



## Execution Environment for ELECTRE Applications

Denis Creusot, Philippe Lemoine, Olivier Roux, Yvon Trinquet, Antonio Kung, Olivier Marbach, Carlos Serrano-Morales

### ► To cite this version:

Denis Creusot, Philippe Lemoine, Olivier Roux, Yvon Trinquet, Antonio Kung, et al.. Execution Environment for ELECTRE Applications. 3rd European Software Engineering Conference, ESEC '91, Oct 1991, Milan, Italy. pp.147-165, 10.1007/3540547428\_47 . hal-00453462

**HAL Id: hal-00453462**

**<https://hal.science/hal-00453462>**

Submitted on 18 Apr 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Execution environment for ELECTRE applications

**Denis Creusot, Philippe Lemoine, Olivier Roux, Yvon Trinquet**  
 LAN-ENSM, Ecole Nationale Supérieure de Mécanique, Equipe Temps Réel  
 1, rue de la Noë, F-44072 Nantes Cedex 03, France

**Antonio Kung, Olivier Marbach, Carlos Serrano-Morales**  
 TRIALOG Informatique  
 9, rue du Château d'Eau, F-75010 Paris, France

## Abstract

This paper describes an execution environment for reactive systems specified in ELECTRE. ELECTRE allows the specification of a real-time application's temporal behaviour in terms of sequential entities called modules, of events, of relations between modules like parallelism, and of relations between modules and events like preemption. ELECTRE is based on a design and implementation approach enforcing the separation of the sequential part of the application (i.e. module specification), the event part of the application (i.e. event specification), and the control part of the application (i.e. reaction to events). This separation is also reflected at the execution level which includes a control unit, a module unit and an event unit. The execution environment is supplemented by a display system, which can be used for simulation, debugging or monitoring purposes. The display system is a multiwindow facility based on two main types of representations : a structural representation and a temporal representation.

**Keywords:** Reactive systems, real-time parallel systems, visualisation and monitoring, execution system.

## 1 Introduction

This paper describes a specification, programming and execution environment based on ELECTRE [Elloy 85], acronym for Exécutif et Langage de Contrôle Temps-réel REparti (Language and Executive for Distributed Real-Time Control), a language allowing the description of the temporal behaviour of real-time control processes.

ELECTRE is one among several approaches intended to model a real-time system during its specification stage in order to be able to perform tasks such as analysis and formal checking of timing and event properties. Other approaches are specific "Real-Time Logics" [Jahanian 86, Ostroff 90], the use of event histories [Dixon 86, Faulk 88], the use

of data-flow diagrams [Ward 86] or petri net analysis [Valette 88], and specific languages [Harel 90, Aeuernheimer 86]. The approaches of CCS [Milner 80] and Lotos [Brinksma 85] have influenced our work. ELECTRE is a specific language allowing the specification of behaviours concerning such notions as process preemption and process blocking. It does not support the description of the sequential process itself. ELECTRE can be classified as an asynchronous language, adapted for the programming of reactive systems, real-time systems in particular.

Among other functions, ELECTRE can be used in the following three ways :

- Validation of temporal behaviour specification. A simulator [Creusot 88] is currently available. It accepts external stimuli representing events, and shows to the user the modification of the state of the ELECTRE specification.
- Monitoring the dynamic behaviour of an application. ELECTRE expressions are used as redundant specifications, either to support debugging as in [Bruegge 83], or in order to provide a fault-tolerant mechanism for limiting detection latency of faults due to synchronization errors.
- Programming specification of the application's concurrency aspects. ELECTRE is used as a programming language. This is the approach which we have chosen and that we describe in this paper.

The rest of the paper is structured as follows. We first briefly describe ELECTRE. The kind of specification and programming environment in which ELECTRE can be included is then sketched, and the resulting execution environment, with a particular emphasis on display facilities, is presented in detail. Finally a conclusion with a description of current research is provided.

This project is partially funded by the French Ministry of Industry under the STRIN programme. The initial development of a simulator [Creusot 88] within the Electre project has been funded by Renault-Automobiles. It was used as a specification tool in the field of embedded systems.

## 2 Brief description of the ELECTRE language

Like some other languages [Benveniste 91], the ELECTRE language emerged from research concerning effective ways to express process behaviour and synchronization in reactive systems. [Pnueli 86, Harel 85, Benveniste 91] characterize those systems (e.g. real-time systems) by their reactions to external stimuli (e.g. sensors signals) in order to produce outputs (e.g. actuator commands). Moreover, those systems have to react to and act on an environment that constraints the reaction rate.

ELECTRE is based on path expression theory [Campbell 74]. Path expressions were mainly developed for the synchronization of concurrent processes sharing a resource. They allow the description of how concurrent processes are coordinated in the sharing of a resource. They are a convenient way of expressing constraints that processes must meet in order to guarantee that actions concerning a given shared object are executed in an orderly manner.

Path expressions have also been used as a debugging mechanism to monitor the dynamic behaviour of a computation [Bruegge 83]. In this case, path expressions are used as redundant specifications of the expected behaviour of the program being debugged in order to detect potential deviations. ELECTRE uses path expressions to specify the temporal behaviour of reactive systems.

The basic idea in the design of the ELECTRE language is to separate process synchronization constraints from algorithmic descriptions which are encapsulated in entities called *modules*. Four different behaviours concerning how modules may run with regard to other modules are available : modules may execute sequentially, in parallel, or exclusively, or a module may be repeated. These are denoted in the language by syntactic symbols *space* ,  $\parallel$  ,  $|$  , and  $*$  respectively.

ELECTRE can express whether the execution of a module has to start upon a certain event occurrence, and whether it can be interrupted by another event occurrence. ELECTRE actually deals with two entities, modules and events. These entities are designated by identifiers appearing in ELECTRE programs together with operators. The operators  $\uparrow$  and  $/$  correspond to two different types of preemption operators, and  $:$  corresponds to a module activation operator :

- The  $\uparrow$  operator associates a module with an event (e.g.  $M \uparrow e$ ). It indicates that the occurrence of  $e$  while  $M$  is active has the effect of preempting  $M$ .
- The  $/$  operator associates a module with an event (e.g.  $M/e$ ). It also indicates that the occurrence of  $e$  while  $M$  is active has the effect of preempting  $M$ , but preemption is mandatory. If  $e$  has not occurred while  $M$  is active,  $M$  must wait for the occurrence of  $e$  upon completion.
- The  $:$  operator associates an event with a module (e.g.  $e : M$ ). It indicates that upon occurrence of event  $e$ , the module  $M$  must be activated.

Thus modules can be considered as task sections which include no synchronization or blocking point. Their execution code may be expressed in any sequential language (e.g. C). A module may be in one of the following states :

- not existing. It has not been activated, or it has completed,
- active. It has been activated,
- interrupted. An event occurrence preempted its execution, and the module may be either resumed or restarted.

Since an event is linked to the ordered history of the occurrences of a specific signal, an event may be in one of the following states :

- not existing. No occurrence was memorized,
- memorized. There were one or several occurrences which have been noted but not yet acted upon,

- active. An occurrence was taken into account and it gave way to the activation of a module which has not yet completed.

In order accurately to express the temporal control in a synchronous or asynchronous application, some further properties may be associated with events and with modules.

Properties concerning events express how an event occurrence is memorized. After an event is used, it is erased from memory, i.e. *consumed* in ELECTRE terms.

There are four properties qualifying the memory of events : the default property, the @ property, the # property and the \$ property. Events are qualified by preceding their identifiers by one of the three symbols (e.g. \$e) or no symbol in the case of the default property. In the default property, an event occurrence is consumed upon the completion of the module it activated. The @ symbol qualifies "fleeting" events. Their occurrence cannot be memorized. Therefore, such events are either taken into account and consumed when they occur, or their occurrence is lost. The # symbol qualifies an event for which all occurrences are memorized. Finally, the \$ symbol qualifies events which activate a module and whose occurrence is deleted from the memory upon the activation.

Properties concerning modules express preemption properties. There are three properties qualifying modules : the default property, the > property and the ! property. Modules are qualified by preceding their identifiers by one of the two symbols (e.g. > M) or no symbol in the case of the default property. In the default property, a module may be preempted at any time, and if it has to be activated again, it is resumed at the point where it was preempted. The > symbol qualifies modules which are restarted rather than resumed when they are reactivated. The ! symbol qualifies modules which cannot be preempted when they are active.

The above description describes the basic semantics of the language. The reader may refer to [Perraud 92] for an exhaustive and formal description of the operational semantics of the language. Figures 1 and 2 provide two temporal diagrams illustrating the execution of an ELECTRE program :

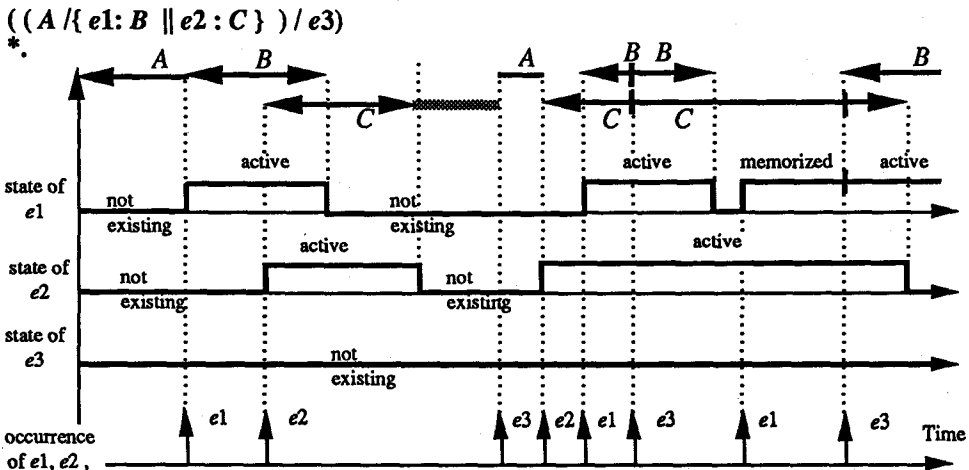


Figure 1. Execution of an ELECTRE program.

This first example shows the preemption of module *A* by the occurrences of events *e1* and *e2*, which have the effect of activating modules *B* and *C* respectively. This sequence is itself preempted by the occurrence of event *e3*, which provokes the repetitive execution of the sequence (and leads either to the resumption or to the reactivation of *A*). Note that the second occurrence of *e3* preempts modules *B* and *C* transiently. Since *e2* and *e1* are not consumed yet, they must be taken into account again, leading to the immediate resumption of modules *B* and *C*. The thick grey lines correspond to waiting states. For instance, *C* is waiting the occurrence of event *e3*.

$(A / \{e1: B \parallel \{e2: C \parallel e3: D\} : E\})^*$ .

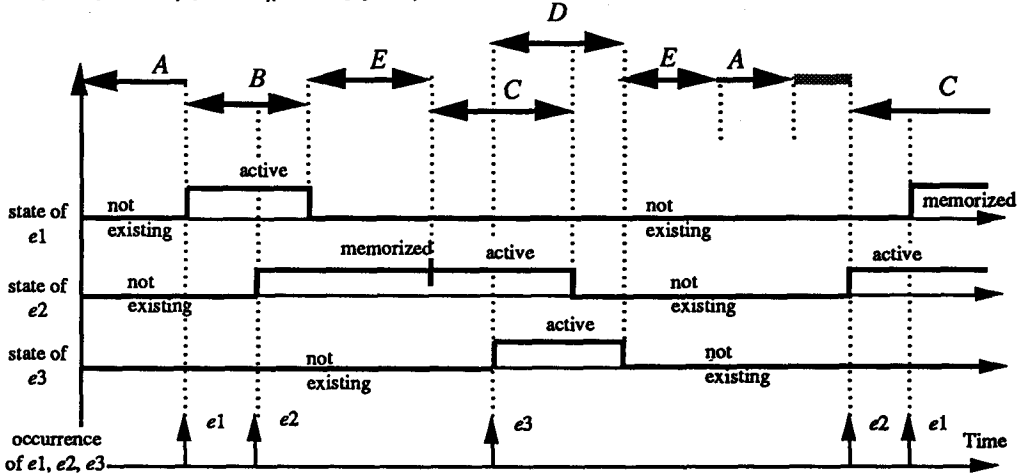


Figure 2. Execution of an ELECTRE program.

In this example, *A* starts to run and may be preempted by either *e1* to activate *B* exclusively, or *e2* or *e3* to activate respectively modules *C* or *D* in parallel. Moreover, the completion of *B*, or of *C* or *D*, activates module *E*, and this whole sequence is then repeated.

An example of an ELECTRE program specifying the behaviour of processes is the well-known Readers-Writers problem [Elloy 85]. Assume that a given resource can be read simultaneously by three reader processes *R1*, *R2*, *R3* or written by one of the two writer processes *W1* and *W2*, but *W1* and *W2* cannot write at the same time. Events *r1*, *r2*, *r3* and *w1*, *w2* refer to read and write requests made from the modules *R1*, *R2*, *R3*, *W1*, *W2* respectively.

If no priority is given, the control structure is given by :

$(1 / \{\{r1: R1 \parallel r2: R2 \parallel r3: R3\} \mid w1: W1 \mid w2: W2\})^*$ .

The interrupt structure enclosed in the brackets following the ' / ' symbol indicates that all the events are at the same level and may preempt the background task, designated by 1. Writing by *W1* and writing by *W2* must take place exclusively, and reading must take place exclusively from writing. The inner brackets indicate that when reading occurs, concurrent reading is allowed. With such a control structure, it is possible that writers

never take control. Any time there is a reader, starvation of writers can occur. In order to avoid this, the control structure below gives priority to writers. Each writer can preempt any running task in the following control structure :

$$((1/\{r1: R1 \parallel r2: R2 \parallel r3: R3\})/\{w1: W1 \mid w2: W2\}) * .$$

As a result, any writing request will preempt on-going readings. However, this structure is not totally correct, since events initiating the writers are mandatory : that is when reading is completed it is mandatory to await a write request, although it should be possible to have subsequent reading requests. Furthermore, when writing is performed and completed, control should be given back to the preempted reading tasks. Finally, since the shared resource is modified by writers, the reading task should be restarted at the beginning ( $>$ ). The correct program of the control structure is :

$$((1/\{r1: > R1 \parallel r2: > R2 \parallel r3: > R3\}) \uparrow \{w1: W1 \mid w2: W2\}) * .$$

### 3 Specification and programming environment

#### 3.1 Environment framework

An environment based on ELECTRE includes tools allows :

- the specification of the asynchronous part of the application in terms of concurrency and preemption by events. This is achieved through ELECTRE programs which permit the specification of concurrency, sequencing of modules, and preemption of modules by events.
- the specification of the sequential part of the applications, i.e. modules created with standard tools and languages such as editors and compilers.
- the specification of events. Many entities can be represented as events (e.g. interrupts related to physical external events, software or hardware internal events). Those events can be combined into higher-level events (e.g. every four clock ticks). Ideally tools for expressing relations between events and tools allowing the specification of ELECTRE events are necessary. This area is beyond the scope of the project and will be part of future research.

#### 3.2 Design and generation

The methodology follows the following steps :

- Isolate events which affect execution.
- Refine parallel entities into smaller entities, i.e. modules which contain no synchronisation point. They may contain request for signalling events.
- Derive an ELECTRE specification to describe the behaviour of the application.

Once the specification is available, tools for application compilation and generation can be used.

## 4 Execution environment

### 4.1 Execution subsystem

This section presents the ELECTRE execution system. It first explains its basic architecture and the dependencies between its main units. It then describes the control unit and the module unit.

#### 4.1.1 Architecture

The execution system is based on the architecture shown in the figure below.

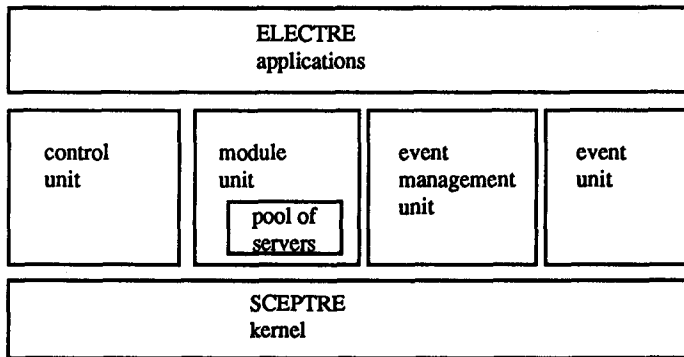


Figure 3. Execution architecture.

This architecture is based on the SCEPTRE standard [SCEPTRE 84], a real-time kernel on top of which higher level services can be provided. Most services are provided by the VDX executive services [Kung 89] to which specific services have been added to support ELECTRE.

The specific ELECTRE services are the following :

- **ELECTRE control unit** : this unit "interprets" ELECTRE control expressions and controls activation of modules. In order to do so, the event unit notifies each ELECTRE event occurrence to the control unit. The control unit then decides upon actions to be performed concerning modules. This entails requests to the module unit. The module unit notifies the control unit of module terminations. Basically, the control unit checks the occurrence of the notified event against a specification represented as a tree.
- **module unit** : this unit handles services allowing the control unit to request activation, termination, preemption, and resumption of a module. For efficiency reasons, modules are not implemented as processes of the underlying real-time executive. Execution of ELECTRE modules is ensured by a pool of server processes. Activation requests are enqueued on a first come / first serve basis.



- **event unit** : this unit receives notification of lower level events, and then decides whether an ELECTRE event has occurred. For instance the ELECTRE event *time out of 50 ms* might be handled by the 50th occurrence of the lower level event *tick 1ms* . Upon detection of occurrence, the event unit calls the event manager unit.
- **event manager unit** : this unit allows the specification of dependencies between ELECTRE events. Those dependencies concern logical and temporal combination of events which cannot be expressed in an ELECTRE specification (e.g. *e1 and e2 occurred, e1 or e2 occurred, e1 and then e2 occurred*).

#### 4.1.2 Dependencies between subsystems

The picture below summarizes dependencies between units. The arrows indicate the interactions between units.

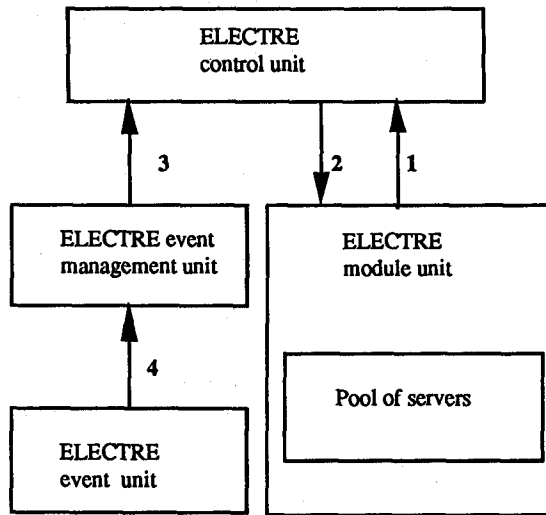


Figure 4. ELECTRE subsystem interfaces.

4 interfaces are identified :

- **interface 1** : the module unit notifies the control unit of module termination.
- **interface 2** : the control unit "interprets" the ELECTRE specification upon receiving event stimuli and requests the module unit for activation, preemption, resumption, or termination of modules.
- **interface 3** : the event manager unit handles dependencies between ELECTRE events and interacts with the control unit.
- **interface 4** : the event unit notifies the event manager unit of an ELECTRE event occurrence.

### 4.1.3 Control unit

The control unit uses ELECTRE expressions to determine which actions to perform in response to the notification of event occurrences. According to the kind of event and the current state of the expression, the control unit can activate or preempt modules or simply memorize the occurrence.

The control unit does not directly interpret the ELECTRE expressions but instead interprets data structures which contain :

- static parameters associated with modules (e.g. identifiers, initialization parameters),
- characterization of the events (e.g. identifiers, type),
- specification of the application represented by an "interpretation tree" which is computed from the ELECTRE sentence before the execution. This tree contains the semantics of the ELECTRE sentence, i.e. all the causal relations between modules or between modules and events.

Two interpretation trees are created. One is directly based on the syntactic and semantic structure of the control expression (structural interpretation tree), and the second one is an optimized version of the first. Figure 5 shows two examples of non-optimized structural interpretation trees.

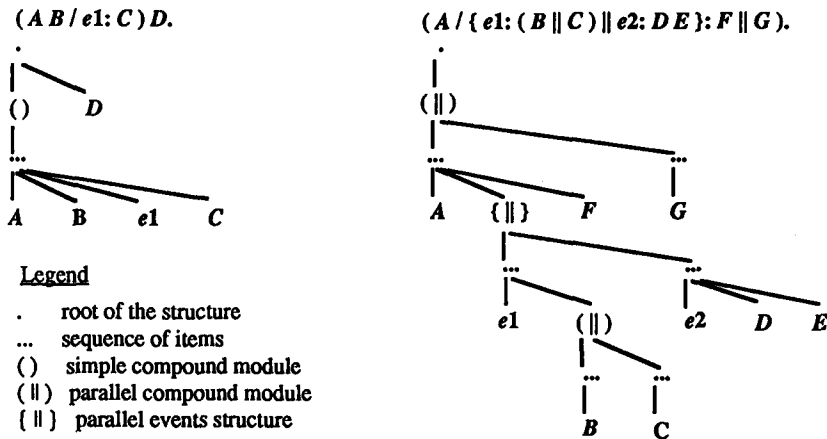


Figure 5. ELECTRE non-optimized structural interpretation trees.

While the building of structural interpretation trees at compile time is fairly simple, interpretation algorithms are often complicated because it is necessary to perform a systematically recursive analysis of the tree. Algorithms for on-line interpretation and off-line compilation can be found in [Creusot 88].

The control unit accesses the tree data structure after receipt of :

- the notification of an ELECTRE event issued from the event manager unit, or

- the notification of the completion of a module issued from the module unit.

As a consequence of the notification, the control unit uses the current state of the tree to activate or preempt modules with the services of the module unit, or it simply memorizes the occurrence according to the type of the event. In the latter case, active modules are not preempted and the occurrence will be taken into account later.

In a real-time context, interpretation algorithms must be efficient which is not the case when using the non-optimized tree. Since the interpretation tree is a structural copy of the initial ELECTRE expression, it "sinks" the entities which direct evolution of the application, i.e. the events. Consequently, a search must be performed for each event occurrence.

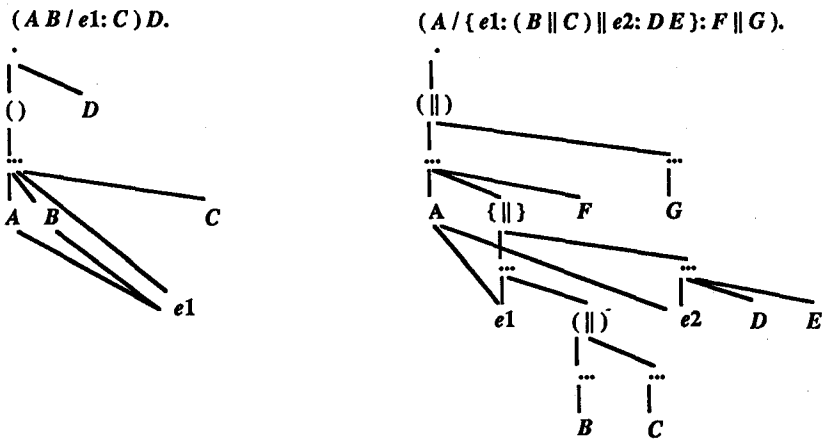


Figure 6. ELECTRE optimized structural interpretation trees.

Another approach based on an optimized interpretation tree has been proposed [Lemoine 90]. All events (those in the sentence and module completion events) are direct entry points of a new structure. The major interest of this approach is to "wire" the causal relations between events, modules and underlying structures (e.g. parallel, compound) in the tree. On-line analysis of such a structure is more efficient even though there are still cases when a recursive search is necessary. Figure 6 shows the result of optimization.

#### 4.1.4 Module unit

The module unit interacts with the control unit. It provides services for activation, preemption, abortion, and resumption. It also calls a termination service of the control unit.

Activation is a service of the module unit requested by the control unit upon event occurrence (e.g.  $e : M$ ), or in a sequence of modules (e.g.  $M1M2$ ) or in a repetition (e.g.  $(M1/e2 : M3)^*$ ).

Preemption is requested upon event occurrence (e.g.  $M/e$ ). It is assumed that the ! preemption property qualifying modules is directly handled by the control unit which

therefore does not request the preemption service. To implement preemption, the module unit uses the preemption services made available by the underlying SCEPTRE kernel.

Abortion is requested for event occurrences implying termination (e.g.  $> M/e$ ). In this case subsequent references to the module correspond to activation (e.g.  $(> M/e)^*$ ).

Service resumption is requested in order to resume a module after preemption. For instance, if  $(M/e)$  is executed in a loop, the second reference to  $M$  corresponds to its resuming.

Termination is reported to the control unit by the module unit in order to allow it to proceed in the interpretation of the ELECTRE control expression.

The implementation of modules is based on a client-server model. A pool of task servers is preallocated when the system starts. This implementation is more efficient because tasks need not be created upon module activation. The number of tasks in the pool is decided by the application. While a known upper bound is the number of modules in an ELECTRE expression, it is not known how to compute statically the exact degree of concurrency of a given ELECTRE expression. If the actual degree of concurrency is greater than the number of tasks in the pool, requests for module activation are enqueued on a FIFO request queue.

## 4.2 Display subsystem

This section presents the ELECTRE display subsystem. Its purpose is to display in real-time or in pseudo real-time (i.e. possibly with some latency) information on ELECTRE execution states. Two types of representation display are used. One focuses on structural aspects, and the other on temporal aspects.

The display subsystem is a key element of an ELECTRE execution system. It is mainly intended for debugging and monitoring needs and to some extent for simulation and validation purposes. Most aspects of the display system are generic and could be generalized to other language approaches.

Figure 7 shows the display environment. The display system is entirely dependent on the ELECTRE control unit, in the sense that it interacts with it. In order to limit display overhead at the main processor level and to allow future extensions for a distributed version of the ELECTRE control unit, the display system runs on another processor connected to the main processor through a local area network. Communication is based on facilities provided by VDX. The display system is based on the Microsoft Windows environment.

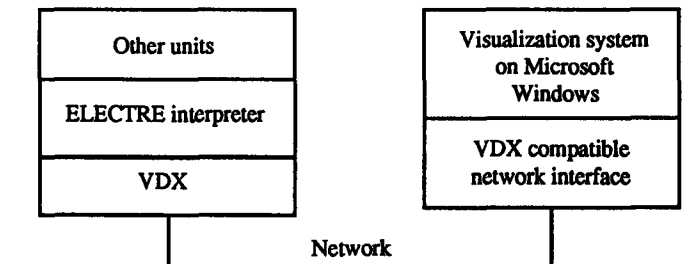


Figure 7. Display system environment.

#### 4.2.1 Display system organisation

**Static and dynamic data structures.** The display system manipulates data structures describing static and dynamic aspects of an ELECTRE control expression. Static aspects relate to information which is independent of execution. This concerns the text of the ELECTRE expression, the interpretation tree structure, and preemption relations between events and modules. Static information is provided at compile time and loaded on the display system before execution starts. Dynamic aspects relate to information concerning a given execution. It mainly concerns modifications of the state of the interpretation tree, i.e. the beginning or the end of an ELECTRE sentence, an event, or the end of a module. These modifications are called *occurrences* in the rest of the document.

Static data are used to implement the various types of display representations explained below. Dynamic data can be saved in an execution history file in order to be reused later. The user of the display system can switch the input of data at any time from the stream provided by the ELECTRE control unit to a history file. To achieve this, the display system is divided in two units : the interface unit and the visualization unit. The interface unit either receives information from the control unit and dispatches it to a history file and to the visualization unit, or it reads information from a history file and dispatches it to the visualization unit. The visualization unit reads dynamic data from the interface unit and displays them.

**Display modes.** When the overall system is started, static information must be made available to the display system through loading directives. At this point, the default display mode is the *disconnected*, or off-line, mode, that is, the display system is not connected to the ELECTRE control unit, i.e. to the application itself. In this mode, the visualization unit obtains information directly from history files. In the *connected*, or on-line, mode, the display system is connected to the application. Data come directly from the control unit and can subsequently be saved in a history file.

Two further submodes are defined in the connected mode, the *decoupled* and the *coupled* submode. The coupled submode is the default submode. In this submode, data transmitted from the control unit is directly displayed. A user wishing to replay a sequence during the execution can set the decoupled submode. In this submode, data coming from the interpreter are redirected to a temporary file while a history file (typically the current one) can be replayed as in the unconnected mode. Upon returning to the coupled mode, the temporary file is first used to update the display with all changes that occurred during the decoupled phase.

**Simulation, debugging and monitoring facilities.** When the display system is in the disconnected mode or in the decoupled submode of the connected mode, the resulting system can be used for simulation. The display of modifications can be performed either on a step by step basis or by an adjustable timer. When the display system is in the coupled submode of the connected mode, the resulting system can be used for debugging and monitoring. In order to help in detecting specific occurrences, the display system can be stopped and resumed at specific points. The user can define breakpoints and decoupling points on a given occurrence. Breakpoints have the effect of stopping the

application itself. Decoupling points have the effect of forcing the display system to be in the decoupled submode.

#### 4.2.2 Display representations.

Two main types of representation are discussed here : structural representation and temporal representation. The examples below use the following ELECTRE sentence.

$$((A \parallel B) * / \{e1\} : (C / e2 \parallel D / \{e3 : E \mid e4 : FG \mid \{e5 : H \parallel e6\} : I \parallel J)) \uparrow e7.$$

**Structural representation.** This kind of representation is mainly based on a tree structure derived from the syntactic and semantic structure of an ELECTRE expression. Figure 8 shows the representation icons used to display basic entities (modules, events), basic relations (sequentiality of modules, necessary preemption, optional preemption), and syntactic non-terminal entities (simple compound module, parallel compound module, repetitive module, simple compound event, exclusive compound event and parallel compound event). Non-terminal entities have two representations : a simplified representation and an extended representation.

Basic representations

A	module
(e1)	event
(e2) →	started module
—	sequentiality of modules
— / —	necessary preemption
— ^ —	optional preemption

Simplified and extended representations

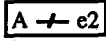
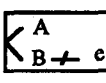
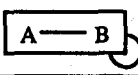

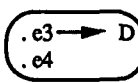
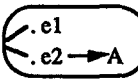
()		simple compound module
(II)		parallel compound module
0*		repetitive module
{ }		simple compound event
{ I }		exclusive compound event
{ II }		parallel compound event

Figure 8. Structural display icons.

The user can select between the simplified and the extended representation by clicking on the representation. Figure 9 shows an example of a display with all extended representations. When the simplified representation is used, all details concerning the corresponding non-terminal entities are hidden.

The purpose of the simplified representation is twofold. It helps in encapsulating some parts of the ELECTRE sentence and it also reduces the size of the overall display. Vertical and horizontal scrolling are also supported, but it was felt that they did not preclude for the need of a simplified representation.

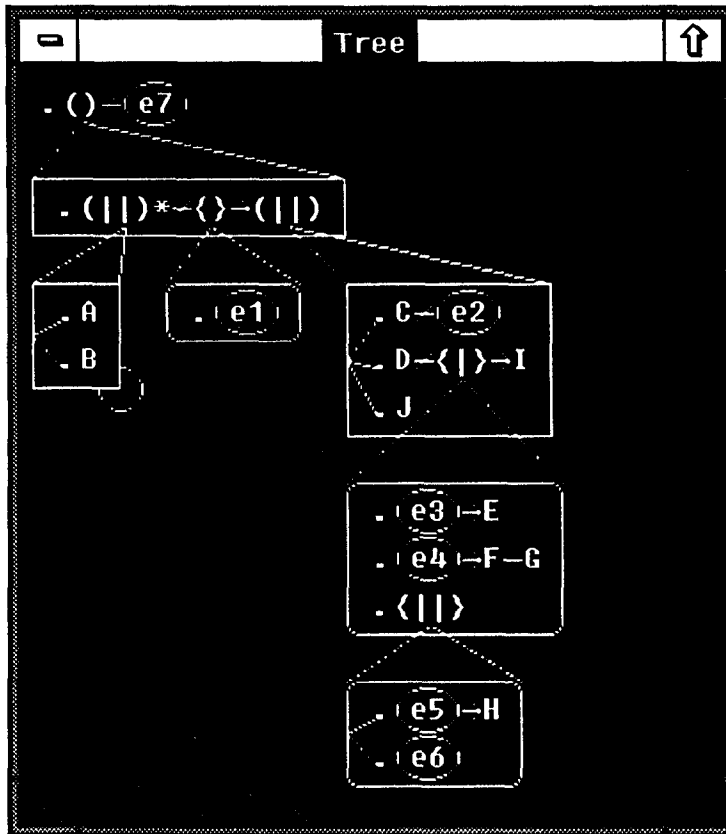


Figure 9. Structural display.

The effect of event occurrences on the structural display is to change the color of the affected element in the representation. For instance the occurrence of event *e5* will change the color of the entity representing the event in the tree.

Our experiments on the use of structural displays showed that they bring a good overall view of the ELECTRE expression. On the other hand, they are not very clear for parallel structures or events and give no information on the sequentiality of occurrences.

**Temporal representation.** Temporal display representations are based on chronograms. They focus on the state of events (memorized, not existing, active), on the sequencing and states of modules, and on the concurrency of modules. Figure 10 shows the representation icons used to display basic entities.

$\leftarrow A \text{ ---}$	module start	$\text{--- } A \text{ )}$	module preempted
$\text{--- } A \rightarrow$	module end	$\text{--- } A \text{  }$	module stopped
$\text{--- } A \text{ ---}$	no change	$\{ A \text{ ---}$	module restarted
$\text{--- } \text{---}$	parallel sequence ended	$[ \quad ]$	parallel structure
0	event awaited		

Figure 10. Temporal display icons.

The display representation uses the horizontal axis to represent time. To avoid scaling problems, the temporal axis is not defined by dates, but by occurrences. Horizontal lines show the behaviour of a given module. Figure 11 shows a temporal representation display.

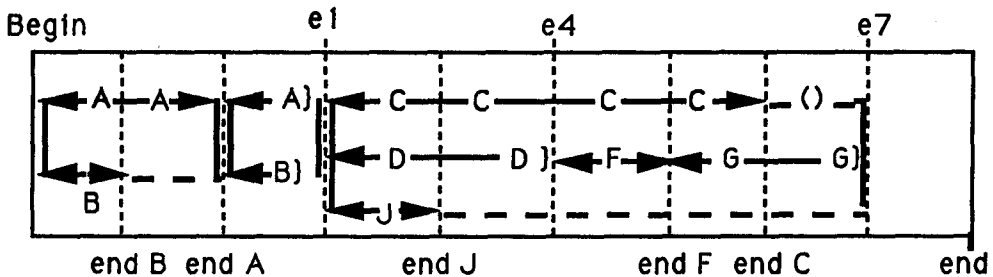


Figure 11. Temporal representation display.

The effect of occurrences on the temporal display is to scroll the window from right to left. Our experiments on the use of temporal displays showed that they bring a good overall sense of concurrency aspects of ELECTRE expressions.

**Other representations.** Other display facilities that have been made available are

- the textual display of the ELECTRE expression.
- the display of preemption relations between events and modules.
- a map corresponding to a simplified display of the structural representation. It is used to locate points and navigate in the structure. Clicking in a location of the map will cause the structural display window to show the corresponding part of the tree structure.
- the display of information concerning occurrences. This is obtained by clicking directly in the corresponding representation of the structural display. Temporary windows are used to provide information like the state of an occurrence, the date of an event, the location in the structure, and so forth.



Figure 12 is an example of an overall display.

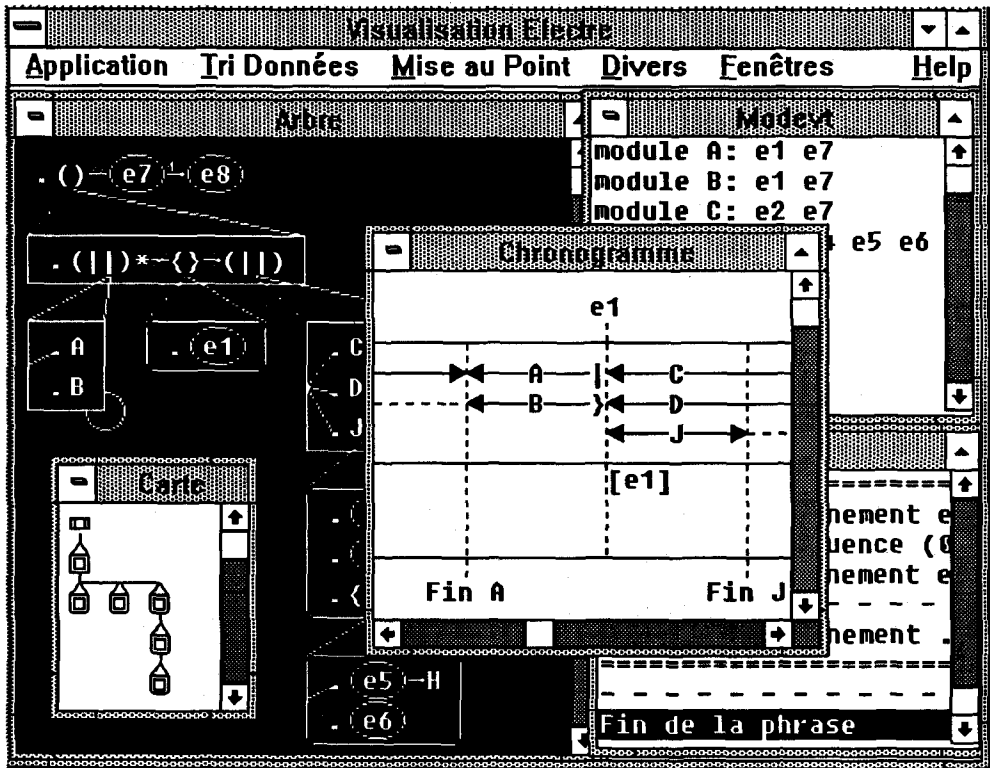


Figure 12. Overall display.

## 5 Conclusion

The project began in mid-1989 and was completed early in 1991, on a 250Kbit/s CSMA/CD network supplied by Compex at Annecy, France, using the VDX distributed executive provided by Renault. It runs on PCs. Experimentation on the resulting system is continuing. The project has shown the feasibility of directly using ELECTRE expressions for execution and constitutes only one step toward the identification and provision of a comprehensive range of tools for the specification and programming of safety-critical, distributed, real-time applications.

To this end, four main research directions are currently being investigated : the direct generation of an ELECTRE compiler from its formal semantic specification, fault-tolerance support, expression of timing constraints and distributed systems.

Compiler generation is based on the building of a rewriting system based on an attribute grammar describing ELECTRE. This system leads to the generation of a transition

system which can be used as a compiler.

Concerning fault-tolerance, the ability to specify fault-tolerant behaviours of real-time applications in ELECTRE is being investigated. In particular, extensions have already been studied to allow the programmer to express either active or passive software redundancy techniques applied to modules.

Research is being carried out on the specification of critical timing constraints. Those specifications are associated with modules. The objective is to use the specifications in order to help select the appropriate scheduling policy.

The distribution of modules in a Local Area Network environment is also investigated. The goal is to have ELECTRE programs describe the behaviour of a global distributed system. Thus, the main issue is to make sure that each local action is compatible with the global ELECTRE program. The approach involves replicating the ELECTRE program at each site of the network and adding distributed synchronization techniques to ensure that the control sequence progress is the same everywhere.

## 6 Acknowledgements

We would like to thank Jean-Pierre Elloy, the head of the ENSM (Ecole Nationale Supérieure de Mécanique) real-time research team for the support he provided for our project. We are also grateful to Jérôme Billion, Adam Mirowsky and Nabil Zakhama for their participation in the implementation of the ELECTRE execution environment.

## Bibliography

- [Aeuernheimer 86 ] B. Aeurnheimer, R.A. Kermmerer. *RT-ASLAN : A Specification Language for Real-Time Systems*. IEEE Transactions on Software Engineering, Vol. SE-12, n. 9, pp. 879-889, September 1986.
- [Benveniste 91 ] A. Benveniste, G. Berry. *Real-Time Systems Design and Programming*. Special section of the Proceedings of the IEEE on real-time programming. To appear in autumn 1991.
- [Bruegge 83 ] B. Bruegge, P. Hibbard. *Generalized Path Expressions : A High Level Debugging Mechanism*. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, pp. 34-44, March 1983.
- [Brinksma 85 ] E. Brinksma. *A tutorial on Lotos*. IFIP Workshop on Protocol Specifications, Testing and Implementation, Moissac, 1985.
- [Campbell 74 ] R.H. Campbell, A.N.Habermann. *The Specification of Process Synchronization by Path Expressions*. Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, pp. 89-102, December 1974.
- [Creusot 88 ] D. Creusot. *Guide d'utilisation du système ELECTRE Version PC*. Rapport de contrat Renault, VEH-ELE-D1, October 1988.

- [Deplanche 88 ] A.-M. Deplanche, J.-P. Elloy, O. Roux. *Redundancy in Fault Tolerant Real-time Process Control Systems*. Congrès mondial IMACS, Paris, July 1988.
- [Dixon 86 ] R.D. Dixon, D. Hemmendinger. *Analyzing Synchronization Problems by Using Event Histories as Languages*. pp. 183-188, 1986.
- [Elloy 85 ] J.P. Elloy, O. Roux. *ELECTRE : A Language for Control Structuring in Real-Time*. The Computer Journal, Vol. 28, n. 5, 1985.
- [Faulk 88 ] S.R. Faulk, D.L. Parnas. *On Synchronization in Hard-Real-Time Systems*. ACM, Vol. 31, n. 3, March 1988.
- [Harel 85 ] D. Harel, A. Pnueli. *On the Development of Reactive System : Logic and Models of Concurrent Systems*. NATO ASI Series, Vol. 13 (K.R.Apt, ed.), Springer-Verlag, New-York, pp. 477-498, 1985.
- [Harel 90 ] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Stull-Trauring, M. Trakhtenbrot. *STATEMATE : A Working Environment for the Development of Complex Reactive Systems*. IEEE Transactions on Software Engineering, Vol. 16, n. 4, (K.R.Apt, ed.), pp. 403-414, April 1990.
- [Jahanian 86 ] F. Jahanian, A. Mok. *Safety Analysis of Timing Properties in Real-Time Systems*. IEEE Transactions on Software Engineering, Vol. SE-12, n. 9, pp. 890-904, September 1986.
- [Kung 89 ] A. Kung, I. Lacrouts-Cazenave, C. Serrano-Morales. *Interconnection of Vehicle Software Components*. Working conference on decentralized systems. IFIP W.G.10.3, Lyon, December 1989.
- [Lemoine 90 ] P. Lemoine, Y. Trinquet, J. Perraud. *Une proposition de modification de la structure d'arbre ELECTRE*. Internal report LAN, 1990.
- [Milner 80 ] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Springer-Verlag, n. 92, 1980.
- [Ostroff 90 ] J.S. Ostroff. *A Logic for Real-Time Discrete Event Processes*. IEEE Control System Magazine, pp. 95-102, June 1990.
- [Perraud 92 ] J. Perraud, O. Roux, M. Huou. *Operational Semantics of a Kernel of the Electre Language*. To appear in Theoretical Computer Science, n. 100, November 1992.
- [Pnueli 86 ] A. Pnueli. *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends*. Current Trends in Concurrency (Bakker & Al. eds.). Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, Berlin, pp. 510-584, 1986.
- [SCEPTRE 84 ] SCEPTRE. TSI, Vol. 3, n.1, January-February 1984.

- [Valette 88 ] R. Valette, M. Paludetto, B. Porcher-Labreuil, P. Farail. *Approche Orientée Objet HOOD et Réseaux de Petri pour la Conception de Logiciel Temps-Réel*. Journées Internationales sur le Génie Logiciel et ses Applications, Toulouse (France), December, 1988.
- [Ward 86 ] P.T. Ward. *The Transformation Schema: an Extension of the Data Flow Diagram to Represent Control and Timing*. IEEE Transaction on Software Engineering, Vol. SE-12, n. 2, pp. 198-210, February 1986.