed and *B* is to be started. This can be synthesized as

$$\mathsf{A\,B.} \overset{c_A}{\mapsto} \mathsf{B.} \overset{c_B}{\mapsto} \mathsf{nil}$$

which means that if c_A occurs and the current behaviour is program A B. (implicitly meaning that *A* is running) the new behaviour after c_A has occurred is program B.: we call this the *rewriting* of the program. Similarly, if c_B occurs, i.e. module *B* completes, no program is to be run. This is the meaning of the nil program, which corresponds to the "empty" program. This rather simple notation will be of great help all along this section.

2.1.2. Concurrency

Real-time processes are sometimes physically concurrent. The intrinsic concurrency must be handled by a real-time language, and a mechanism to describe it must be provided. Concurrency is written in *ELECTRE* by putting two modules on both sides of the symbol ||. The program A || B. denotes a behaviour where modules *A* and *B* are being run concurrently. The behaviour denoted by the program (with c_A and c_B

having the same meaning as above) is

$$\mathsf{A} \parallel \mathsf{B}. \overset{c_A}{\mapsto} \mathsf{B}.$$

or

$$\mathsf{A} \parallel \mathsf{B}. \overset{c_B}{\mapsto} \mathsf{A}.$$

This means that both A and B are in state *ready*[3] and if either side of the parallel construct ends, the next program to be run is the remaining side. It is time we introduced two complementary symbols [and] which allow one to group many components in a same structure. For instance, consider the program [A B] || [C D].. It means that sequential components [A B] are to be run concurrently with [C D]. A parallel structure is completed when both sides of the symbol || have completed.

2.1.3. *Repetitive construct*

One of the major features of industrial processes is that they are to run forever unless something wrong has happened. This is why a repetitive construct allowing to describe cyclic processes is provided in all real-time languages. In *ELECTRE* the symbol * applied to a structure, means the structure is to be repeated an infinite number of times. A program like A*. corresponds to A followed by A, followed by A, followed by A and so on. Some other examples are given below to make the reader more familiar with the three constructs we have already seen:

$$[\mathsf{A}* \parallel \mathsf{B}] \overset{c_B}{\mapsto} [\mathsf{A}*]$$

$$[\mathsf{A}* \parallel \mathsf{B}] \overset{c_A}{\mapsto} [\mathsf{A}* \parallel \mathsf{B}]$$

$$[\mathsf{A} \parallel \mathsf{B}]* \overset{c_A}{\mapsto} \mathsf{B}[\mathsf{A} \parallel \mathsf{B}]* \overset{c_B}{\mapsto} [\mathsf{A} \parallel \mathsf{B}]*$$

From the last example, it is clear that the behaviour denoted by the program [A || B]* is the same as the one denoted by [A || B][A || B]*.

One can point out that the subject of the * operator is fairly important. In the next program

$$[\mathsf{A}* \parallel \mathsf{B}*] \overset{c_B}{\mapsto} [\mathsf{A}* \parallel \mathsf{B}*]$$

A can be run more than once while B is run, and vice versa. In the previous program [A || B]*, the same number of A and B have been run at any time during the execution.

---

[3] See Fig. 1.

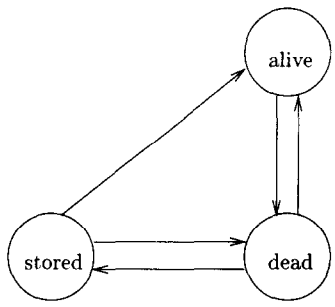

Fig. 2. The possible states of an event.

The term *reactive* for the language means that the system underlying the application described in *ELECTRE* reacts on any occurrence of event by issuing commands to the process that makes it go to a new stable state. We have until now introduced the way we handle tasks, and in the next section we are going to detail the effects of the *events*.

2.2. The events

The second type of entity *ELECTRE* can handle is the event class. Events here are logical events which are linked with software or hardware signals. Like modules, events may be in three distinct states (Fig. 2).

- State *dead* means either that no occurrence of the event has ever happened or that its occurrence has been treated.
- State *stored* tells us that an occurrence of the event has occurred, but has not yet been treated.
- State *alive* means that an occurrence of the event has activated a module structure which is not yet completed.

Event structures are usually built up with the | constructor. When single, it means a logical *exclusive or*, when doubled or tripled[4] they stand for the logical *and* that can be interpreted here as concurrency. Here are a few examples of behaviours:

$$\{e_1 \parallel e_2\} \overset{e_1}{\mapsto} \{e_2\}$$

$$\{e_1 \mid e_2\} \overset{e_1}{\mapsto} \mathsf{nil}$$

One can point out that in an event structure like $\{e_1 \parallel e_2\}$, both e_1 *and* e_2 must occur before what follows is proceeded. That is, the remainder of the program is proceeded after event structure $\{e_1 \parallel e_2\}$ has rewritten in *nil*.

[4] Differences between || and ||| are not crucial for our paper; they hint at how these structures are considered to have ended.

Now that we have seen the two basic types of components an *ELECTRE* program can be made of, we can describe how they interact. This is the subject of the next section.

2.3. Putting together modules and events

As we will see, events act as clocks ticking at unpredictable instants. Reactions of a program occur at every instant of these clocks. Events (including the completion of modules i.e. c_{module} event) "schedule" the activation and preemption of modules.

These latest features which involve taking into account *discrete events* and scheduling tasks by *suspending*, *resuming* or *activating* them are basic concepts of real-time programming.

2.3.1. Launching a module

To illustrate the operators introduced later, we shall take an example which we will be referred throughout this section. We shall consider a robot, which can move in one direction on a conductor rail. We modellize the robot by its engine which can be *on* or *off*. The engine activity corresponds to module *MOVING* in *ELECTRE*. When this module is in state *ready* (see Section 2.1), the robot is moving, and when in state *idle* or *stopped* the engine is off and the robot does not move any more. An event *present* corresponds to a sensor indicating that the robot is on a particular location and can be loaded. The last module we will use is *LOADING* (which corresponds to the loading phase of the robot). First, we describe the activity "when the robot is on the right place, it can be loaded". This can be turned to "every time event *present* occurs, launch module *LOADING*". The following structure allows to describe such a situation in *ELECTRE*. When used together with the symbol : (a colon), as in $e_1 : \mathsf{B}$, an occurrence of e_1 entails the activation of module B:

$$e_1 : \mathsf{B} \overset{e_1}{\mapsto} \mathsf{B}$$

So the robot loading phase can be described by the program *present*: LOADING. This is not the only meaning of this syntactic construct. One of the major features of the *ELECTRE* language is hidden within this construct : the liveliness of events. An event becomes *alive* at the time it occurs if it activates a module. This state of liveliness lasts until the moment the module it has activated completes (in the sense of natural end). The event then moves into the state *dead*. This transition is referred to as *consumption*. In addition, $\{e_1 \parallel e_2\} : \mathsf{B}$ will need occurrences of e_1 and e_2 to launch module B, but none of the two events will become alive since it is the *event structure* that has activated module B.

2.3.2. The preemption mechanism

Before being loaded, the robot must stop on the right place. In fact, module *MOVING* must be suspended during the loading phase. This accounts for the use of

a *preemptive* mechanism in *ELECTRE* to suspend modules. The colon symbol which enables one to launch modules has its counterparts: the symbols / and ↑. As in the program A/e., an occurrence of e would stop A and end the program

$$\mathsf{A/e.} \overset{e}{\mapsto} \mathsf{nil}$$

Thus, stopping the robot can be modelled by the program MOVING/*present* and the robot global behaviour can be the program [MOVING/*present*:LOADING]*., which implies that the robot task is repeated forever. More sophisticated constructs can be built: $\mathsf{A}/\{e_1 \mid e_2\}$ indicates that the structure enclosed in braces can preempt module A. Thus, an occurrence of either e_1 or e_2 ends the program. Let us consider now a parallel event structure $\mathsf{A}/\{e_1 \parallel e_2\}$., the behaviours of which is given below (where **1** is a special module for waiting, which will be discussed in the following):

$$\mathbf{1}/\{e_1 : A \parallel e_2 : B\}. \overset{e_1}{\mapsto} [A \parallel \mathbf{1}/e_2 : B] \overset{e_2}{\mapsto} [A \parallel B]$$

Of course, the semantics of the two symbols / and ↑ are not similar. In fact, their behaviours differ only when the structure threatened of preemption has ended before any preempting events have occurred. The / symbol is called the *necessary preemption* symbol. In the program A/e. if module A completed before e occurs, then it is again *necessary* to wait for e to occur before any further execution. On the contrary, the program A ↑ e. ends either when A completes or e occurs. The next section will highlight more formally these major differences.

2.3.3. The unit module: **1**

In the previous program A/e., we have avoided saying what would happen if the completion of module A occurs before event e does. We just claim that event e was mandatory being waited for. But how to express this? A special module called the *unit* module and denoted by **1** is an *idle* module which is doing nothing and can be thought of as a background task (on the contrary, other modules are supposed to complete in a bounded time). This particular module is also an never-ending module. Thus, waiting for event e is equivalent to the program **1**/e.. Now we can give all the possible behaviours corresponding to the two previous programs

$$\mathsf{A/e.} \overset{c_A}{\mapsto} \mathbf{1}/\mathsf{e.} \text{ followed by } \mathbf{1}/\mathsf{e.} \overset{e}{\mapsto} \mathsf{nil}$$

and

$$\mathsf{A/e.} \overset{e}{\mapsto} \mathsf{nil}$$

$$\mathsf{A} \uparrow \mathsf{e.} \overset{c_A}{\mapsto} \mathsf{nil}$$

$$\mathsf{A} \uparrow \mathsf{e.} \overset{e}{\mapsto} \mathsf{nil}$$

We can then see in program A/e. that event e is bound to happen before any further sequential execution; hence, the name of *mandatory preemption* associated with the / symbol.

As in the real world, event can be of different natures. For instance, event *present* we used above stands for the arrival of the robot on a stop. But the robot can bounce on it making event event *present* occurring a lot of times. This can be a tedious task to try and handle such a situation unless the "bouncing" nature of event *present* can be taken into account. In *ELECTRE* this can be done using *qualified event*. This is why properties characterizing their intrinsic natures can be associated with the events.

2.3.4. *Properties of the events*

A major feature of the *ELECTRE* language is that it enables the program writer to associate *properties* with every event. Apart from the standard property (which is the default one), three others are available.

Standard: An event appearing bare like "e" is a standard event which may be stored at most once.

Multiple-storage: An event like "$\#e$" is a multiple-storage event, all the occurrences of event e are to be treated.

Fleeting: "$@e$" is the so-called fleeting event. If not waited for, it is ignored.

Early-consumed: "$e" is consumed as soon as it is treated. It can never be alive even if it activates a module like in A/$e : B..

All these properties play an important role in the language but there is no point in describing the way they are treated in the rewriting rule since it is quite trivial an exercise. These properties emerged from real-time and asynchronous considerations. We intended to model the event as a complex structure following the approach of Winskel [28].

2.3.5. *Properties of the modules*

Properties may also be attached to modules. In fact, two properties can be added to the standard one.

Standard: A module appearing bare like "A" is a standard module. If preempted and activated again, it resumes at the point it was interrupted.

Initial restart: A module qualified with the $>$ as in $>A$ means that if module A happens to be preempted, it will start again at the initial point and not at the point it was interrupted.

Nonpreemptive: The symbol ! before a module identifier as in "$!A$" means that module A cannot be interrupted if it is running. When initiated, it must complete before any event is taken into account.

As an example of the nonpreemptive qualifier, we take the program !A/e : B.. The behaviours denoted by this program under the sequence of occurrences e, c_A are

$$!A/e : B. \overset{e}{\mapsto} !A\ B. \overset{c_A}{\mapsto} B.$$

Now that we have glimpsed the main features of the language, we shall go into details about the rewriting process.

3. An execution module for ELECTRE programs: transitions systems

We describe in this section the underlying execution model for the programs written in the *ELECTRE* language. We introduce the system through an example, before we give the definitions, and then we will point out the advantages and difficulties of such a modellization.

3.1. Example

Before being more specific about the system, let us take the program of the example given in Section 2.3.2 which is slightly complicated using the repetition operation for the initial parallel structure

$$[\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \quad (p_0)$$

From this program, it is obvious that two events may occur, namely e_1 and e_2. Moreover, remember that the unit module denoted by **1** stands for the idle task, which means that none of the module A and B is currently running when the program to execute is (p_0) above.

Since the following transition specifies the behaviour of the parallel construct

$$[\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]. \overset{e_1}{\mapsto} [A \parallel \mathbf{1}/e_2 : B].$$

and since the repetitive operation involves a sequential construct (Section 2.1.3), we easily get the appropriate reactions for the whole program upon the occurrence of each event

$$[\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \overset{e_1}{\mapsto} [A \parallel \mathbf{1}/e_2 : B][\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \quad (p_0 \to p_1)$$

$$[\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \overset{e_2}{\mapsto} [\mathbf{1}/e_1 : A \parallel B][\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \quad (p_0 \to p_2)$$

We will show in the next section how these transitions are calculated.

Let us call (p_1) (resp. (p_2)) the new program obtained after the e_1-transition (resp. e_2-transition), from the program (p_0).

In each case, we get the program which is left to be run. Besides, we know that the program "$[A \parallel \mathbf{1}/e_2 : B][\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*$" specifies that A is now running while B is still idle, since e_2 is being waited for. As a matter of fact, e_1 is alive while e_2 is deal (cf. Section 2.2).

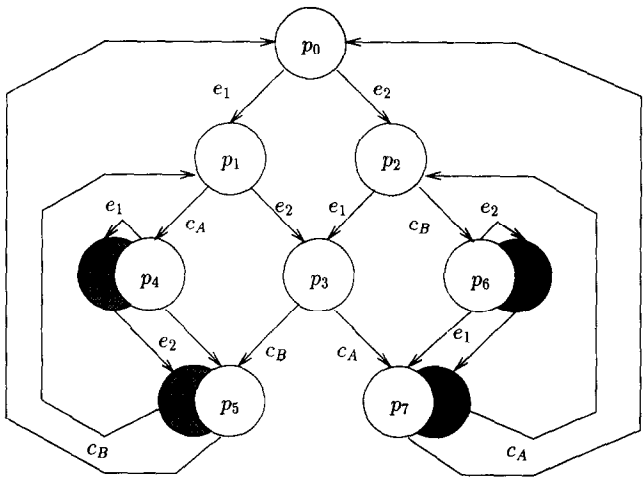

Fig. 3. The transition system of the example.

From the state where (p_1) specifies the program which is currently left to be executed, it is obvious that three events can now occur: e_1 which has no effect upon the execution (and thus leaves the program unchanged), e_2 which is being waited for, and c_A the completion of the currently running module A. The occurrence of e_2 starts the second parallel branch and leads to the following program:

$$[A \parallel B][\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \quad (p_3)$$

where both A and B are running, and when they have successively completed, the specification of the execution is given by the initial repetitive structure (p_0) anew. Notice that (p_3) may also be reached from (p_2) symmetrically after the occurrence of e_2. From (p_1) again, the latter possible event (i.e. c_A) terminates the first parallel branch which is expressed by the program (p_4)

$$[\mathbf{1}/e_2 : B][\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \quad (p_4)$$

from which the occurrence of e_2 entails the transition reaching the new state, the program of which is

$$[B][\mathbf{1}/\{e_1 : A \parallel e_2 : B\}]*. \quad (p_5)$$

The above-depicted transition system for the execution of the program is illustrated below (see Fig. 3)[5].

From this figure, the following meaningful feature has to be noticed. The state which is associated with the program "$[\mathbf{1}/e_2 : B]$" (p_4) was cloned in order to take into

[5] The states p_6 and p_7 are, respectively, similar to p_4 and p_5: they are reached by the symmetric transitions from p_2 and p_3 exchanging e_2 with e_1 and c_A with c_B.

account the memorizationof one new occurrence of e_1. Indeed, states are not featured only with the program, but also with the event memorization (at most one). In the example, this eventually leads to different transitions, upon the occurrence of c_B, from the cloned states associated with the program "$[B]$". Taking into account the states of events (as well as those of modules) is characteristic of our approach since it is linked with the asynchronous nature of the language: the number of memorizations of event occurrences is not bounded, and thus we will use a FIFO-transition system to build a comprehensive model of execution for *ELECTRE* programs (see Section 6).

3.2. Transition system: definitions

Four meaningful observations emerged from the example:

- any event occurrence (or module completion) transforms a program in a program,
- the program determines the future possible reactions,
- different paths can reach the same state,
- even if the execution of the program is indefinitely repeated, only a finite number of states build up the transition system for the program.

This induces a formal model of execution for the *ELECTRE* programs. Each program is mapped to a transition system where states and transitions are defined as follows.

State: It denotes a couple $\langle Text, Context\rangle$ where the first component *Text* is the remainder program to be run and the second component *Context* is a snapshot of the current configuration. This configuration reveals the operational status of the basic components (modules and events) of the original program. Figs. 1 and 2 show these different status: *idle*, *ready* or *stopped* for modules and: *dead*, *stored* or *alive* for events. From the operational point of view, this information is significant since, for instance, there is a difference between the two states: $\langle$B., A stopped, B ready, e alive$\rangle$ and $\langle$B., A idle, B ready, e alive$\rangle$ (which can be reached from the state: $\langle$A ↑ e : B., A ready, B idle, e dead$\rangle$).

Transition: It is labelled by either a single event or a currently possible module completion, which is indeed a particular event. Clearly, the number of events to be considered in one state is finite. We do not allow two events to occur simultaneously within the same transition, since the continuous model of time leads to the asynchrony hypothesis. Nevertheless, properties of the event allowed in an *ELECTRE* program lead us to use a special device called a FIFO-transition system [11, 26] which is derived from the pushdown automaton described in [3, 25]. The main difference is the FIFO-list used in an FIFO-automaton instead of the LIFO-list used in a pushdown automaton.

3.3. Issues of such a mapping

The transition system is the operational model for the execution of an *ELECTRE* program. The following three advantages ensue from this achievement.

- The obtained transition systems are deterministic. This ensures that any sequence of event occurrences will always lead to the same reaction. This is a safe property for real-time systems;
- furthermore, the possible reaction during a program execution can be foreseen since it corresponds to one particular path in the transition system. Thus, some behavioural properties (such as safety or liveness properties) can be verified for the program of an application. This is achieved owing to the transition systems analyzers;
- lastly, the efficiency of the underlying real-time executive is improved by using such a model, since it is very straightforward to find out the transition to fire (and consequently the state to reach any event occurrence). Moreover, the transitions hold the operational instructions that must be performed on modules.

On the other hand, the following two requirements must be fulfilled:

- the language must be endowed with a formal semantics which allows the definition of the depicted transition system. Such a semantics is presented in Section 4;
- we have to be sure that this transition system is finite whatever the source program is. Otherwise, the compilation might be impossible for some *ELECTRE* programs. This finiteness is proved in Section 5.

4. Dynamic semantics of ELECTRE programs

In the previous sections, we have shown how to build up a transition system corresponding to the behaviour of an *ELECTRE* program.

The keypoint in this building process is the calculus of the program which is left to be executed after an event has occurred. This point has been left apart so far, and what we will explain in this section is how this calculus is achieved.

We shall first introduce basic notions of *rewriting of derivations*,[6] which will be illustrated in two examples.

4.1. Operational semantics

Given an *ELECTRE* program p, if an event e occurs, the program which is left to be executed is p′ denoted by

$$\mathsf{p} \overset{e}{\mapsto} \mathsf{p}'$$

The calculus of p′ is based on the *derivation of* p (an abstract syntax tree) according to the *ELECTRE* grammar and starting from axiom S, $S \overset{*}{\to} \mathsf{p}$. In the sequel, we adopt

[6] This stands for rewriting of abstract syntax trees, which are derivations.

the notation introduced by Plotkin [21],

$$Env \vdash t \triangleright t'$$

which means that under environment *Env*, the *evaluation* (rewriting) of t is t'. Consequently, we can denote the *rewriting of the derivationof* p (brought about by an occurrence of event e) in the following way:

$$\{e\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S \xrightarrow{*} \mathsf{p}')$$

where

- $. \rightarrow .$ stands for derivation made of a single step,
- $. \xrightarrow{*} .$ stands for a derivation of zero or more steps,
- $\{e\}$ is the *environment*, i.e. the event that brought about the rewriting.

This rewriting phase yields program p′ as the terminal string of the evaluation of $S \xrightarrow{*} \mathsf{p}$ (the rewriting of the derivation $S \xrightarrow{*} \mathsf{p}$).

4.1.1. Conditional rewriting rules

The operational semantics of the *ELECTRE* language consists of a set of rules of the form

$$\frac{\{e\} \vdash d_1 \triangleright d_1' \ \wedge \ \{e\} \vdash d_2 \triangleright d_2' \ \wedge \ \cdots \ \wedge \ \{e\} \vdash d_n \triangleright d_n'}{\{e\} \vdash d = d_1 d_2 \ldots d_n \triangleright d' = \Phi(d_1', d_2', \ldots, d_n')}$$

giving the evaluation (rewriting of derivations) of a derivation d from the rewriting of subderivations of $d, d_1, d_2, \ldots, d_n$: this is equivalent to the calculus of a synthesized attribute (the evaluation) on the *ELECTRE* grammar.

4.1.2. Initializing step of the calculus: the axioms

The axioms are provided by production rules of the *ELECTRE* grammar which derives terminal symbols. As an example of such a rule, we shall take production rule $E ::= \mathsf{e}$ which derives event identifier (e) of the language. The conditional rewriting rules for production $E ::= \mathsf{e}$ are

$$\{e'\} \vdash (E \rightarrow \mathsf{e}) \triangleright \mathsf{nil} \quad \text{if } e = e'$$

$$\{e'\} \vdash (E \rightarrow \mathsf{e}) \triangleright (E \rightarrow \mathsf{e}) \quad \text{if } e \neq e'$$

where *nil* is the empty derivation, which means that if the considered occurrence of event (the environment) is actually e, then the corresponding derivation "vanishes" in the rewritten derivation. Equivalent initializing rewriting rules are given for all production rules, the right-hand side of which is made only of terminal symbols.

In the sequel, we will distinguish the evaluation of a derivation (the rewriting of the derivation) using the prime symbol (′): whenever X' appears in a derivation, it denotes terminal symbol X of the *ELECTRE* grammar as well.

To sum up, the calculus of the rewriting of a derivation $S \xrightarrow{*} \mathsf{p}$ under environment $\{e\}$ amounts to proving

$$\{e\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S' \xrightarrow{*} \mathsf{p}')$$

for some p' and as a result we get the program p' which is left to be executed after event $\{e\}$ has occurred. We give two practical examples of rewriting of derivations involving a parallel and a repetitive production rule.

4.2. μ-ELECTRE

From now on, we will only consider a subset of the *ELECTRE* grammar, which we call μ-*ELECTRE*: everything proved or said about this subgrammar is straightforwardly extended to the whole *ELECTRE* grammar.

The μ-*ELECTRE* grammar we use is given in Fig. 4.[7]

Production rule 1 derives a repetitive structure: the meaning of the two "P" will become clearer at the end of this section.

We use two different parallel symbols (rules 2 and 4), one for parallel event structures $\|_e$ and the other for parallel control structures $\|_c$: there is no syntactic need to do so; we only distinguish them for semantic reasons.

$$S ::= [P] * ([P]) \quad (1)$$

$$P ::= C \,[]\, C \,\|_c\, P \quad (2)$$

$$C ::= \mathbf{1}/I \,[]\, M \quad (3)$$

$$I ::= \{K \,\|_e\, K\} \,[]\, K \quad (4)$$

$$K ::= E : M \quad (5)$$

$$E ::= \mathsf{e} \quad (\textit{event\_identifier}) \quad (6)$$

$$M ::= \mathsf{M} \quad (\textit{Module\_identifier}) \quad (7)$$

Fig. 4. μ-*ELECTRE* grammar.

We focus now on the rewriting of derivations of program

$$[\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \,\|_e\, \mathsf{e}_2 : \mathsf{B}\}] *$$

when e_1 occurs. This program is practically translated in

$$\mathsf{p} = [\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \,\|_e\, \mathsf{e}_2 : \mathsf{B}\}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \,\|_e\, \mathsf{e}_2 : \mathsf{B}\}])$$

[7] The symbol [] stands for alternative in production rules.

before rewriting, and the corresponding leftmost derivation begins with

$$S \to [P_1^1] * ([P_1^2]) \to [C_2] * ([P_2]) \to [\mathbf{1}/I_3] * ([P_3]) \to \cdots$$

To obtain the program p′ which is left to be executed after e_1 has occurred, we need to prove

$$\{e_1\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S \xrightarrow{*} \mathsf{p}')$$

4.2.1. Rewriting of derivations on a parallel event structure

The rewriting rules associated with each production rule of *μ-ELECTRE* grammar are given in the appendix. As stated by rule 1,

$$\{e_1\} \vdash (S \to [P_1^1] * ([P_1^2]) \xrightarrow{*} \mathsf{p}) \triangleright (S \xrightarrow{*} \mathsf{p}')$$

holds if we can prove

$$\{e_1\} \vdash (P_1^1 \xrightarrow{*} \tilde{p}_1^1) \triangleright (P_1' \xrightarrow{*} \tilde{p}_1')$$

or

$$\{e_1\} \vdash (P_1^1 \xrightarrow{*} \tilde{p}_1^1) \triangleright \mathrm{nil}.$$

One of the two previous assertions holds if we can prove a series of assertions in the top-down way following the steps of the derivation starting from P_1^1:

Derivations	Production rule involved
$P_1^1 \to C_2$	$P ::= C$
$\to \mathbf{1}/I_3$	$C ::= \mathbf{1}/I$
$\to \mathbf{1}/\{K_4^1 \parallel_e K_4^2\}$	$I ::= \{K \parallel_e K\}$
$\to \mathbf{1}/\{E_5 : M_5 \parallel_e K_5\}$	$K ::= E : M$
$\to \mathbf{1}/\{\mathsf{e}_1 : M_6 \parallel_e K_6\}$	$E ::= \mathsf{e}_1$
$\to \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e K_7\}$	$M ::= \mathsf{A}$
$\to \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e E_8 : M_8\}$	$K ::= E : M$
$\to \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : M_9\}$	$E ::= \mathsf{e}_2$
$\to \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}$	$M ::= \mathsf{B}$

Obviously, the truth of the various assertions will depend on the axioms. Each rewriting of derivations starting from a nonterminal X_n (the subscript of which is n) depends on the rewriting of the subderivations starting from nonterminals derived from X_n and the subscripts of which are $n + 1$. This is why we practically proceed the

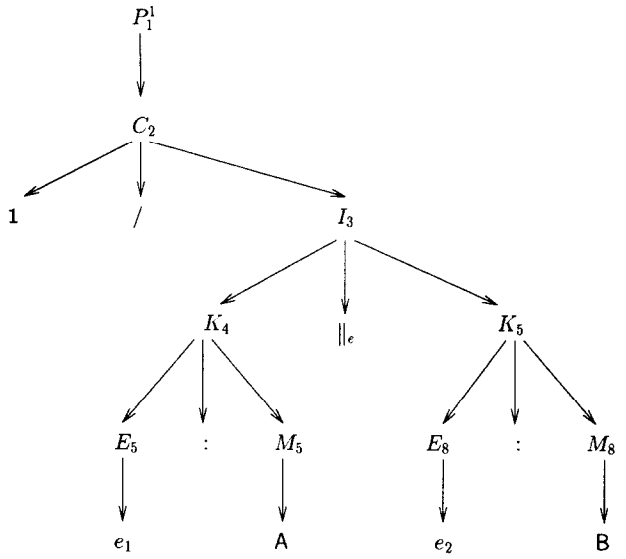

Fig. 5. Derivation tree starting from P_1^1.

calculus in the bottom-up way, from the leaves of the derivation tree corresponding to program p (Fig. 5) to the root.

Initializing step: For production rule $E ::= \mathsf{e}$, the rewriting rules given in the appendix are

$$\{e'\} \vdash (E \to \mathsf{e}) \triangleright \mathrm{nil} \quad \text{if } e = e',$$

$$\{e'\} \vdash (E \to \mathsf{e}) \triangleright (E' \to \mathsf{e}) \quad \text{if } e \neq e'.$$

Thus, we can deduce from these rules the following two facts:

$$\{e_1\} \vdash (E_5 \vdash \to e_1) \triangleright \mathrm{nil}$$

$$\{e_1\} \vdash (E_8 \to \mathsf{e}_2) \triangleright (E' \to \mathsf{e}_2)$$

As similar conditional rewriting rules exist for production rule $M ::= \mathsf{M}$, we also have

$$\{e_1\} \vdash (M_6 \to \mathsf{A}) \triangleright (M' \to \mathsf{A})$$

$$\{e_1\} \vdash (M_9 \to \mathsf{B}) \triangleright (M' \to \mathsf{B})$$

Bottom-up rewriting of derivations: Given the four previous facts, we can prove rules in the two derivations starting, respectively, from K_4^1 and K_5 applying, respectively, rewriting rules (5.2) and (5.1):

$$\frac{\{e_1\} \vdash (E_5 \to \mathsf{e}_1) \triangleright \mathrm{nil} \ \wedge \ \{e_1\} \vdash (M_6 \to \mathsf{A}) \triangleright (M' \to \mathsf{A})}{\{e_1\} \vdash (K_4^1 \to E_5 : M_5 \overset{*}{\to} \mathsf{e}_1 : \mathsf{A}) \triangleright (C' \to M' \overset{*}{\to} \mathsf{A})}$$

and

$$\frac{\{e_1\} \vdash (E_8 \to \mathsf{e}_2) \triangleright (E' \to \mathsf{e}_2) \ \wedge \ \{e_1\} \vdash (M_9 \to \mathsf{B}) \triangleright (M' \to \mathsf{B})}{\{e_1\} \vdash (K_5 \to E_8 : M_8 \xrightarrow{*} \mathsf{e}_2 : \mathsf{B}) \triangleright (K' \to E' : M' \xrightarrow{*} \mathsf{e}_2 : \mathsf{B})}$$

Now it is possible to climb one step up in the derivation tree of Fig. 5 and to prove with rule (4.4):

$$\frac{\{e_1\} \vdash (K_4^1 \xrightarrow{*} \mathsf{e}_1 : \mathsf{A}) \triangleright (C' \xrightarrow{*} \mathsf{A}) \ \wedge \ \{e_1\} \vdash (K_5 \xrightarrow{*} \mathsf{e}_2 : \mathsf{B}) \triangleright (K' \xrightarrow{*} \mathsf{e}_2 : \mathsf{B})}{\{e_1\} \vdash (I_3 \to \{K_4^1 \parallel_e K_4^2\} \xrightarrow{*} \{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}) \triangleright (P' \to C' \parallel P_1' \xrightarrow{*} \mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B})}$$

The last two steps can be proved using, respectively, rules (3.4) and (2.3):

$$\frac{\{e_1\} \vdash (I_3 \xrightarrow{*} \{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}) \triangleright (P' \xrightarrow{*} \mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B})}{\{e_1\} \vdash (C_2 \to \mathbf{1}/I_3 \xrightarrow{*} \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}) \triangleright (P' \xrightarrow{*} \mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B})}$$

and

$$\frac{\{e_1\} \vdash (C_2 \to \mathbf{1}/I_3 \xrightarrow{*} \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}) \triangleright (P' \xrightarrow{*} \mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B})}{\{e_1\} \vdash (P_1^1 \to C_2 \xrightarrow{*} \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}) \triangleright (P' \xrightarrow{*} \mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B})}$$

Eventually, we get

$$\{e_1\} \vdash (P_1^1 \xrightarrow{*} \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}) \triangleright (P' \xrightarrow{*} \mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B})$$

4.2.2. Rewriting of derivations in a repetitive structure

To prove $\{e_1\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S' \xrightarrow{*} \mathsf{p}')$ for some p', it remains to apply rule (1.1):

$$\frac{\{e_1\} \vdash (P_1^1 \xrightarrow{*} \mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}) \triangleright (P' \xrightarrow{*} \mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B})}{\{e_1\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S' \xrightarrow{*} [\mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\})])}$$

where $\mathsf{p}' = [\mathsf{A} \parallel_c \mathbf{1}/\mathsf{e}_2 : \mathsf{B}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\})]$. The terminal string of this new derivation is then the program which is left to be executed after e_1 has occurred.

To illustrate the use of the second rewriting rule (1.2) associated with the repetitive production, just consider the sequence of occurrences of events e_2 followed by c_A (end of module A) in the program obtained after the occurrence of e_1; the corresponding sequence of rewriting is[8]

$$[\mathsf{A} \parallel_c \mathbf{1}/\{\mathsf{e}_2 : \mathsf{B}\}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}]) \overset{e_2}{\mapsto} [\mathsf{A} \parallel_c \mathsf{B}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}])$$

and

$$[\mathsf{A} \parallel_c \mathsf{B}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}]) \overset{c_A}{\mapsto} [\mathsf{B}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \parallel_e \mathsf{e}_2 : \mathsf{B}\}])$$

[8] We return for a while to the intuitive notation of Section 2, but the results can be deduced with the rewriting rules.

In the later program (we refer to as p'') the derivation of which looks like

$$S \to [P_1^1] * ([P_1^2]) \xrightarrow{*} \mathsf{p}''$$

if c_B occurs, with environment $\{c_B\}$, we can prove

$$\{c_B\} \vdash (P_1^1 \xrightarrow{*} \mathsf{B}) \triangleright nil$$

Now the only way to prove

$$\{c_B\} \vdash (S \xrightarrow{*} \mathsf{p}'') \triangleright (S' \xrightarrow{*} \mathsf{p}''')$$

is to use rule (1.2) which yields

$$\frac{\{c_B\} \vdash (P_1^1 \xrightarrow{*} \mathsf{B}) \triangleright nil}{\{c_B\} \vdash (S \to [P_1^1] * ([P_1^2]) \xrightarrow{*} \mathsf{p}'') \triangleright (S' \xrightarrow{*} [\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \|_{\mathsf{e}} \mathsf{e}_2 : \mathsf{B}\}] * ([\mathbf{1}/\{\mathsf{e}_1 : \mathsf{A} \|_{\mathsf{e}} \mathsf{e}_2 : \mathsf{B}\}]))}$$

which is the next cycle and also the initial one.

4.3. Building the transition system: an endless task?

The rewriting rules of the appendix associated with each production rule of the *μ-ELECTRE* grammar provide an operational semantics for the *ELECTRE* language.

From this semantics, we have shown in Section 3 how to build a transition system modellizing the behaviour of an *ELECTRE* program. Each state of this transition system corresponds to an *ELECTRE* program which is the part of the initial program left to be executed from this point. As a matter of fact, 2 states of the transition system are different if the corresponding programs are syntactically different.[9]

Then, given an *ELECTRE* program p, we want to build the transition system associated with p. The initial state of this transition system corresponds to program $\mathsf{p}_0 = \mathsf{p}$. Suppose it is possible to "rewrite" p_0 on occurrences of events appearing in p_0 such that we obtain a countable infinite set of *different* programs

$$\mathsf{p}_0 \overset{e_{i_1}}{\mapsto} \mathsf{p}_1 \overset{e_{i_2}}{\mapsto} \mathsf{p}_2 \mapsto \cdots \overset{e_{i_n}}{\mapsto} \mathsf{p}_n \mapsto \cdots$$

then the number of states of the associated transition system would be infinite.

We will show in the next section that this situation cannot arise. Therefore, the transition system associated with an *ELECTRE* program is finite, and the algorithm to build it completes in a finite time: it is a means of compiling *ELECTRE* programs into transition systems.

A semantics is "sound" if basic requirements are fulfilled: what we denote by basic requirement is, for instance, proving that it is *complete* in the sense that at least one

[9] In this sense two programs describe the same behaviours if their texts are strictly equal; no other equivalence on behaviours should be considered.

theorem of the form

$$\{e\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S \xrightarrow{*} \mathsf{p}')$$

can be proved under all environment and for all programs p. Another point should be considered on the rewriting rules; for instance, rule (2.6)

$$\frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (P_1' \xrightarrow{*} \tilde{p}_1') \ \wedge \ \{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright (P_2' \xrightarrow{*} \tilde{p}_2')}{\{e\} \vdash (P_1 \to C \parallel_c P_2 \xrightarrow{*} \tilde{c} \parallel_c \tilde{p}_1) \triangleright (P' \xrightarrow{*} \tilde{p}_1' \parallel_c \tilde{p}_2')}$$

is tricky for the *possibility* of finding a derivation $P' \xrightarrow{*} \tilde{p}_1' \parallel_c \tilde{p}_2'$ is not obvious; actually if no derivation can be found we could be "stuck". And last but not least, it would be wise to state that the semantics is deterministic, i.e.

$$\{e\} \vdash d \triangleright d' \ \wedge \ \{e\} \vdash d \triangleright d'' \ \Rightarrow \ d' = d''.$$

The finiteness of the transition system can also be viewed as a property in our operational semantics: we again follow the idea of Plotkin [21, pp. 45–50] where a standard tool for proving properties in operational semantics is *structural induction*:

> "Another possibility [to prove statements in an operational semantics using structural induction] is to use induction on some measure of the *size* of the proof ..."

In the next section we will only prove the property leading to the finiteness of the transition system: the other ones, completeness, absence of "stuck", determinism, can be checked for as well.

5. Finiteness of the transition system associated with an *ELECTRE* program

The aim of this section is to prove that given an *ELECTRE* program p, the number of syntactically distinct programs obtained by rewriting p under whatever sequence of events is finite.

5.1. Sketch of the proof

As we pointed out at the end of Section 4, we will establish a property in the *ELECTRE* operational semantics that enables us to carry out the finiteness of the transition system. Before dealing with this property, we give the frame of the global proof, based on this property.

First we define the two sets

$$M_{\mathsf{p}} = \{\mathsf{M}_1, \mathsf{M}_2, \ldots, \mathsf{M}_j\} \quad \text{and} \quad E_{\mathsf{p}} = \{\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_k\},$$

which, respectively, denote the set of module identifiers (M_{p}) and the set of event identifiers (E_{p}) written in a program p. During the rewriting phase of program p under

whatever occurrence of event, no new identifier appears: this can be seen directly in the conditional rewritig rules (see the appendix). Thus, if $M_{\mathsf{p}'}$ and $E_{\mathsf{p}'}$ are, respectively, the set of module identifiers and the set of event identifiers written in program p' with p' such that

$$\exists e \quad \{e\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S \xrightarrow{*} \mathsf{p}')$$

then

$$M_{\mathsf{p}'} \subseteq M_{\mathsf{p}} \quad \text{and} \quad E_{\mathsf{p}'} \subseteq E_{\mathsf{p}}$$

This can be established also by structural induction in the rewriting rules.

Indeed, terminal symbols might appear in the rewriting of the derivation of program p (see rules (4.4), (4.5) or (4.6) for instance), but these symbols belong to the finite set

$$T^{\mu} = \{\mathbf{1}, [,], (,), *, \|_e, \|_c, \{, \}, /, :\}$$

which are terminal symbols of the *μ-ELECTRE* grammar (see Fig. 4) different from event and module identifiers.

In view of the preceding remarks, we can deduce that under whatever sequence of rewriting of derivations starting from program p,

$$\mathsf{p} \overset{e_{i_1}}{\mapsto} \mathsf{p}_1 \overset{e_{i_2}}{\mapsto} \mathsf{p}_2 \mapsto \cdots \overset{e_{i_n}}{\mapsto} \mathsf{p}_n \mapsto \cdots$$

all the lexical units of programs p_r belong to the finite set

$$M \cup E \cup T^{\mu}$$

Therefore, if we can prove that the "length" Λ, some measure which is greater than the number of lexical units contained in the terminal string of a derivation such as $S \xrightarrow{*} \mathsf{p}$, is monotonic on the p_r and decreases as

$$\forall r \quad r' \in \mathbb{N},\ r \leqslant r' \ \Rightarrow\ \Lambda(S \xrightarrow{*} \mathsf{p}_r) \geqslant \Lambda(S \xrightarrow{*} \mathsf{p}_{r'})$$

then we could establish that under whatever sequence of rewriting of derivations starting from $S \xrightarrow{*} \mathsf{p}$,

$$\{e_{i_1}\} \vdash (S \xrightarrow{*} \mathsf{p}) \triangleright (S \xrightarrow{*} \mathsf{p}_{i_1})$$

$$\{e_{i_2}\} \vdash (S \xrightarrow{*} \mathsf{p}_{i_1}) \triangleright (S \xrightarrow{*} \mathsf{p}_{i_2})$$

$$\vdots$$

$$\{e_{i_n}\} \vdash (S \xrightarrow{*} \mathsf{p}_{i_{n-1}}) \triangleright (S \xrightarrow{*} \mathsf{p}_{i_n})$$

$$\vdots$$

the number of distinct derivations $S \xrightarrow{*} \mathsf{p}_r$, $r \in \mathbb{N}$ is finite, for all the terminal strings are made of symbols of a $M_p \cup E_p \cup T^{\mu}$ and of "length" less than $\Lambda(S \xrightarrow{*} \mathsf{p})$, and the *μ-ELECTRE* grammar is unambiguous.

In the end of this section, we define an appropriate measure on derivations, so that it is monotonic and decreases with the rewriting and of course bounds up the number of lexical units of the terminal string of a derivation.

5.2. *The proof*

5.2.1. *Preliminary definitions*

Definition 5.1. In the *μ-ELECTRE* grammar, we define the length $\lambda(x)$ of terminal symbols x by

- $\lambda(x) = 1\ \forall x \in T^{\mu}$,
- $\lambda(x) = 1\ \forall x =$ module_identifier or event_identifier.

Definition 5.2. The synthesized attribute Λ is defined in the *μ-ELECTRE* grammar as follows: for all the production rules of the *μ-ELECTRE* grammar except rule 1, the value of Λ on the left-hand side nonterminal symbol is the sum of

- the value of Λ on nonterminal symbols of the right-hand side,
- and the value of λ (length) on the terminal symbols of the right-hand side.

For the null derivation, $\Lambda(nil) = 0$.
On production rule 5, $K ::= E : M$, this yields

$$\Lambda(K) = \Lambda(E) + \lambda(:) + \Lambda(M).$$

On production rule 1, $S ::= [P_1] * ([P_2])$, the value of Λ on S is given by

$$\Lambda(S) = \max(\Lambda(P_1) + \Lambda(P_2), 2.\Lambda(P_2)) + \underbrace{\lambda(*) + \lambda(() + \lambda()) + 2.\lambda([) + 2.\lambda(])}_{7}.$$

The attribute Λ defined in this way will "decrease" in the rewriting of a derivation.

Definition 5.3. In the set of derivations D starting from nonterminal symbols of the *μ-ELECTRE* grammar, we extend attribute Λ (which becomes polymorphic) into a mapping $\Lambda : D \to \mathbb{N}$:

$$\forall d \neq nil \in D, \quad d = X \overset{*}{\to} \tilde{x}, \qquad \Lambda(d) = \Lambda(X)$$

$$d = nil, \qquad \Lambda(d) = 0$$

Applied on derivations of type $S \overset{*}{\to} \mathsf{p}$ (which derives the axiom into a *μ-ELECTRE* program), $\Lambda(S \overset{*}{\to} \mathsf{p})$ bounds up the number of lexical units written in program p.

5.2.2. *A partial ordering on D*

Clearly, what we intend to show on a rewriting of derivations

$$\{e\} \vdash (S \overset{*}{\to} \mathsf{p}) \triangleright (S \overset{*}{\to} \mathsf{p}')$$

is

$$\Lambda(S \xrightarrow{*} \mathsf{p}) \geqslant \Lambda(S \xrightarrow{*} \mathsf{p}')$$

This proof shall be achieved by *induction* (standard induction) on the length of the derivations; since there exists a recursive production rule in the *μ-ELECTRE* grammar, we cannot easily prove this decreasing property: actually on a derivation starting with production rule 2 ($P_1 ::= C \|_c P_2$), we cannot easily state on the rewriting rule (2.4),

$$\frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (C' \xrightarrow{*} \tilde{c}') \ \wedge \ \{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright (P_2' \xrightarrow{*} \tilde{p}_2')}{\{e\} \vdash (P_1 \to C \|_c P_2 \xrightarrow{*} \tilde{c} \|_c \tilde{p}_1) \triangleright (P_1' \to C' \|_c P_2' \xrightarrow{*} \tilde{c}' \|_c \tilde{p}_2')}$$

that

$$\Lambda(P_1 \to C \|_c P_2 \xrightarrow{*} \tilde{c} \|_c \tilde{p}_1) \geqslant \Lambda(P_1' \to C' \|_c P_2' \xrightarrow{*} \tilde{c}' \|_c \tilde{p}_2')$$

without the assumptions

$$\Lambda(C \xrightarrow{*} \tilde{c}) \geqslant \Lambda(C' \xrightarrow{*} \tilde{c}')$$

and

$$\Lambda(P_2 \xrightarrow{*} \tilde{p}_2) \geqslant \Lambda(P_2' \xrightarrow{*} \tilde{p}_2')$$

We are trying to prove the second one: such a problem can be handled easily by introducing a structure on the set of derivations D.

Definition 5.4. The binary relation $\sqsubseteq$ is defined on D by

$$d_1, d_2 \in D, \quad d_2 \sqsubseteq d_1 \Leftrightarrow \begin{cases} d_2 = Y \xrightarrow{*} y \\ d_1 = X \xrightarrow{*} \alpha Y \beta \xrightarrow{*} xyz \\ \alpha \xrightarrow{*} x \text{ and } \beta \xrightarrow{*} z \end{cases}$$

where

- X and Y are nonterminal symbols of the *μ-ELECTRE* grammar,
- x, y, and z, are strings made only of terminal symbols,
- α and β are sequences of either terminal or nonterminal symbols.

$\sqsubseteq$ is a partial ordering on D; roughly speaking, $d_2 \sqsubseteq d_1$ if d_2 is a subderivation of d_1.

Moreover, the relation $\sqsubseteq$ on D is *well-founded* in the sense that every decreasing sequence of derivations in D is stationary (*nil* is the null derivation in D).

Consequently, we can apply the *structural induction rule* [8, 24] to $(D, \sqsubseteq)$: if P is a property on D, and if

$$(\forall d \in D, (P(d'), \forall d' \sqsubset d)) \Rightarrow P(d)$$

then $\forall d \in D$, $P(d)$.

($\sqsubset$ is the strict ordering induced by $\sqsubseteq$).

We shall see the use of this rule in the next two sections.

5.2.3. The decreasing property

We now consider proving property P which will enable us to conclude that the transition system associated with a *μ-ELECTRE* program is finite. The expression of P on D is

$$(P) \qquad d \in D, \quad P(d) \equiv (\forall e, \{e\} \vdash d \triangleright d', \Lambda(d) \geqslant \Lambda(d')).$$

Let $d \in D$ be a derivation,

$$d = X \to y_0 Z_0 \dots y_n Z_n y_{n+1} \overset{*}{\to} y_0 \tilde{z}_0 \dots y_n \tilde{z}_n y_{n+1}.$$

As we can apply the structural induction rule to $(D, \sqsubseteq)$, the problem reduces to: assuming P holds on each subderivation $d_i = Z_i \to \tilde{z}_i$, prove that P holds on d.

Since there are more than one rewriting rule associated with production rule $X ::= y_0 Z_0 \dots y_n Z_n y_{n+1}$, we must investigate all the cases. Actually, what must be proved is that whatever the rewriting rule associated with production rule $X ::= y_0 Z_0 \dots y_n Z_n y_{n+1}$, the following property holds:

$$(\forall i \in [0, n], P(d_i)) \Rightarrow P(d).$$

Therefore, to confirm P holds on D, we only need to establish the following proposition.

Proposition 5.1. *For each production rule* $X ::= y_0 Z_0 \dots y_n Z_n y_{n+1}$ *of the μ-ELECTRE grammar, and for each conditional rewriting rule*

$$\forall e \quad \frac{\{e\} \vdash (Z_0 \overset{*}{\to} \tilde{z}_0) \triangleright (Y_0' \overset{*}{\to} \tilde{y}_0') \wedge \dots \wedge \{e\} \vdash (Z_n \overset{*}{\to} \tilde{Z}_n) \triangleright (Y_n' \overset{*}{\to} \tilde{y}_n')}{\{e\} \vdash (X \overset{*}{\to} \tilde{x}) \triangleright (T' \overset{*}{\to} \tilde{t}')}$$

if

$$\forall i \in [0, n] \quad \Lambda(Z_i \overset{*}{\to} \tilde{z}_i) \geqslant \Lambda(Y_i' \overset{*}{\to} \tilde{y}_i')$$

then

$$\Lambda(X \overset{*}{\to} \tilde{x}) \geqslant \Lambda(T' \overset{*}{\to} \tilde{t}')$$

We shall prove that Proposition 5.1 holds for a subset of the production rules of the *μ-ELECTRE* grammar.

We begin by giving the proof of Propositon 5.1 on production rules that derive only terminal symbols, rules 6 and 7. Then, we show in nontrivial cases how the proof can be achieved on production rules 4 and 1 since they revealed to be the more intricate ones as they, respectively, deal with the parallel and repetitive structures. For the remaining rules, (2, 3, 5), the calculus can be easily achieved.

Proof of Proposition 5.1. *On production rule* 6. The two conditional rewriting rules for $E ::= \mathrm{e}$ under environment $\{e'\}$ are (see the appendix)

$$\{e'\} \vdash (E \to \mathrm{e}) \triangleright \mathrm{nil} \quad \text{if } e = e'$$

$$\{e'\} \vdash (E \to \mathrm{e}) \triangleright (E' \to \mathrm{e}) \quad \text{if } e \neq e'$$

Thus, proving that Proposition 5.1 holds on $E ::= \mathrm{e}$ is straightforward: in the two possible rewriting rules, $\Lambda(E \to \mathrm{e}) \geqslant \Lambda(E' \to \mathrm{e})$ (since $\Lambda(nil) = 0$ and $\Lambda(E') = \Lambda(E)$). The same holds for rule 7.

Proof. *On production rule 4*

Cases 1 and 2: On rewriting rules (4.1) and (4.2), the proof is straightforward and not developed here.

Case 3: We consider here the conditional rewriting rule (4.3),

$$\frac{\{e\} \vdash (K_1 \overset{*}{\to} \tilde{k}_1) \triangleright (K_1' \overset{*}{\to} \tilde{k}_1') \ \wedge \ \{e\} \vdash (K_2 \overset{*}{\to} \tilde{k}_2) \triangleright (K_2' \overset{*}{\to} \tilde{k}_2')}{\{e\} \vdash (I \to \{K_1 \|_e K_2\} \overset{*}{\to} \{\tilde{k}_1 \|_e \tilde{k}_2\}) \triangleright (I' \to \{K_1' \|_e K_2'\} \overset{*}{\to} (\tilde{k}_1' \|_e \tilde{k}_2'\})}$$

The aim here is to conclude that $\Lambda(I \overset{*}{\to} \tilde{i}) \geqslant \Lambda(I' \overset{*}{\to} \tilde{i}')$ under the structural induction assumptions

$$\Lambda(K_1 \overset{*}{\to} \tilde{k}_1) \geqslant \Lambda(K_1' \overset{*}{\to} \tilde{k}_1')$$

and

$$\Lambda(K_2 \overset{*}{\to} \tilde{k}_2) \geqslant \Lambda(K_2' \overset{*}{\to} \tilde{k}_2')$$

The calculus of Λ (the attribute form) in the derivation

$$I' \to \{K_1' \|_e K_2'\} \overset{*}{\to} \{\tilde{k}_1' \|_e \tilde{k}_2'\}$$

yields

$$\begin{aligned} \Lambda(I') &= \lambda(\{) + \Lambda(K_1') + \lambda(\|_e) + \Lambda(K_2') + \lambda(\}) \\ &\leqslant \underbrace{\lambda(\{) + \Lambda(K_1) + \lambda(\|_e) + \Lambda(K_2) + \lambda(\})}_{\Lambda(I)} \end{aligned}$$

which is the expected result.

Case 4: We now consider the conditional rewriting rule (4.4):

$$\frac{\{e\} \vdash (K_1 \overset{*}{\to} \tilde{k}_1) \triangleright (C' \overset{*}{\to} \tilde{c}') \ \wedge \ \{e\} \vdash (K_2 \overset{*}{\to} \tilde{k}_2) \triangleright (K_2' \overset{*}{\to} \tilde{k}_2')}{\{e\} \vdash (I \to \{K_1 \|_e K_2\} \overset{*}{\to} \{\tilde{k}_1 \|_e \tilde{k}_2\}) \triangleright (P' \to C'' \|_c P_1' \overset{*}{\to} \tilde{c}' \|_c \mathbf{1}/\tilde{k}_2')}$$

The proof of Proposition 5.1 for this conditional rewriting rule consists in proving $\Lambda(I \overset{*}{\to} \tilde{i}) \geqslant \Lambda(P' \overset{*}{\to} \tilde{p}')$ under the structural induction assumptions

$$\Lambda(K_1 \overset{*}{\to} \tilde{k}_1) \geqslant \Lambda(C' \overset{*}{\to} \tilde{c}')$$

and

$$\Lambda(K_2 \overset{*}{\to} \tilde{k}_2) \geqslant \Lambda(K_2' \overset{*}{\to} \tilde{k}_2')$$

If we detail the rewriting of the derivation $P' \to C' \|_c P'_1 \overset{*}{\to} \tilde{c}' \|_c \mathbf{1}/\tilde{k}'_2$ we obtain[10]

$$
\begin{aligned}
P' &\to C' \|_c P'_1 \\
&\overset{*}{\to} \tilde{c}' \|_c P'_2 \\
&\to \tilde{c}' \|_c C'_3 \\
&\to \tilde{c}' \|_c \mathbf{1}/I'_4 \\
&\to \tilde{c}' \|_c \mathbf{1}/K'_5 \\
&\overset{*}{\to} \tilde{c}' \|_c \mathbf{1}/\tilde{k}'_2
\end{aligned}
$$

The two derivations $K'_2 \overset{*}{\to} \tilde{k}'_2$ and $K'_5 \overset{*}{\to} \tilde{k}'_2$ are equal since the *μ-ELECTRE* grammar is unambiguous; consequently,

$$\Lambda(K'_2 \overset{*}{\to} \tilde{k}'_2) = \Lambda(K'_5 \overset{*}{\to} \tilde{k}'_2)$$

The calculus of the attributable Λ in the derivation starting from P' runs as follows:

$$
\begin{aligned}
\Lambda(P') &= \Lambda(C') + \lambda(\|_c) + \Lambda(P'_1) \\
&= \Lambda(C') + \lambda(\|_c) + \Lambda(C'_3) \\
&= \Lambda(C') + \lambda(\|_c) + \lambda(\mathbf{1}) + \lambda(/) + \Lambda(I'_4) \\
&= \Lambda(C') + \lambda(\|_c) + \lambda(\mathbf{1}) + \lambda(/) + \Lambda(K'_5) \\
&= \Lambda(C') + \lambda(\|_c) + \lambda(\mathbf{1}) + \lambda(/) + \Lambda(K'_2) \\
&\leqslant \Lambda(K_1) + \lambda(\|_c) + \lambda(\mathbf{1}) + \lambda(/) + \Lambda(K_2)
\end{aligned}
$$

Since

$$\Lambda(I) = \lambda(\{) + \Lambda(K_1) + \lambda(\|_e) + \Lambda(K_2) + \lambda(\})$$

it suffices to show that

$$\lambda(\|_c) + \lambda(\mathbf{1}) + \lambda(/) \leqslant \lambda(\{) + \lambda(\|_e) + \lambda(\})$$

which is true as both sides of this inequality equal 3.

Cases 5 and 6: The same calculus can be achieved for the rewriting rules (4.5) and (4.6), which proves that Proposition 5.1 holds for this production rule.

Proof. *On production rule 1*

Case 1: For conditional rewriting rule (1.1),

$$\frac{\{e\} \vdash (P_1 \overset{*}{\to} \tilde{p}_1) \triangleright (P'_1 \overset{*}{\to} \tilde{p}'_1)}{\{e\} \vdash (S \to [P_1] * ([P_2]) \overset{*}{\to} [\tilde{p}_1] * ([\tilde{p}_2])) \triangleright (S \to [P'_1] * ([P'_2]) \overset{*}{\to} [\tilde{p}'_1] * ([\tilde{p}_2]))}$$

[10] This is the only possible derivation for the *μ-ELECTRE* grammar is unambiguous.

we only need to show that $\Lambda(S \xrightarrow{*} \tilde{s}) \geqslant \Lambda(S' \xrightarrow{*} \tilde{s}')$ under the assumption

$$\Lambda(P_1 \xrightarrow{*} \tilde{p}_1) \geqslant \Lambda(P_1' \xrightarrow{*} \tilde{p}_1')$$

Clearly, $P_2' \xrightarrow{*} \tilde{p}_2 = P_2 \xrightarrow{*} \tilde{p}_2$, and therefore $\Lambda(P_2' \xrightarrow{*} \tilde{p}_2') = \Lambda(P_2 \xrightarrow{*} \tilde{p}_2)$. The calculus of attribute Λ in the derivation $S' \to [P_1']*([P_2']) \xrightarrow{*} [\tilde{p}_1']*([\tilde{p}_2])$ runs as follows:

$$\begin{aligned}\Lambda(S') &= \lambda([) + \Lambda(P_1') + \lambda(]) + \lambda(*) + \lambda(() + \lambda([) + \Lambda(P_2') + \lambda(]) + \lambda()) \\ &\leqslant \Lambda(P_1) + \Lambda(P_2) + 7 \\ &\leqslant \underbrace{\max(\Lambda(P_1) + \Lambda(P_2), 2.\Lambda(P_2)) + 7}_{\Lambda(S) \text{ (see Definition 5.2)}}\end{aligned}$$

which is the expected inequality.

Case 2: The second conditional rewriting rule (1.2) is

$$\frac{\{e\} \vdash (P_1 \xrightarrow{*} \tilde{p}_1) \triangleright nil}{\{e\} \vdash (S \to [P_1]*([P_2]) \xrightarrow{*} [\tilde{p}_1]*([\tilde{p}_2])) \triangleright (S \to [P_1']*([P_2']) \xrightarrow{*} [\tilde{p}_2]*([\tilde{p}_2]))}$$

The *μ-ELECTRE* grammar is unambiguous; this entails

$$P_2 \xrightarrow{*} \tilde{p}_2 = P_1' \xrightarrow{*} \tilde{p}_2 = P_2' \xrightarrow{*} \tilde{p}_2$$

and consequently

$$\Lambda(P_2 \xrightarrow{*} \tilde{p}_2) = \Lambda(P_1' \xrightarrow{*} \tilde{p}_2) = \Lambda(P_2' \xrightarrow{*} \tilde{p}_2)$$

The calculus of Λ in S' runs as follows:

$$\begin{aligned}\Lambda(S') &= \lambda([) + \Lambda(P_1') + \lambda(]) + \lambda(() + \lambda([) + \Lambda(P_2') + \lambda(]) + \lambda()) \\ &= \Lambda(P_2) + \Lambda(P_2) + 7 \\ &\leqslant \underbrace{\max(\Lambda(P_1) + \Lambda(P_2), 2.\Lambda(P_2)) + 7}_{\Lambda(S)}\end{aligned}$$

which is once again the expected inequality.

This accounts for the particular formula defined to calculate the value of attribute Λ in this rule (Definition 5.2). Had we defined the calculus of Λ in $S ::= [P_1]*([P_2])$ as

$$\Lambda(S) = \lambda([) + \Lambda(P_1) + \lambda(]) + \lambda(() + \lambda([) + \Lambda(P_2) + \lambda(]) + \lambda())$$

we could not have concluded anything about the decreasing of Λ in the conditional rewriting rule (1.2).

Similar calculus can be carried out for all the conditional rewriting rules of the *μ-ELECTRE* grammar: this concludes all the cases and hence the proof of Proposition 5.1. □

We are now able to state the following theorem.

Theorem 5.1. *For each derivation* $d \in D$,

$$\forall e \quad \{e\} \vdash d \triangleright d', \qquad \Lambda(d) \geqslant \Lambda(d')$$

5.2.4. The finiteness theorem

We have pointed out in Section 5.1 that given

- a *μ-ELECTRE* program p,
- the two sets M_p and E_p which, respectively, are the set of module identifiers and the set of event identifiers written in p,

under whatever sequence of rewriting of derivations starting from program p

$$p \overset{e_{i_1}}{\mapsto} p_1 \overset{e_{i_2}}{\mapsto} p_2 \mapsto \cdots \overset{e_{i_n}}{\mapsto} p_n \mapsto \cdots$$

all the lexical units of programs p_i belong to the finite set

$$M_p \cup E_p \cup T^{\mu}$$

Applied to program p_i, Theorem 5.1 establishes that

$$\forall e, \forall i \quad \{e\} \vdash (S \overset{*}{\rightarrow} p_i) \triangleright (S \overset{*}{\rightarrow} p_{i+1}), \qquad \Lambda(S \overset{*}{\rightarrow} p_i) \geqslant \Lambda(S \overset{*}{\rightarrow} p_{i+1})$$

and consequently,

$$\forall e_{i_1} e_{i_2} \ldots e_{i_n} \quad \Lambda(S \overset{*}{\rightarrow} p) \geqslant \Lambda(S \overset{*}{\rightarrow} p_n)$$

As $\Lambda(S \overset{*}{\rightarrow} p_n)$ is greater than the number of lexical units written in program p_n,[11] all the programs obtained from p by a sequence of rewriting of derivations belong to the finite set

$$(M \cup E \cup T^{\mu})^{\Lambda(S \overset{*}{\rightarrow} p)}$$

This entails the *finiteness theorem.*

Finiteness Theorem. *The set of programs obtained under any sequence of rewriting of derivations from a program* p *is finite.*

5.3. About the finiteness of the transition system

Owing to the Finiteness Theorem, one can conclude that the transition system associated with a *μ-ELECTRE* program is finite: a state of this transition system corresponds to a program obtained by rewriting of derivations; a transition stands for a reaction to an event.

Practically, as it was stated in Section 3, a state of the transition system is composed of two components, the program which is left to be executed and the states of the

[11] This can also be stated by structural induction.

components (modules and events). This leads to a model of execution for *ELECTRE* programs, with occurrences of events stored at most once. This model cannot deal with multiple storage events.

6. FIFO-transition system to handle multiple storage events

6.1. What are multiple storage events?

The *ELECTRE* language allows us to use all occurrences of events which are to be treated: an occurrence of an event may be stored in order to be processed later. In fact, if such an occurrence cannot be processed at the moment it occurs, then we want it to be processed *as soon as possible*. Besides, it is necessary to treat these events in the order they have occurred. The semantics of multiple storage events is not given by the operational semantics. The main reason is that considering an unbound number of occurrences of events stored in the *state* $\langle P, C \rangle$ may bring about an infinite number of different states (differing only in the list of multiple storage events arrived so far). The idea is to drop all these occurrences out of the state $\langle P, C \rangle$ and treat them differently. These remarks lead to the use of a FIFO-list to handle both the storage and the ordering of occurrences of multiple storage events.

6.2. Transitions associated with multiple storage events

The operational semantics, though not describing how the occurrences of multiple storage events are handled, can determine *when* an event is to be stored (this can be deduced from the state $\langle P, C \rangle$). In this section, we use a model we call FIFO-transition system, derived from FIFO-automata [26, 11] and pushdown automata [3, 25]. A FIFO-transition system is a transition system with a memory capability managed as a FIFO-list, and in this model we use the following notation:

- events in the *ELECTRE* program we consider belong to a finite set $E = \{e_1, e_2, \ldots, e_n\}$, and multiple storage events are given by a subset $E_M \subseteq E$, where $M \subseteq [1, n]$,
- $\delta(q, e, A) = [q', B]$, where $e \in E$ and $A, B \in E_M$; means that in state q, processing e when A is the first item of the FIFO-list takes us in state q', A is removed from the FIFO-list and B is added to the FIFO-list.

More precisely, since we do not allow simultaneous processing of events (asynchronous assumption), transitions will be of the following forms:

- $\delta(q, e, \lambda) = [q', B]$ means that whatever is the first item of the FIFO-list (λ), processing e takes us in state q' and B is added to the FIFO-list,
- $\delta(q, \lambda, e) = [q', B]$ means that whatever is to be processed (λ), if e is the first item of the FIFO-list, then go in state q', remove e from the FIFO-list and add B to the FIFO-list.

Consequently, transitions of the form $\delta(q, e, A) = [q', B]$, where both e and $A \neq \lambda$ are ruled out by the asynchronous assumption.

If we want to add *nothing* to the FIFO-list, we use the ε symbol instead of B in the above statements.

Now suppose the occurrence of event e is to be stored if it happens in state q (q stands for a couple $\langle P, C \rangle$). Then we simply create a transition of the form

$$\delta(q, e, \lambda) = [q, e]$$

The effect of this transition is to add e to the FIFO-list with no change in the state of the FIFO-transition system.

Symmetrically, transitions are created to perform the *batch processing* of events which have been stored in the FIFO-list. When such a transition is created is quite easy to understand: every time a transition of the form $\delta(q, e, \lambda) = [q', \varepsilon]$ exists, corresponding to the *immediate processing* of an occurrence of *e if it occurred in state q*, we allow the batch processing of an occurrence of the same event *e in state q if it has occurred and is the first item of the FIFO-list.* Every transition of the form $\delta(q, e, \lambda) = [q', \varepsilon]$ involving a multiple storage event entails the creation of a FIFO-transition which has the same effect and corresponds to the batch processing of the stored occurrence of the event.

If in state q a transition of the form

$$\delta(q, e, \lambda) = [q', \varepsilon]$$

exists, then we create a corresponding transition for batch processing

$$\delta(q, \lambda, e) = [q', \varepsilon].$$

An insidious thing results from this building method: the FIFO-transition system seems to become nondeterministic. This is due to the *semantics* of the notation used for FIFO-automata, which permits the two transitions t and $\tilde{t}$ with

$$t\colon\ \delta(q, e, \lambda) = \{q', \varepsilon] \quad \text{and} \quad \tilde{t}\colon\ \delta(q, \lambda, e) = [q', \varepsilon]$$

to be "fired" simultaneously (an occurrence of e can be immediately processed *and* if an occurrence of e is the first item of the FIFO-list, it can be batch processed too).

This nondeterminism is ruled out by giving priority to $\tilde{t}$. It corresponds to taking into account *as soon as possible* the stored occurrence of an event, before any immediate processing of any occurrence of event.

From all this, it results that any *ELECTRE* program can be modellized by a FIFO-transition system; we denote $\mathscr{F}$ in which we give priority to the batch processing of FIFO-list items whenever it is possible.[12]

[12] This storage management prevents the memorized events processing from being infinitely delayed.

This last model is a comprehensive model of execution for *ELECTRE* programs.

7. Conclusion

We built up a model of an *ELECTRE* program using a couple ⟨*program*, states of the components⟩.

The asynchronous approach has benefitted greatly from the formal results of the synchronous approach (determinism, proofs of programs and so on).

In fact, due to the variety of properties available in the events, we have chosen a FIFO-transition system to describe the execution of the program. We proved that the transition system performed during the compiling phase is a finite-state machine. This result is meaningful since the *ELECTRE* asynchronous language deals with multiple memorized occurrences of the events.

A first "toy" implementation of a compiler has been made using the UNIX utilities *LEX* and *YACC*. Programs modelling well-known problems like *readers and writers*, *the mouse handler* have been written and compiled to obtain a FIFO-transition system. The execution of the programs corresponding to paths in the transition system can be simulated or executed with the tools SILEX and EXILE which were designed specifically for *ELECTRE* programs. A second version of the *ELECTRE* compiler is under development using the CENTAUR system [6].

Compiling an *ELECTRE* program to a FIFO-transition system has many advantages, such as determinism, efficiency and, moreover, temporal properties can be checked in this transition system.

Indeed, the FIFO-transition system can be "converted" under certain assumptions to standard[13] finite-state transition system. Roughly, two simple automata[14] and a synchronization constraint [2] are extracted from the FIFO-transition system. Then the MEC [1] software which is a tool for analysing and constructing automata performs the synchronization of the systems with regard to the synchronization constraint. Various properties can be checked using MEC: for instance, we have shown that the program used to model the *readers–writers* problem actually has no deadlocks, provides mutual exclusion between the readers and the writers and that there is no starvation.

Other directions are presently investigated (e.g. comparisons between the asynchronous and synchronous approaches, distributed implementation of the *ELECTRE* language, and so on) in order to develop the *ELECTRE* environment as a powerful tool for describing real-time applications.

[13] Standard means with no memory capabilities.

[14] One of which is the FIFO-list.

Appendix: Conditional rewriting rules for *μ-ELECTRE*

Rule 1

$S ::= [P_1] * ([P_2])$

$$(1.1)\quad \frac{\{e\} \vdash (P_1 \xrightarrow{*} \tilde{p}_1) \triangleright (P_1' \xrightarrow{*} \tilde{p}_1')}{\{e\} \vdash (S \to [P_1] * ([P_2]) \xrightarrow{*} [\tilde{p}_1] * ([\tilde{p}_2])) \triangleright (S \to [P_1'] * ([P_2']) \xrightarrow{*} [\tilde{p}_1'] * ([\tilde{p}_2]))}$$

$$(1.2)\quad \frac{\{e\} \vdash (P_1 \xrightarrow{*} \tilde{p}_1) \triangleright nil}{\{e\} \vdash (S \to [P_1] * ([P_2]) \xrightarrow{*} [\tilde{p}_1] * ([\tilde{p}_2])) \triangleright (S \to [P_1'] * ([P_2']) \xrightarrow{*} [\tilde{p}_2] * ([\tilde{p}_2]))}$$

Rule 2

$P ::= C$

$$(2.1)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (C' \xrightarrow{*} \tilde{c}')}{\{e\} \vdash (P \to C \xrightarrow{*} \tilde{c}) \triangleright (P' \to C' \xrightarrow{*} \tilde{c}')}$$

$$(2.2)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright nil}{\{e\} \vdash (P \to C \xrightarrow{*} \tilde{c}) \triangleright (nil)}$$

$$(2.3)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (P' \xrightarrow{*} \tilde{p}')}{\{e\} \vdash (P \to C \xrightarrow{*} \tilde{c}) \triangleright (P' \xrightarrow{*} \tilde{p}')}$$

$P_1 ::= C \parallel_c P_2$

$$(2.4)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (C' \xrightarrow{*} \tilde{c}') \ \wedge\ \{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright (P_2' \xrightarrow{*} \tilde{p}_2')}{\{e\} \vdash (P_1 \to C \parallel_c P_2 \xrightarrow{*} \tilde{c} \parallel_c \tilde{p}_1) \triangleright (P_1' \to C' \parallel_c P_2' \xrightarrow{*} \tilde{c}' \parallel_c \tilde{p}_2')}$$

$$(2.5)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright nil \ \wedge\ \{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright (P_2' \xrightarrow{*} \tilde{p}_2')}{\{e\} \vdash (P_1 \to C \parallel_c P_2 \xrightarrow{*} \tilde{c} \parallel_c \tilde{p}_1) \triangleright (P' \xrightarrow{*} \tilde{p}_2' \parallel_c \tilde{p}_2')}$$

$$(2.6)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (P_1' \xrightarrow{*} \tilde{p}_1') \ \wedge\ \{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright (P_2' \xrightarrow{*} \tilde{p}_2')}{\{e\} \vdash (P_1 \to C \parallel_c P_2 \xrightarrow{*} \tilde{c} \parallel_c \tilde{p}_1) \triangleright (P' \xrightarrow{*} \tilde{p}_1' \parallel_c \tilde{p}_2')}$$

$$(2.7)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (C' \xrightarrow{*} \tilde{c}') \ \wedge\ \{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright nil}{\{e\} \vdash (P_1 \to C \parallel_c P_2 \xrightarrow{*} \tilde{c} \parallel_c \tilde{p}_1) \triangleright (P' \to C' \xrightarrow{*} \tilde{c}')}$$

$$(2.8)\quad \frac{\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright (P_1' \xrightarrow{*} \tilde{p}_1') \ \wedge\ \{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright nil}{\{e\} \vdash (P_1 \to C \parallel_c P_2 \xrightarrow{*} \tilde{c} \parallel_c \tilde{p}_1) \triangleright (P' \xrightarrow{*} \tilde{p}_1')}$$

The cases where both $\{e\} \vdash (C \xrightarrow{*} \tilde{c}) \triangleright nil$ and $\{e\} \vdash (P_2 \xrightarrow{*} \tilde{p}_2) \triangleright nil$ cannot arise since simultaneity is not allowed.

Rule 3

$C ::= M$

$$(3.1)\quad \frac{\{e\} \vdash (M \xrightarrow{*} \tilde{m}) \triangleright (M' \xrightarrow{*} \tilde{m}')}{\{e\} \vdash (C \to M \xrightarrow{*} \tilde{m}) \triangleright (C' \to M' \xrightarrow{*} \tilde{m}')}$$

$$(3.2)\quad \frac{\{e\} \vdash (M \xrightarrow{*} \tilde{m}) \triangleright nil}{\{e\} \vdash (C \to M \xrightarrow{*} \tilde{m}) \triangleright nil}$$

$C ::= \mathbf{1}/I$

$$(3.3)\quad \frac{\{e\} \vdash (I \xrightarrow{*} \tilde{i}) \triangleright (I' \xrightarrow{*} \tilde{i}')}{\{e\} \vdash (C \to \mathbf{1}/I \xrightarrow{*} \mathbf{1}/\tilde{i}) \triangleright (C' \to \mathbf{1}/I' \xrightarrow{*} \mathbf{1}/\tilde{i}')}$$

$$(3.4)\quad \frac{\{e\} \vdash (I \xrightarrow{*} \tilde{i}) \triangleright (P' \xrightarrow{*} \tilde{p}')}{\{e\} \vdash (C \to \mathbf{1}/I \xrightarrow{*} \mathbf{1}/\tilde{i}) \triangleright (P' \xrightarrow{*} \tilde{p}')}$$

$$(3.5)\quad \frac{\{e\} \vdash (I \xrightarrow{*} \tilde{i}) \triangleright (C' \xrightarrow{*} \tilde{c}')}{\{e\} \vdash (C \to \mathbf{1}/I \xrightarrow{*} \mathbf{1}/\tilde{i}) \triangleright (C' \xrightarrow{*} \tilde{c}')}$$

$$(3.6)\quad \frac{\{e\} \vdash (I \xrightarrow{*} \tilde{i}) \triangleright nil}{\{e\} \vdash (C \to \mathbf{1}/I \xrightarrow{*} \mathbf{1}/\tilde{i}) \triangleright nil}$$

Rule 4

$I ::= K$

$$(4.1)\quad \frac{\{e\} \vdash (K \xrightarrow{*} \tilde{k}) \triangleright (K' \xrightarrow{*} \tilde{k}')}{\{e\} \vdash (I \to K \xrightarrow{*} \tilde{k}) \triangleright (I' \to K' \xrightarrow{*} \tilde{k}')}$$

$$(4.2)\quad \frac{\{e\} \vdash (K \xrightarrow{*} \tilde{k}) \triangleright (C' \xrightarrow{*} \tilde{c}')}{\{e\} \vdash (I \to K \xrightarrow{*} \tilde{k}) \triangleright (C' \xrightarrow{*} \tilde{c}')}$$

$I ::= \{K_1 \|_e K_2\}$

$$(4.3)\quad \frac{\{e\} \vdash (K_1 \xrightarrow{*} \tilde{k}_1) \triangleright (K_1' \xrightarrow{*} \tilde{k}_1') \ \wedge\ \{e\} \vdash (K_2 \xrightarrow{*} \tilde{k}_2) \triangleright (K_2' \xrightarrow{*} \tilde{k}_2')}{\{e\} \vdash (I \to \{K_1 \|_e K_2\} \xrightarrow{*} \{\tilde{k}_1 \|_e \tilde{k}_2\}) \triangleright (I' \to \{K_1' \|_e K_2'\} \xrightarrow{*} \{\tilde{k}_1' \|_e \tilde{k}_2'\})}$$

$$(4.4)\quad \frac{\{e\} \vdash (K_1 \xrightarrow{*} \tilde{k}_1) \triangleright (C' \xrightarrow{*} \tilde{c}') \ \wedge\ \{e\} \vdash (K_2 \xrightarrow{*} \tilde{k}_2) \triangleright (K_2' \xrightarrow{*} \tilde{k}_2')}{\{e\} \vdash (I \to \{K_1 \|_e K_2\} \xrightarrow{*} \{\tilde{k}_1 \|_e \tilde{k}_2\}) \triangleright (P' \to C' \|_c P_1' \xrightarrow{*} \tilde{c}' \|_c \mathbf{1}/\tilde{k}_2')}$$

$$(4.5)\quad \frac{\{e\} \vdash (K_1 \xrightarrow{*} \tilde{k}_1) \triangleright (K_1' \xrightarrow{*} \tilde{k}_1') \ \wedge\ \{e\} \vdash (K_2 \xrightarrow{*} \tilde{k}_2) \triangleright (C' \xrightarrow{*} \tilde{c}')}{\{e\} \vdash (I \to \{K_1 \|_e K_2\} \xrightarrow{*} \{\tilde{k}_1 \|_e \tilde{k}_2\}) \triangleright (P' \to C' \|_c P_1' \xrightarrow{*} \tilde{k}_1' \|_c \mathbf{1}/\tilde{c}')}$$

$$(4.6)\quad \frac{\{e\} \vdash (K_1 \xrightarrow{*} \tilde{k}_1) \triangleright (C_1' \xrightarrow{*} \tilde{c}_1') \ \wedge\ \{e\} \vdash (K_2 \xrightarrow{*} \tilde{k}_2) \triangleright (C_2' \xrightarrow{*} \tilde{c}_2')}{\{e\} \vdash (I \to \{K_1 \|_e K_2\} \xrightarrow{*} \{\tilde{k}_1 \|_e \tilde{k}_2\}) \triangleright (P' \to C_1' \|_c P_1' \xrightarrow{*} \tilde{c}_1' \|_c \tilde{c}_2')}$$

Rule 5

$K ::= E : M$

$$(5.1)\quad \frac{\{e\} \vdash (E \to \tilde{e}) \rhd (E' \to \tilde{e}') \ \wedge\ \{e\} \vdash (M \to \tilde{m}) \rhd (M' \to \tilde{m}')}{\{e\} \vdash (K \to E : M \xrightarrow{*} \tilde{e} : \tilde{m}) \rhd (K' \to E' : M' \xrightarrow{*} \tilde{e}' : \tilde{m}')}$$

$$(5.2)\quad \frac{\{e\} \vdash (E \to \tilde{e}) \rhd nil \ \wedge\ \{e\} \vdash (M \to \tilde{m}) \rhd (M' \to \tilde{m}')}{\{e\} \vdash (K \to E : M \xrightarrow{*} \tilde{e} : \tilde{m}) \rhd (C' \to M' \xrightarrow{*} \tilde{m}')}$$

Note that the cases

$$\{e\} \vdash (E \to \tilde{e}) \rhd (\text{whatever}) \ \wedge\ \{e\} \vdash (M \to \tilde{m}) \rhd (nil)$$

may not arise for if module m is to be activated by event e it cannot be already active.

Rule 6

$E ::= \mathsf{e}$

$$\{e'\} \vdash (E \to \mathsf{e}) \rhd nil \quad \text{if } e = e'$$

$$\{e'\} \vdash (E \to \mathsf{e}) \rhd (E \to \mathsf{e}) \quad \text{if } e \neq e'$$

Rule 7

$M ::= \mathsf{M}$

$$\{e'\} \vdash (M \to \mathsf{A}) \rhd \mathrm{nil} \quad \text{if } e' = c_A$$

$$\{e'\} \vdash (M \to \mathsf{A}) \rhd (M \to \mathsf{A}) \quad \text{if } e' \neq c_A$$

References

[1] A. Arnold, Mec: a system for constructing and analysing transition systems, Tech. Report, Laboratoire Bordelais de Recherche en Informatique, Bordeaux, France, 1988.

[2] A. Arnold, Systèmes de transitions finis et sémantique des processus communicants, *Technique Sci. Inform.* **9**(3) (1990).

[3] J.-M. Autebert, *Langages Algébriques* (E.R.I. MASSON, 1987).

[4] G. Berry, P. Couronné and G. Gonthier, Programmation synchrone des systèmes réactifs: le langage ESTEREL, *Technque Sci. Inform.* **6** (1987) 305–316.

[5] G. Berry and R. Sethi, From regular expressions to deterministic automata, *Theoret. Comput. Sci.* **48** (1986) 117–126.

[6] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Khan, B. Lang and V. Pascual, Centaur: the system, in: *Proc. 3rd Ann. Symp. on Software Development Environments (SIGSOFT '88)*, Boston, USA, 1988.

[7] J.A. Brzozowski, Derivatives of regular expressions, *ACM* **4** (1964) 481–494.

[8] R.M.B. Burstall, Proving properties of programs by structural induction, *Comp. J.* **12** (1969) 41–48.

[9] R.H. Campbell and N. Haberman, *The Specification of Process Synchronization by Path Expressions*, Vol. 16 (Springer, Berlin, 1973) 89–102.

[10] P. Caspi, D. Pilaud, N. Halbwachs and J.A. Plaice, LUSTRE: a declarative language for programming synchronous systems, in: *Proc. 14th ACM Symp. on Principles of Programming Languages*, Munich, Germany, 1987.

[11] A. Cherubini, C. Citrini, S.C. Reghizzi and D. Mandroli, Quasi-real-time fifo automata, breadth-first grammars and their relations, *Theoret. Comput. Sci.* **85** (1991) 171–203.
[12] E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specification, *ACM Trans. Programming Languages Systems* **8** (1986) 244–263.
[13] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, Programmation et vérification des systèmes réactifs: le langage LUSTRE, *Technique Sci. Inform.* **10** (1991) 139–158.
[14] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Truaring and M. Trakhtenbrot, STATEMATE: a working environment for the development of complex reactive systems, *IEEE Trans. Software Eng.* **16** (1990) 403–414.
[15] D. Harel and A. Pnueli, On the development of reactive systems, in: K.R. Apt, ed., *Logics and Models of Concurrent Systems*, Vol. 13 NATO ASI Series (Springer, New York, 1985) 477–498.
[16] P.E. Lauer, P.R. Torrigiani and M.W. Shields, COSY – a system specification language based on paths and processes, *Acta Inform.* **12** (1979) 109–158.
[17] P. Le Guernic, A. Benveniste, P. Bournai and T. Gautier, SIGNAL: a data-flow oriented language for signal processing, *IEEE Trans. ASSP* **ASSP-34** (1986) 362–374.
[18] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92 (Springer, Berlin, 1980).
[19] J.S. Ostroff, *Automated Verification of Timed Transition Models*, Lecture Notes in Computer Science, Vol. 407 (Springer, Berlin, 1989) 247–256.
[20] J. Perraud, O. Roux and M. Huou, Operational semantics of a kernel of the language ELECTRE, *Theoret. Comput. Sci.* **97** (1992) 83–104.
[21] G.D. Plotkin, A structural approach to operational semantics, Tech. Report DAIMI FN-19, University of Aarhus, Denmark, 1981.
[22] P.J. Queille and J. Sifakis, Fairness and related properties in transition systems – a temporal logic to deal with fairness, *Acta Inform.* (1993) 195–220.
[23] O. Roux, D. Creusot, F. Cassez and J.-P. Elloy, Le langage réactif asynchrone ELECTRE, *Technique et Science Informatiques* **11**(5) (1992) 35–66.
[24] D. Scott, Logic and programming languages, *Commun. ACM* **20** (1977) 634–641.
[25] T. Sudkamp, *Languages and Machines* (Addison-Wesley, Reading, MA, 1988).
[26] B. Vauquelin and P.F. Zanettacci, Automates à files, *Theoret. Comput. Sci.* **11** (1980) 221–225.
[27] D. Vergamini, Verification on distributed systems: an experiment, Tech. Report 934, INRIA, Sophia-Antipolis, France, 1989.
[28] G. Winskel, Event structures semantics for ccs and related languages, in: *Proc. 9th Internat. Colloq. on Automata, Languages and Programming*, Aarhus (1982) 561–576.
[29] W. Zielonka, Notes on finite asynchronous automata, *Theoret. Inform. Appl.* **2** (1987) 99–135.