Operational semantics of a kernel of the language ELECTRE *

Jean Perraud, Olivier Roux and Marc Huou

L.A.N./E.N.S.M. (U.A. CNRS No. 823), 1, rue de la Noë, 44072 Nantes Cedex 03, France

Communicated by M. Nivat
Received October 1989
Revised May 1990

Abstract


The real-time language ELECTRE describes behaviours of a real-time application using tasks called modules. Tasks are activated and pre-empted by events that come either from tasks themselves or from the controlled real-time application. To describe a current state of the application one needs both a program ELECTRE and the history of past event occurrences. We give operational semantics for a kernel of the language using a transition system whose transitions are calculated by attribute evaluation on a context-free grammar. It proves that any event occurrence turns any state into a new one, in a deterministic way.

1. Introduction

Synchronous languages [1, 2, 6] refer to a discrete observation of a centralized real-time application and therefore implicitly or explicitly include discrete modelizations of the time, in order to determine accurately what should be simultaneous and what should not.

Contrarily, the language ELECTRE [3] refers to a continuous observation of a distributed real-time application through event occurrences, only assuming that event occurrences never are simultaneous. Thus the language ELECTRE does not reinvent nor describe real time, which is taken into account merely through the occurrences of events whose origin is not described.

The real-time application is divided in such a way that elementary tasks, called modules, no longer include synchronization of blocking points. A program describes

* Works about ELECTRE are supported by the Centre National de la Recherche Scientifique (CNRS France) via the coordinated research program on Cooperation Concurrency and Communication (PRC C3), the French Ministry of Research and Technology, and several French industrial companies.
activation and pre-emption of modules by events, without module procedures written in any classical imperative language, nor depending on the distribution of modules throughout an interconnected computer network. Modules may run under sequential, repetitive, concurrent, and conditional composition.

Works about ELECTRE include a run-time environment: compiler, interpreter and the associated executive.

The ELECTRE program \( a/e_1; b \) means that any occurrence of event \( e_1 \) that drops in during the execution of module \( a \) stops it and starts up the execution of module \( b \). Under program \( ab/e_1 \) every occurrence of \( e_1 \) can stop \( b \) only. If an occurrence of \( e_1 \) drops in during the execution of module \( a \) then it is memorized. It will be taken into account when module \( a \) ends and module \( b \) will not run because it should stop as soon as it would start. Thus, in order to describe a current state of the real-time application, one needs both an ELECTRE program and the history of past event occurrences. At the current state \( (ab/e_1, e_2e_1; e_2e_1) \), module \( a \) is running while several event occurrences have dropped in: occurrences of the unconcerned event \( e_2 \) and occurrences of \( e_1 \) that cannot be taken into account. The occurrences of events and the terminal ends of completed modules turn the current states of the application into other ones. Therefore we describe operational semantics using a labelled transition system whose configurations are the current states of the application and whose transitions are labelled by module ends or by events.

The deletion of event occurrences from the history is called consumption. In order to calculate the semantics of a given ELECTRE program, we first calculate consumptions because they are implicit in the initial language syntax, and we incorporate them into the program by translation into an intermediate language (Section 2.3). The transition system is built for the intermediate language (Section 3). It is deterministic and "complete apart from dead states" (Section 4).

With implementation in mind, we have used attribute grammars [4]. Because this is less concise than the structural approach [7], we only settle here the main evaluation rules.

## 2. Presentation of the language

### 2.1. Syntax and expected semantics

Here we present a context-free grammar generating a kernel of the language. Nonterminal symbols are capitals, and every other symbol is a terminal symbol. The start symbol is \( S \). Productions follow, together with comments referring to underlying ideas which will be settled by the semantics

\[
E ::= e_1 \mid e_2 \mid \cdots \mid e_n \quad \text{(list of event identifiers)}
\]

\[
Q ::= E \mid @E \mid $E \mid #E \quad \text{(qualified events)}
\]

With an event may be associated a property that determines, according to the program, the way its occurrences are consumed. Except for fleeting events (\( @ \)),
which are never recorded, the occurrences of an event are recorded gradually as they drop in and, in case of multiple memorization ($\#$), only the oldest one is forgotten at consumption time; otherwise, all its occurrences are deleted. The consumption of a recorded event occurs either when it is taken into account ($\$), or when the module (or control structure) that it has activated terminates (default case, i.e. standard events). Examples are, for instance,

\[
(a/e_1:b, e_2) \xrightarrow{\epsilon_t} (b, e_2),
\]

\[
(ab/e_1:c, e_1e_2e_1) \xrightarrow{f_a} (c, e_1e_2e_1) \xrightarrow{f_r} (NIL, e_2),
\]

\[
(ab/$e_1:c, e_1e_2) \xrightarrow{f_a} (c, e_2),
\]

\[
(ab/#e_1:c, e_1e_2e_1) \xrightarrow{f_a} (c, e_1e_2e_1) \xrightarrow{f_r} (NIL, e_2e_1);
\]

where $f_a$ stands for the terminal end of module $a$.

\[
K ::= Q \mid \{I\};
\]

A compound event is a qualified event or a bracketed interruption structure. An interesting use of bracketed interruption structures arises in the case of unnecessary pre-emption (see below).

\[
I ::= K \mid K:C;
\]

an interruption structure is made of a compound event that may activate a control structure.

\[
M ::= a \mid b \mid \cdots \mid z \mid \# (\text{list of module identifiers});
\]

the symbol $\#$ stands for an “empty” module, which is used to wait for an event; this module has no terminal end.

\[
G ::= M \mid [C];
\]

a compound module is either a module or a square-bracketed control structure.

\[
R ::= G \mid G^*;
\]

a repetitive module is a compound module or a repetitive compound module (followed by an asterisk).

\[
([ab]^*, NIL) \xrightarrow{f_a} (b[ab]^*, NIL);
\]

delayed consumptions ensure casting revivals of interrupted modules:

\[
([a/e_1:b/e_2:c]^*, NIL) \xrightarrow{\epsilon_t} ([b/e_2:c][a/e_1:b/e_2;c]^*, e_1)
\]

\[
\xrightarrow{\epsilon_t} ([c[a/e_1:b/e_2;c]^*, e_1e_2)
\]

\[
\xrightarrow{f_r} ([b/e_2;c][a/e_1:b/e_2;c]^*, e_1).
\]

\[
C ::= R \mid RC \mid R/I \mid R^*K \mid R^*K:C;
\]

a control structure may be either a repetitive module, or a repetitive module sequentially followed by a control structure, or a repetitive module under threat of an interruption structure.
The difference between necessary (/) and unnecessary (*) pre-emption appears when the repetitive module terminates, in which case it is necessary to know whether the interruption structure is K or K : C. For example,

\[
\begin{align*}
(a/{e,:b}:c, \text{NIL}) &\xrightarrow{e_1} (bc, e_1), \\
(a^*{e,:b}:c, \text{NIL}) &\xrightarrow{e_1} (bc, e_1),
\end{align*}
\]

\[
\begin{align*}
(a/{e,:b}:c, \text{NIL}) &\xrightarrow{e_2} (l/{e,:b}:c, \text{NIL}), \\
(a^*{e,:b}:c, \text{NIL}) &\xrightarrow{e_2} (c, \text{NIL}),
\end{align*}
\]

\[
\begin{align*}
(a/{e,:b}, \text{NIL}) &\xrightarrow{e_2} (l/{e,:b}, \text{NIL}), \\
(a^*{e,:b}, \text{NIL}) &\xrightarrow{e_2} (\text{NIL}, \text{NIL}).
\end{align*}
\]

Finally, we have
\[
S ::= C;,
\]

an ELECTRE program is a control structure terminated by a period.

This grammar, called the source grammar, is unambiguous (the proof is standard) and generates a kernel of the language ELECTRE, which we will call the source language with reference to the translation envisaged.

**Example (Leftmost derivation)**

\[
\begin{align*}
S &\rightarrow C \rightarrow R^*K : C \rightarrow G^*K : C \rightarrow M^*K : C \rightarrow a^*K : C \rightarrow a^*\{I\} : C. \\
&\rightarrow a^*\{K : C\} : C \rightarrow a^*\{Q : C\} : C. \\
&\rightarrow a^*\{E : C\} : C \rightarrow a^*\{e_i : C\} : C \rightarrow a^*\{e_i : R\} : C \rightarrow a^*\{e_i : G\} : C. \\
&\rightarrow a^*\{e_i : M\} : C \rightarrow a^*\{e_i : b\} : C. \\
&\rightarrow a^*\{e_i : b\} : R \rightarrow a^*\{e_i : b\} : G \rightarrow a^*\{e_i : b\} : M \rightarrow a^*\{e_i : b\} : c.
\end{align*}
\]

### 2.2. Fleeting events and semantic correctness

The above grammar allows any module or event identifier to appear more than once in a program, and allows event qualification to depend on the instances of the event identifier. However, we want the fleeting nature of some events to result in the immediate forgetting of their occurrences, regardless of the point in the program where the information about this nature is. The fleeting nature of an event must not depend upon the different instances of its identifier in the program. We establish the following semantic rule.

A program is **semantically correct** if and only if either all the instances of an event identifier in this program follow the symbol @, or none. More generally, each time this requirement is fulfilled in a terminal string ~x resulting from a leftmost derivation X + ~x, we call it a significant derivation.

This semantic rule is checked using a two attributes system. The result is \(\text{NOF}(\sim x)\), the set of event identifiers appearing in \(\sim x\) at least once without following the symbol @, and \(\text{FLE}(\sim x)\) with \(\text{FLE}(X) = \{e_i | @ e_i \text{ appears in } \sim x\}\).
Example. The derivation $I \rightarrow S e_1 : a \oplus e_2$ is a significant derivation because $\text{NOF}(S e_1 : a \oplus e_2) = \{e_1\}$ and $\text{FLE}(S e_1 : a \oplus e_2) = \{e_2\}$ have an empty intersection.

2.3. Calculation and expression of consumptions: the target-language

A transition such as $(a/e,: bc, e_2) \rightarrow (bc, e_2e,)$ would lose the information that the terminal end of $b$ must consume $e_1$, which is implicit in program $a/e,: bc$ but does not appear in program $bc$; we shall make it explicit in program $a/e,: b@e,$ which will become $b@e,c$. Therefore, such information is first calculated and then incorporated into the language syntax by translation into the new so-called target-language where consumptions appear explicitly and whose so-called target-grammar has two new $G$ productions,

$$G ::= M \circ Q \mid [C] \circ Q.$$  

This target-grammar is unambiguous as well. (There is at most one leftmost derivation $X \rightarrow \sim x$ with given $X$ and $\sim x$. As before, the proof is standard. We will use this to get a deterministic transition system, calculating transitions by attribute evaluations.)

The calculation of consumptions and the translation into the target-language are done by a three-attributes stacked-up system on the source-grammar. (Throughout this paper we use simple attribute systems always ensuring successful and deterministic evaluation. If the attributes of a system can be numbered so that each of them depends only upon the previous ones, we say that each attribute is stacked up on the previous ones.)

One can feel that there is an "implicit" attribute IMP which synthesizes, for each instance $Y$ of a nonterminal in the leftmost derivation

$$X \rightarrow \sim x Y \sim Z \rightarrow \sim x \sim y \sim Z \rightarrow \sim x \sim y \sim z,$$

the terminal string $\sim y$ derived from $Y$.

The inherited attribute "consumptions", called CSM, is stacked up on attribute IMP. It is defined upon nonterminals $C, R, G, M$ and terminals $[, ], 1, a, b, c, \ldots, z$. It takes its values among terminal strings NIL, $e_i$, $\# e_i$ and its evaluation rules are given in the following way: Since CSM is an inherited attribute, for each production and each instance in its right-hand part of any symbol on which CSM is defined (listed above), an evaluation rule gives the value of CSM as a function of IMP values and already known CSM values. Main evaluation rules are the following. If the control structure $C$ is activated by an event not qualified or qualified by $\#$, then it has to consume this event:

$$I ::= K : C, \text{CSM}(C) ::= \text{if } \text{IMP}(K) = e_i \text{ or } \# e_i, \text{then } \text{IMP}(K) \text{ else NIL.}$$

If an event activates a nonbracketed sequential structure, it must be consumed when the first term of this structure ends. $e_i$ must be consumed at the end of $b$ under
program \( a/e_1; bc \):
\[
C_1 ::= RC_2, \text{CSM}(C_2) := \text{NIL}, \text{CSM}(R) := \text{CSM}(C_1).
\]

Nevertheless, under program \( a/e_1;[bc] \), event \( e_1 \) must be consumed at the end of \( [bc] \) and not at the end of \( b \). The terminal end of general module \([C]\) is distinguished by \( ]\):
\[
G ::= [C], \text{CSM}(C) := \text{NIL}, \text{CSM()} := \text{CSM}(G).
\]

The synthesized attribute “translation”, called TRN, is stacked up on attributes IMP and CSM. It is defined upon every nonterminal and it takes its values among the target-grammar derivations of terminal strings from nonterminals. Thus, there will be no need to prove that the result of the translation is a syntactically correct program of the target-language. The values of TRN corresponding to \( G \) productions depend on the values of CSM, as in the following rule:
\[
G ::= [C] and \text{TRN}(C) = (C \rightarrow c) and \text{CSM()} = \#e_1,
\]
\[
\text{TRN}(G) := (G \rightarrow [C] \otimes Q \rightarrow [c] \otimes Q \rightarrow [-c] \otimes \#E \rightarrow [c] \otimes \#e_1).
\]

The other evaluation rules only rebuild the derivation.

Since the source-grammar is unambiguous, each source-language program has a unique leftmost derivation \( S \Rightarrow \sim s \) for which the attribute evaluation gives, for each instance of terminal symbols \(], l, a, b, c, \ldots, z\) in program \( \sim s \), an unique value of CSM among terminal strings \( \text{NIL}, e_1, \#e_1 \). This points out what must be consumed at the terminal end of a module or a control structure. Moreover, the result of evaluating TRN(S) upon this derivation does exist and it is unique. It is the target-grammar derivation \( S \Rightarrow \sim s' \), the result of which is \( \sim s' \), called the translation of \( \sim s \).

**Example.** Translation of \( a/\#e_1;b \), which has the following leftmost derivation according to the source-grammar:
\[
S_0 \rightarrow C_1. \rightarrow R_2/I_2. \rightarrow G_3/I_3. \rightarrow M_a/I_4. \rightarrow a/I_5. \rightarrow a/K_6;C_6. \rightarrow a/Q_7;C_7.
\]
\[
\rightarrow a/\#E_8;C_8. \rightarrow a/\#e_1;C_9. \rightarrow a/\#e_1;R_{10}.
\]
\[
\rightarrow a/\#e_1;G_{11}. \rightarrow a/\#e_1;M_{12}. \rightarrow a/\#e_1;b.
\]
\[
\text{CSM}(a) = \text{NIL}; \quad \text{CSM}(b) = \#e_1
\]
\[
\text{TRN}(G_{11}) = G \rightarrow M \otimes Q \rightarrow b \otimes Q \rightarrow b \otimes \#E \rightarrow b \otimes \#e_1 \quad \text{because}
\]
\[
\text{CSM}(M_{12}) = \#e_1;
\]
\[
\text{TRN}(S_0) = S \rightarrow C. \rightarrow R/I. \rightarrow G/I. \rightarrow M/I. \rightarrow a/I. \rightarrow a/K:C. \rightarrow a/Q:C.
\]
\[
\rightarrow a/\#E:C. \rightarrow a/\#e_1:C.
\]
\[
\rightarrow a;\#e_1:R. \rightarrow a/\#e_1:G. \rightarrow a/\#e_1:M \otimes Q. \rightarrow a/\#e_1:b \otimes Q.
\]
\[
\rightarrow a/\#e_1:b \otimes \#E. \rightarrow a/\#e_1:b \otimes \#e_1.
\]
Translation and semantic correctness

As for the source-language, we want either all the instances of an event identifier to follow symbol @, or none. Thus, we use semantically correct programs and significant derivations again.

If, in the source program, a module identifier is preceded by #e_; or e_; not following the symbols @ or $, then the translation results in inserting, after this module identifier, the symbol @ followed by #e, or e, respectively. Thus, the instances of event identifiers added by translation never follow the symbols @ or $. As a matter of fact, they will satisfy the requirement of semantic correctness, and we can assert that the translation of a semantically correct source-program is always a semantically correct target-program.

3. Transitions calculation

3.1. Environments and unavoidable deletions

At intermediate steps of the calculation of a transition, we introduce fleeting events and module ends in the history of past occurrences. However, by its nature, a module end is taken into account when it occurs and there is no need to record it: fleeting events and module ends will never appear in a configuration.

An environment, say Env, is a finite sequence of module ends or event identifiers. Because fleeting events depend on the program x, Env@(-x) will denote the new environment achieved from Env by deleting both fleeting events and module end identifiers (unavoidable deletions).

3.2. The transition system

In order to determine configurations and to calculate transitions by attribute evaluation, we will work with pairs (derivation, environment). For a pair (X → ~x, Env) we will evaluate on the derivation X → ~x an attribute depending on the environment Env. The synthesized attribute rewrite of derivations in the environment Env, namely RED_{Env}, is defined upon every nonterminal; it is stacked up on attributes IMP, NOF, FLE, extended to the target-grammar. The result RED_{Env}(X → ~x) of its evaluation will be a new pair (X' → ~x', Env') or (NIL, Env'), and we shall write (X → ~x, Env) RED (X' → ~x', Env') as well as RED_{Env}(X → ~x) = (X' → ~x', Env').

The steady pairs are on the one hand the pairs (NIL, Env) whose second term is an environment, and on the other hand the pairs (X → ~x, Env) for which Red_{Env}(X → ~x) = (X → ~x, Env).

The configurations of the transition system are the steady pairs whose term is either the empty string NIL or a significant derivation that begins with the start symbol S.
Its transitions are defined from the attribute RED\textsubscript{Env} in the following way: Let $(S \rightarrow \neg s, \text{Env})$ be a configuration, $e_i$ (or $f_a$) be a label, and $\text{Env}.e_i$ be the string achieved by the concatenation of symbol $e_i$ to string $\text{Env}$; there is a transition labelled by $e_i$ from configuration $(S \rightarrow \neg s, \text{Env})$ to a configuration $(S \rightarrow \neg s', \text{Env}')$ (respectively $(\text{NIL}, \text{Env}')$) if, and only if, there is a finite sequence of successive calculations

\[
(S \rightarrow \neg s, \text{Env}.e_i) \xrightarrow{\text{RED}} (S \rightarrow \neg s_1, \text{Env}_1) \xrightarrow{\text{RED}} \cdots \xrightarrow{\text{RED}} (S \rightarrow \neg s', \text{Env}')
\]

(respectively $(S \rightarrow \neg s_n, \text{Env}_n) \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}'))$. (Notice that if $X \rightarrow \neg x$ is a significant derivation and if $\text{RED}_{\text{Env}}(X \rightarrow \neg x) = (Y \rightarrow \neg y, \text{Env}')$, then $Y \rightarrow \neg y$ will be a significant derivation as well, because NOF($\neg y$) will be included in NOF($\neg x$) and FLE($\neg y$) in FLE($\neg x$).)

Such a transition has the following meaning: In a current state of the controlled application, the recorded occurrences make up an environment, say Env, the remainder of the control is specified by an Electre program, say $\neg s$, which begins with the identifier of a running module, say $a$, possibly preceded by opening square brackets, and this application state is steady for, in the absence of a new event occurrence, the execution of module $a$ should go up to its terminal end. When event $e_i$ occurs, the controlled application is turned into a possibly new state which is specified by the steady pair $(S \rightarrow \neg s', \text{Env}')$ (respectively $(\text{NIL}, \text{Env}')$). The initial configurations of this transition system are such configurations as $(S \rightarrow \neg s, \text{NIL})$, with an empty history of event occurrences.

In order to calculate the semantics of a given ELECTRE program we first translate it into the target-language. At last we deal with the accessible part of the transition system from initial configuration $(S \rightarrow \neg s, \text{NIL})$, where $S \rightarrow \neg s$ is the result of the translation. Moreover, we are concerned only with transitions whose labels are either the terminal end of a running module or an event appearing in the program.

**Example.** The semantics of program $a^*\{Se_i:b\}$, which is equal to its translation, and whose leftmost derivation according to the target-grammar is

\[
S \rightarrow C \rightarrow R^*K \rightarrow G^*K \rightarrow M^*K \rightarrow a^*K \rightarrow a^*[l] \rightarrow a^*[K:C] \rightarrow a^*[Q:C] \\
\rightarrow a^*[SE:C] \rightarrow a^*[Se_i:C] \rightarrow a^*[Se_i:R] \rightarrow a^*[Se_i:G] \rightarrow a^*[Se_i:M] \rightarrow a^*[Se_i:b].
\]

We will get the $f_a$-labelled transition from the initial configuration $(S \rightarrow a^*[Se_i:b], \text{NIL})$ by first calculating RED\textsubscript{Env} for Env = $f_a$ upon the above derivation, and then by further calculating, if necessary, up to a steady pair. Afterwards, we will seek the $e_i$-labelled transition, and then resume from the reached configurations.

\[
(S \rightarrow a^*[Se_i:b], \text{NIL}) \xrightarrow{f_a} (\text{NIL}, \text{NIL})
\]

\[
(S \rightarrow a^*[Se_i:b], \text{NIL}) \xrightarrow{e_i} (S \rightarrow b, \text{NIL})
\]

\[
(S \rightarrow b, \text{NIL}) \xrightarrow{f_b} (\text{NIL}, \text{NIL}).
\]
3.3. The set of possible values of $\text{RED}_{\text{Env}}$ upon a leftmost derivation $X \rightarrow \sim x$

Three cases may come up depending on nonterminal $X$.

1. A control structure threatened by the preemption of an interruption structure may evolve only if there is no preemption. Therefore an interruption structure $I \rightarrow \sim i$ is said to be idle in environment $\text{Env}$ if it is steady apart from unavoidable deletions, $\text{RED}_{\text{Env}}(I \rightarrow \sim i) = (I \rightarrow \sim i, \text{Env}@(-i))$. Furthermore, an interruption structure evolving without reaching its end is rewritten into a control structure, and this is the same for compound events. For example when $e_i$ occurs, interruption structure $e_i:a$ and compound event $\{e_i:a\}$ must be rewritten into control structure $a$.

\[
X = K \text{ or } I,
\]
\[
\text{RED}_{\text{Env}}(X \rightarrow \sim x) = (\text{NIL}, \text{Env}') \text{ or } (X \rightarrow \sim x, \text{Env}@(-x)) \text{ or } (C \rightarrow \sim c, \text{Env}').
\]

2. An Electre program, a control structure, a repetitive module or a compound module may either reach their terminal end, or yield respectively to a new Electre program, control structure, repetitive module or compound module.

\[
X = S, C, R \text{ or } G,
\]
\[
\text{RED}_{\text{Env}}(X \rightarrow \sim x) = (\text{NIL}, \text{Env}') \text{ or } (X \rightarrow \sim x', \text{Env}').
\]

3. Since the language does not specify modules programs, the modules have no visible progression except their terminal end. It is the same for simple events.

\[
X = M, E \text{ or } Q,
\]
\[
\text{RED}_{\text{Env}}(X \rightarrow \sim x) = (\text{NIL}, \text{Env}@(-x)) \text{ or } (X \rightarrow \sim x, \text{Env}@(-x)).
\]

The values of the new environment $\text{Env}'$ which are not specified in the above table will be calculated by the attribute evaluation of $\text{RED}_{\text{Env}}$, which always begins with the unavoidable deletions.

3.4. Main evaluation rules of $\text{Red}_{\text{Env}}$

3.4.1. Initial rules

Because the productions of $M$ and $E$ are terminal strings, evaluation is initiated at these nonterminals.

\[
E ::= e_i.
\]

If $e_i$ appears in $\text{Env}$, then $\text{RED}_{\text{Env}}(E \rightarrow e_i) = (\text{NIL}, \text{Env}@((e_i))$ else $\text{RED}_{\text{Env}}(E \rightarrow e_i) = (E \rightarrow e_i, \text{Env}@((e_i))$ (idleness).

\[
M ::= a.
\]

If $f_a$ appears in $\text{Env}$, then $\text{RED}_{\text{Env}}(M \rightarrow a) = (\text{NIL}, \text{Env}@((a))$ else $\text{RED}_{\text{Env}}(M \rightarrow a) = (M \rightarrow a, \text{Env}@((a))$. 


3.4.2. Interruption structures

Interruption structure without activation,

\[ I ::= K. \]

If \( \text{RED}_{env}(K \rightarrow \sim k) = (K \rightarrow \sim k, \text{Env}\@^\sim k) \) then \( \text{RED}_{env}(I \rightarrow K \rightarrow \sim k) = (I \rightarrow K \rightarrow \sim k, \text{Env}\@^\sim k) \) (idleness transfer). Such a rule will now be written:

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (K \rightarrow \sim k, \text{Env}\@^\sim k) \\
(I \rightarrow K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (I \rightarrow K \rightarrow \sim k, \text{Env}\@^\sim k)
\end{align*}
\]

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}') \\
(I \rightarrow K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}')
\end{align*}
\]

where

\[
\text{if } \sim k = e_k \text{ or } S e_k \text{ or } @ e_k \text{ then } \text{Env}' \text{ is achieved by deleting from } \text{Env}' \text{ every instance of } e_k,
\]

\[
\text{if } \sim k = \# e_k \text{ then } \text{Env}' \text{ is achieved by deleting from } \text{Env}' \text{ only the first instance of } e_k,
\]

\[
\text{else } \text{Env}' := \text{Env}'
\]

(simple events that do not activate any control structure are consumed according to their qualification).

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow \sim c, \text{Env}') \\
(I \rightarrow K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow \sim c, \text{Env}')
\end{align*}
\]

Interruption structure including activation,

\[ I ::= K: C. \]

The evaluation rules stand whatever the value of \( \text{RED}_{env}(C \rightarrow \sim c) \) is:

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (K \rightarrow \sim k, \text{Env}\@^\sim k) \\
(I \rightarrow K:C \rightarrow \sim k: \sim c, \text{Env}) & \xrightarrow{\text{RED}} (I \rightarrow K:C \rightarrow \sim k: \sim c, \text{Env}\@^\sim k: \sim c)
\end{align*}
\]

(idleness transfer, if some fleeting event identifier appears in \( \sim c \), then it must be deleted from the environment).

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}') \\
(I \rightarrow K:C \rightarrow \sim k: \sim c, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow \sim c, \text{Env}'\@^\sim k: \sim c)
\end{align*}
\]

where

\[
\text{if } \sim k = S e_k \text{ then } \text{Env}' \text{ is achieved by deleting from } \text{Env}' \text{ every instance of } e_k,
\]

\[
\text{else } \text{Env}' := \text{Env}'
\]

(notice the specific processing of symbol $: following events are consumed as soon as they are taken into account).

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow \sim c', \text{Env}') \\
(I \rightarrow K:C \rightarrow \sim k: \sim c, \text{Env}) & \xrightarrow{\text{RED}}
\end{align*}
\]

\[
(C \rightarrow RC \rightarrow GC \rightarrow [C]C \rightarrow [\sim c']C \rightarrow [\sim c'] \sim c, \text{Env}'\@^\sim k: \sim c)
\]
Operational semantics of a kernel of the language ELECTRE

(3.4.3) Delayed consumptions

\[ G := M \odot Q, \quad G := [C] \odot Q. \]

Evaluation rules for the second production:

\[
\begin{align*}
(C \rightarrow \sim c, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow \sim c', \text{Env'}) \\
(G \rightarrow [C] \odot Q \rightarrow [\sim c] \odot \sim q, \text{Env}) & \xrightarrow{\text{RED}} (G \rightarrow [C] \odot Q \rightarrow [\sim c'] \odot \sim q, \text{Env'}@([\sim c] \odot \sim q)) \\
(C \rightarrow \sim c, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env'}) \\
(G \rightarrow [C] \odot Q \rightarrow [\sim c] \odot \sim q, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env'}@([\sim c] \odot \sim q))
\end{align*}
\]

where

- If \( \sim q = e_k \) then \( \text{Env''} \) is achieved by deleting from \( \text{Env'} \) every instance \( e_k \),
- if \( \sim q \neq e_k \) then \( \text{Env''} \) is achieved by deleting from \( \text{Env'} \) the first instance of \( e_k \),
- else \( \text{Env''} := \text{Env'} \).

(3.4.4) Sequential composition and repetitive structure

\[
C := RC.
\]

The evaluation rules stand whatever the value of \( \text{RED}_{\text{Env}}(C \rightarrow \sim c) \) is.

\[
\begin{align*}
(R \rightarrow \sim r, \text{Env}) & \xrightarrow{\text{RED}} (R \rightarrow \sim r', \text{Env'}) \\
(C \rightarrow RC \rightarrow \sim r \sim c, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow RC \rightarrow \sim r' \sim c, \text{Env'}@([\sim c] \odot \sim q)) \\
(R \rightarrow \sim r, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env'}) \\
(C \rightarrow RC \rightarrow \sim r \sim c, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow \sim c, \text{Env'}@([\sim c] \odot \sim q))
\end{align*}
\]

\[
G := G^*.
\]

\[
\begin{align*}
(R \rightarrow G^* \rightarrow \sim g^*, \text{Env}) & \xrightarrow{\text{RED}} (R \rightarrow G \rightarrow [C] \rightarrow [RC] \rightarrow [GC] \rightarrow [\sim g'R] \rightarrow [\sim g'G]^* \rightarrow [\sim g' \sim g^*], \text{Env'})
\end{align*}
\]

(the execution of a repetitive structure always begins with a first execution of the structure to be repeated).

\[
\begin{align*}
(G \rightarrow \sim g, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env'}) \\
(R \rightarrow G^* \rightarrow \sim g^*, \text{Env}) & \xrightarrow{\text{RED}} (R \rightarrow G^* \rightarrow \sim g^*, \text{Env'})
\end{align*}
\]

(at the terminal end of \( \sim g \), run \( \sim g \) again).
3.4.5. Preemption

Necessary preemption,

\[ C ::= R/I. \]

\[
\begin{align*}
(I \rightarrow \sim i, \text{Env}) & \xrightarrow{\text{RED}} (I \rightarrow \sim i, \text{Env} @ (\sim i)), (R \rightarrow \sim r, \text{Env}) \xrightarrow{\text{RED}} (R \rightarrow \sim r', \text{Env}') \\
(C \rightarrow R/I \rightarrow \sim i, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow R/I \rightarrow \sim r'/\sim i, \text{Env}' @ (\sim r/\sim i))
\end{align*}
\]

(if no preemption occurs, i.e. if the interruption structure is idle, then the execution of \( \sim r \) normally goes on).

\[
\begin{align*}
(I \rightarrow \sim i, \text{Env}) & \xrightarrow{\text{RED}} (I \rightarrow \sim i, \text{Env} @ (\sim i)), (R \rightarrow \sim r, \text{Env}) \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}') \\
(C \rightarrow R/I \rightarrow \sim r/\sim i, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow R/I \rightarrow \sim r/\sim i, \text{Env}' @ (\sim r/\sim i))
\end{align*}
\]

(at the terminal end of \( \sim r \), the preemption must be awaited: this is the main feature of the necessary preemption).

In case of preemption, \( \text{RED}_{\text{Env}}(C \rightarrow R/I \rightarrow \sim r/\sim i) \) does not depend on \( \text{RED}_{\text{Env}}(R \rightarrow \sim r) \):

\[
\begin{align*}
(I \rightarrow \sim i, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}') \\
(C \rightarrow R/I \rightarrow \sim r/\sim i, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}' @ (\sim r/\sim i))
\end{align*}
\]

Unnecessary preemption without activation,

\[ C ::= R^*K. \]

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (K \rightarrow \sim k, \text{Env} @ (\sim k)), (R \rightarrow \sim r, \text{Env}) \xrightarrow{\text{RED}} (R \rightarrow \sim r', \text{Env}') \\
(C \rightarrow R^*K \rightarrow \sim r^*\sim k, \text{Env}) & \xrightarrow{\text{RED}} (C \rightarrow R^*K \rightarrow \sim r^*\sim k, \text{Env}' @ (\sim r^*\sim k))
\end{align*}
\]

(if no preemption occurs, then the execution of \( \sim r \) normally goes on).

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (K \rightarrow \sim k, \text{Env} @ (\sim k)), (R \rightarrow \sim r, \text{Env}) \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}') \\
(C \rightarrow R^*K \rightarrow \sim r^*\sim k, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}' @ (\sim r^*\sim k))
\end{align*}
\]

(the terminal end of \( \sim r \) implies the terminal end of \( \sim r^*\sim k \): the unnecessary preemption is not awaited).

\[
\begin{align*}
(K \rightarrow \sim k, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}') \\
(C \rightarrow R^*K \rightarrow \sim r^*\sim k, \text{Env}) & \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}' @ (\sim r^*\sim k))
\end{align*}
\]

where

- If \( \sim k = e_k \) or \( \text{Se}_k \) or \( \text{@e}_k \) then \( \text{Env}'' \) is achieved by deleting from \( \text{Env}' \) every instance of \( e_k \),

- If \( \sim k = \#e_k \) then \( \text{Env}'' \) is achieved by deleting from \( \text{Env}' \) only the first instance of \( e_k \),

- Else \( \text{Env}'' := \text{Env}' \)
Operational semantics of a kernel of the language ELECTRE

(simple events that do not activate any control structure are consumed according to their qualification).

\[
(K \rightarrow \sim k, Env) \xrightarrow{\text{RED}} (K \rightarrow \sim k, Env') \\
(C \rightarrow R^*K \rightarrow \sim r^* \sim k, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c, Env'@(-r^* \sim k))
\]

Unnecessary preemption including activation,

\[
C ::= R^K C.
\]

\[
(K \rightarrow \sim k, Env) \xrightarrow{\text{RED}} (K \rightarrow \sim k, Env'@(-k)), (R \rightarrow \sim r, Env) \xrightarrow{\text{RED}} (R \rightarrow \sim r', Env')
\]

\[
(C \rightarrow R^*K : C \rightarrow \sim r^* \sim k: \sim c, Env) \xrightarrow{\text{RED}} (C \rightarrow R^*K : C \rightarrow \sim r^* \sim k: \sim c, Env'@(-r^* \sim k: \sim c))
\]

\[
(K \rightarrow \sim k, Env) \xrightarrow{\text{RED}} (K \rightarrow \sim k, Env'@(-k)), (R \rightarrow \sim r, Env) \xrightarrow{\text{RED}} (NIL, Env')
\]

\[
(C \rightarrow R^*K : C \rightarrow \sim r^* \sim k: \sim c, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c, Env'@(-r^* \sim k: \sim c))
\]

(at the terminal end of \( \sim r \), the unnecessary preemption is not awaited and \( \sim c \) starts running; notice that the derivation \( C \rightarrow \sim c \) comes from the evaluation of IMP(C) on the derivation \( R^*K : C \rightarrow \sim r^* \sim k : \sim c \) at the appearing instance of \( C \); in order to refer to this instance of \( C \) we avoided the production \( C ::= \sim R^I \).

\[
(K \rightarrow \sim k, Env) \xrightarrow{\text{RED}} (NIL, Env')
\]

\[
(C \rightarrow R^*K : C \rightarrow \sim r^* \sim k: \sim c, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c, Env'@(-r^* \sim k: \sim c))
\]

where

if \( \sim k = Se_k \) then Env" is achieved by deleting from Env' every instance of \( e_k \),

else Env" := Env'

(notice the specific processing of the symbol $).

\[
(K \rightarrow \sim k, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c', Env')
\]

\[
(C \rightarrow R^*K : C \rightarrow \sim r^* \sim k: \sim c, Env) \xrightarrow{\text{RED}} (C \rightarrow R C \rightarrow G C \rightarrow [C] C \rightarrow [\sim c] C \rightarrow [\sim c'] C \rightarrow [\sim c] C \rightarrow [\sim c'], Env'@(-r^* \sim k: \sim c))
\]

3.4.6. Compound structures

The compound interruption structure \( K ::= \{ I \} \) only induces simple transfers.

Compound control structure,

\[
G ::= [C].
\]

A specific rule is settled to avoid the square brackets to be stacked up in a repetitive structure.

\[
(C \rightarrow \sim c, Env) \xrightarrow{\text{RED}} (C \rightarrow R \rightarrow G \rightarrow \sim g, Env')
\]

\[
(G \rightarrow [C] \rightarrow [\sim c], Env) \xrightarrow{\text{RED}} (G \rightarrow [C] \rightarrow [\sim c], Env')
\]

\[
(C \rightarrow \sim c, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c', Env') \text{ with } C \rightarrow c' \neq C \rightarrow R \rightarrow G \rightarrow \sim g
\]

\[
(G \rightarrow [C] \rightarrow [\sim c], Env) \xrightarrow{\text{RED}} (G \rightarrow [C] \rightarrow [\sim c], Env')
\]

\[
(C \rightarrow \sim c, Env) \xrightarrow{\text{RED}} (NIL, Env')
\]

\[
(G \rightarrow [C] \rightarrow [\sim c], Env) \xrightarrow{\text{RED}} (NIL, Env')
\]

Other productions induce simple transfer evaluation rules.
Example. Evaluation of $\text{RED}_{\text{Env}}$ with $\text{Env} = f_a$, upon the derivation

$$C_1 \rightarrow R_2 \cdot K_2 \rightarrow G_1 \cdot K_3 \rightarrow M_4 \cdot K_4 \rightarrow a^* K_5 \rightarrow a^* \{I_6\} \rightarrow a^* \{K_7: C_7\} \rightarrow a^* \{Q_3: C_9\}$$

$$\rightarrow a^* \{S_{E_0}: C_9\} \rightarrow a^* \{S_{e_1}: C_{10}\} \rightarrow a^* \{S_{e_2}: R_{11}\} \rightarrow a^* \{S_{e_3}: G_{12}\}$$

We just give the values of $\text{RED}_{\text{Env}}$ useful for the final result. $\text{RED}_{\text{Env}}(E_9) = (E \rightarrow e_1, \text{NIL})$ is denoted $(E_9 \rightarrow e_1, f_a) \xrightarrow{\text{RED}} (E \rightarrow e_1, \text{NIL})$, and so on.

4. Properties of the transition system

Since $\text{RED}_{\text{Env}}$, for whatever $\text{Env}$, belongs to a stacked up attributes system on an unambiguous grammar, if there is a transition labelled by $e_i$ (or $f_a$) starting from a configuration $(S \rightarrow \sim S, \text{Env})$, then it is unique: the transition system is deterministic. There is no transition starting from the configurations $(\text{NIL}, \text{Env})$. They are the dead states of the transition system and they stand for the terminal end of the controlled real-time application.

In order to achieve a complete definition of the language semantics, we prove the transition system to be complete “apart from the dead states”: For any configuration $(S \rightarrow \sim S, \text{Env})$ and for any label $e_i$ (or $f_a$), there is a transition starting from $(S \rightarrow \sim S, \text{Env})$.

We argue on the set $D \times E^*$ where $E = \{e_1, e_2, e_3, \ldots\} \cup \{f_a, f_b, f_c, \ldots\}$ and $D$ is the set containing the empty string NIL and all the significant derivations $X \rightarrow \sim x$ from any nonterminal $X$. 
Definition. Let STB be the possibly partial function from $D \times E^*$ into itself, defined as follows: $\text{STB}(\delta, \text{Env}) = (\delta', \text{Env}')$ if, and only if, $(\delta', \text{Env}')$ is a steady pair either equal to $(\delta, \text{Env})$ or for which there is a finite sequence

$$(\delta, \text{Env}) \xrightarrow{\text{RED}} (\delta_1, \text{Env}_1) \xrightarrow{\text{RED}} \cdots \xrightarrow{\text{RED}} (\delta_n, \text{Env}_n) \xrightarrow{\text{RED}} (\delta', \text{Env'}).$$

We will prove this function to be defined for every element of $D \times E^*$, using the length of the environment (i.e. the number $|\text{Env}|$ of symbols in the string Env) as a decrease criteria.

4.1. Environments decrease and further calculations with a decreased environment

Evaluation rules of the attribute RED$_\text{Env}$ were written so as to delete module end and fleeting event identifiers from the environment. This does not decrease the environments containing neither fleeting events nor module end identifiers. Nevertheless, the environment actually decreases each time a control structure ends.

Terminal Environments Lemma. (1) If RED$_\text{Env}(I \rightarrow \sim{i}) = (\text{NIL}, \text{Env}')$, then $|\text{Env}'| < |\text{Env}|$.

(2) If RED$_\text{Env}(K \rightarrow \{I\} \rightarrow \sim{i})) = (\text{NIL}, \text{Env}')$, then $|\text{Env}'| < |\text{Env}|$.

(3) If $X = S, C, R, G$ or $M$ and RED$_\text{Env}(X \rightarrow \sim{x}) = (\text{NIL}, \text{Env}')$, then $|\text{Env}'| < |\text{Env}|$.

Proof. (1) Induction on the derivation length.

(2) Consequence of point (1).

(3) Induction again and point (2). \(\Box\)

Moreover, if a preemption does not occur in a given environment, it is neither expected to occur in a smaller one. And the behaviour specified by a program is expected not to be changed by the deletion from the environment of events which do not appear in the program.

Preemption Idleness Lemma. Let Env and Env' be two environments such that Env' is a substring of Env, if $X = I$, $K$, $Q$ or $E$, and if RED$_\text{Env}(X \rightarrow \sim{X}) = (X \rightarrow \sim{x}, \text{Env}@(-x))$, then RED$_\text{Env}(X \rightarrow \sim{x}) = (X \rightarrow \sim{x}, \text{Env}@(-x))$, and $(X \rightarrow \sim{x}, \text{Env}@(-x))$ is a steady pair.

Foreign Events Lemma. Let Env be an environment, let $X \rightarrow \sim{x}$ be a derivation, let $V$ be a set of event identifiers, let Env $- V$ stand for the environment Env after deletion of all the instances of V-elements; if $V$ has an empty intersection with FLE($\sim{x}$) and with NOF($\sim{x}$), and if RED$_\text{Env}(X \rightarrow \sim{x}) = (Y \rightarrow \sim{y}, \text{Env}')$ (respectively (NIL, Env')), then RED$_\text{Env} - V(X \rightarrow \sim{x}) = (Y \rightarrow \sim{y}, \text{Env} - V)$ (respectively (NIL, Env' - V)).
4.2. The stabilization theorem

**Theorem.** For each significant leftmost target-grammar derivation \( X \rightarrow \sim x \), and for each environment \( \text{Env} \), \( \text{STB}(X \rightarrow \sim x, \text{Env}) \) exists and satisfies the following requirements:

1. If \( X = S, C, R \) or \( G \) then \( \text{STB}(X \rightarrow \sim x, \text{Env}) = (\text{NIL}, \text{Env}') \) or \( (X \rightarrow \sim x', \text{Env}') \).
2. If \( X = K \) or \( I \) then \( \text{STB}(X \rightarrow \sim x, \text{Env}) = (\text{NIL}, \text{Env}') \) or \( (X \rightarrow \sim x, \text{Env}@(\sim x)) \) or \( (C \rightarrow \sim c, \text{Env}') \).
3. If \( X = M, E \) or \( Q \) then \( \text{STB}(X \rightarrow \sim x, \text{Env}) = (\text{NIL}, \text{Env}@(\sim x)) \) or \( (X \rightarrow \sim x, \text{Env}@(\sim x)) \).

**Proof of the theorem.** Once more the proof works by induction on the length of the derivation \( X \rightarrow \sim x \). At length 1 the result is easily checked. Assume the result for every significant derivation with length \( \leq p \), and cut of the first step in a significant derivation with length \( p + 1 \). From the induction hypothesis, on the remaining part of the derivation, for any \( \text{Env} \), \( \text{STB} \) is defined and satisfies the requirements.

Then the result follows a rather repetitive checking, case after case according to the production, and we only look at the case \( R ::= G^* \) for it uses environments decrease, and the case \( C ::= R/I \) for it uses further calculations with a decreased environment.

1. \( R ::= G^* \). The value of \( \text{STB}(G \rightarrow \sim g, \text{Env}) \) may be either \( (G \rightarrow \sim g', \text{Env}') \) or \( (\text{NIL}, \text{Env}') \).

1.1. If \( \text{STB}(G \rightarrow \sim g, \text{Env}) = (G \rightarrow \sim g', \text{Env}') \), there is a finite sequence of calculations

\[
(G \rightarrow \sim g, \text{Env}) \xrightarrow{\text{RED}} (G \rightarrow \sim g_1, \text{Env}_1) \xrightarrow{\text{RED}} \ldots
\]

\[
\xrightarrow{\text{RED}} (G \rightarrow \sim g', \text{Env}') \xrightarrow{\text{RED}} (G \rightarrow \sim g', \text{Env}').
\]

At the first step, evaluation of \( \text{RED}_{\text{env}} \) for production \( R ::= G^* \) gives

\[
(R \rightarrow G^* \rightarrow \sim g^*, \text{Env}) \xrightarrow{\text{RED}} (R \rightarrow G \rightarrow [C] \rightarrow [RC] \rightarrow [GC])
\]

\[
\rightarrow [\sim g_i \sim g^*], \text{Env}_i),
\]

at each remaining step, successive evaluations for productions \( R ::= G, C ::= RC, G ::= [C], R ::= G \) give

\[
(R \rightarrow G \rightarrow [C] \rightarrow [RC] \rightarrow [GC] \rightarrow [\sim g_i \sim g^*], \text{Env}_i)
\]

\[
\xrightarrow{\text{RED}} (R \rightarrow G \rightarrow [C] \rightarrow [RC] \rightarrow [GC] \rightarrow [\sim g_{i+1} \sim g^*], \text{Env}_{i+1})
\]

and at the last step steadiness means: \( \text{STB}(R \rightarrow G^* \rightarrow \sim g^*, \text{Env}) = (R \rightarrow G \rightarrow [C] \rightarrow [RC] \rightarrow [GC] \rightarrow [\sim g' \sim g^*], \text{Env}') \).
(1.2) If $\text{STB}(G \Rightarrow \sim g, \text{Env}) = (\text{NIL}, \text{Env}')$, there is a sequence of calculations
\[
(G \Rightarrow \sim g, \text{Env}) \xrightarrow{\text{RED}} (G \Rightarrow \sim g_1, \text{Env}_1) \xrightarrow{\text{RED}} \cdots
\]
\[
\xrightarrow{\text{RED}} (G \Rightarrow \sim g_k, \text{Env}_k) \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}'),
\]
from which as before
\[
(R \Rightarrow G^* \Rightarrow \sim g^*, \text{Env}) \xrightarrow{\text{RED}} (R \Rightarrow G \Rightarrow [C] \Rightarrow [RC] \Rightarrow [\sim g_1 \sim g^*], \text{Env}_1)
\]
\[
\xrightarrow{\text{RED}} \cdots \xrightarrow{\text{RED}} (R \Rightarrow G \Rightarrow [C] \Rightarrow [RC] \Rightarrow [\sim g_k \sim g^*], \text{Env}_k),
\]
and at the last step evaluations for productions $R ::= G, C ::= RC, G ::= [C]$ and $R ::= G$ give
\[
(R \Rightarrow G \Rightarrow [C] \Rightarrow [RC] \Rightarrow [GC] \Rightarrow [\sim g_k \sim g^*], \text{Env}_k)
\]
\[
\xrightarrow{\text{RED}} (R \Rightarrow G \Rightarrow [C] \Rightarrow [R] \Rightarrow [G^*] \Rightarrow [\sim g^*], \text{Env}')
\]
We now have to find out the existence and the value of $\text{STB}(R \Rightarrow G \Rightarrow [C] \Rightarrow [R] \Rightarrow [G^*] \Rightarrow [\sim g^*], \text{Env}')$ and we work with $\text{STB}(G \Rightarrow \sim g, \text{Env}')$ the existence of which is proved by the induction hypothesis.

(1.2.1) If $\text{STB}(G \Rightarrow \sim g, \text{Env}') = (G \Rightarrow \sim g', \text{Env}'')$, there is a finite sequence of calculations
\[
(G \Rightarrow \sim g, \text{Env}') \xrightarrow{\text{RED}} (G \Rightarrow \sim g_1', \text{Env}_1') \xrightarrow{\text{RED}} \cdots
\]
\[
\xrightarrow{\text{RED}} (G \Rightarrow \sim g', \text{Env}'') \xrightarrow{\text{RED}} (G \Rightarrow \sim g', \text{Env}'').
\]
At the first step successive evaluations for productions $R ::= G^*, C ::= R, G ::= [C]$ and $R ::= G$ give
\[
(R \Rightarrow G \Rightarrow [C] \Rightarrow [R] \Rightarrow [G^*] \Rightarrow [\sim g^*], \text{Env}')
\]
\[
\xrightarrow{\text{RED}} (R \Rightarrow G \Rightarrow [C] \Rightarrow [RC] \Rightarrow [GC] \Rightarrow [\sim g_1' \sim g^*], \text{Env}_1')
\]
(notice that the first evaluation rule for $G ::= [C]$ avoids the square-brackets stacking up).

At remaining steps, working as before we get
\[
\text{STB}(R \Rightarrow G^* \Rightarrow \sim g^*, \text{Env}) = (R \Rightarrow G \Rightarrow [C] \Rightarrow [RC] \Rightarrow [GC] \Rightarrow [\sim g' \sim g^*], \text{Env}'').
\]

(1.2.2) If $\text{STB}(G \Rightarrow \sim g, \text{Env}') = (\text{NIL}, \text{Env}'')$ the sequence of calculations is
\[
(G \Rightarrow \sim g, \text{Env}') \xrightarrow{\text{RED}} (G \Rightarrow \sim g_1', \text{Env}_1') \xrightarrow{\text{RED}} \cdots
\]
\[
\xrightarrow{\text{RED}} (G \Rightarrow \sim g_k', \text{Env}_k') \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}'')
\]
with $|\text{Env}'| < |\text{Env}|$ from the terminal environments lemma. We now have

$$(R \to G^* \to \neg g^*, \text{Env}) \xrightarrow{\text{RED}} \cdots$$

$\xrightarrow{\text{RED}} (R \to G \to [C] \to [R] \to [G^*] \to [\neg g^*], \text{Env}')$

$\xrightarrow{\text{RED}} (R \to G \to [C] \to [RC] \to [GC] \to [\neg g^*], \text{Env}_1') \xrightarrow{\text{RED}} \cdots$

$\xrightarrow{\text{RED}} (R \to G \to [C] \to [RC] \to [GC] \to [\neg g^*], \text{Env}_1'),$

and from $(G \to \neg g^*_1, \text{Env}_1') \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}'_1)$, productions $R ::= G, C ::= RC, G ::= [C]$ and $R ::= G$ give

$$(R \to G \to [C] \to [RC] \to [GC] \to [\neg g^*], \text{Env}_1')$$

$\xrightarrow{\text{RED}} (R \to G \to [C] \to [R] \to [G^*] \to [\neg g^*], \text{Env}'_1).$

and we now have to find out the existence and the value of STB($R \to G^* \to \neg g^*, \text{Env}$) within a finite number of steps.

(2) $C ::= R/I.$

(2.1) If no preemption occurs, we discuss STB($R \to \neg r, \text{Env}$).

(2.1.1) If RED$_{\text{Env}}(I \to \neg i) = (I \to \neg i, \text{Env}_1 @ (-i))$ and $\text{STB}(R \to \neg r, \text{Env}) = (R \to \neg r', \text{Env}')$, there is a finite sequence of calculations

$$(R \to \neg r, \text{Env}) \xrightarrow{\text{RED}} (R \to \neg r_1, \text{Env}_1) \xrightarrow{\text{RED}} \cdots$$

$\xrightarrow{\text{RED}} (R \to \neg r', \text{Env}') \xrightarrow{\text{RED}} (R \to \neg r', \text{Env}',)$

and at its first step we get

$$(C \to R/I \to \neg r/\neg i, \text{Env}) \xrightarrow{\text{RED}} (C \to R/I \to \neg r/\neg i, \text{Env}_1 @ (-i)).$$

Because derivations are significant, elementary calculations show that we may write $\text{Env}_1 @ (-i) = \text{Env}_1 - V$ with the foreign events lemma applying to $V$, which gives

$$(R \to \neg r_1, \text{Env}_1 - V) \xrightarrow{\text{RED}} \cdots \xrightarrow{\text{RED}} (R \to \neg r', \text{Env}' - V)$$

$\xrightarrow{\text{RED}} (R \to \neg r', \text{Env}' - V).$

For any $m$, the preemption idleness lemma gives

$$(I \to \neg i, \text{Env}_m - V) \xrightarrow{\text{RED}} (I \to \neg i, (\text{Env}_m - V) @ (-i)).$$
which induces
\[ (C \to R/I \to \sim r_m/\sim i, \text{Env}_m - V) \xrightarrow{\text{RED}} (C \to R/I \to \sim r_{m+1}/\sim i, \text{Env}_{m+1} - V) \]
\[ (\text{Env}_{m+1} - V)\circ(\sim r_m/\sim i)) \]
with \((\text{Env}_{m+1} - V)\circ(\sim r_m/\sim i) = \text{Env}_{m+1} - V\) which allows to resume.

At the last step steadiness shows that
\[ \text{STB}(C \to R/I \to \sim r/\sim i, \text{Env}) = (C \to R/I \to \sim r'/\sim i, \text{Env}' - V) \].

(2.1.2) If \( \text{RED}_{\text{env}}(I \to \sim i) = (I \to \sim i, \text{Env}@\sim(i)) \) and \( \text{STB}(R \to \sim r, \text{Env}) = (\text{NIL}, \text{Env}') \), from
\[ (R \to \sim r, \text{Env}) \xrightarrow{\text{RED}} (R \to \sim r_1, \text{Env}_1) \xrightarrow{\text{RED}} \cdots (R \to \sim r_k, \text{Env}_k) \xrightarrow{\text{RED}} (\text{NIL}, \text{Env}') \]
this time we get
\[ (C \to R/I \to \sim r/\sim i, \text{Env}) \xrightarrow{\text{RED}} (C \to R/I \to \sim r_1/\sim i, \text{Env}_1 - V) \xrightarrow{\text{RED}} \cdots \xrightarrow{\text{RED}} (C \to R/I \to \sim r_1/\sim i, \text{Env}_k - V) \xrightarrow{\text{RED}} (C \to R/I \to \sim r_k/\sim i, (\text{Env}_k - V)) \].

From \((M \to \upsilon, \text{Env}' - V) \xrightarrow{\text{RED}} (M \to \upsilon, (\text{Env}' - V))\) successive evaluations for productions \(G ::= M, R ::= G, C ::= R/I\) (using the preemption idleness lemma again) give
\[ (C \to R/I \to G/I \to M/I \to \upsilon/\sim i, \text{Env}' - V) \xrightarrow{\text{RED}} (C \to R/I \to G/I \to M/I \to \upsilon/\sim i, (\text{Env}' - V)) \],
and thus
\[ \text{STB}(C \to R/I \to \sim r/\sim i, \text{Env}) = (C \to R/I \to \upsilon/\sim i, \text{Env}' - V) \].

(2.2) If a preemption occurs, i.e. if \( \text{RED}_{\text{env}}(I \to \sim i) \neq (I \to \sim i, \text{Env}@\sim(i)) \), either
\[ \text{STB}(I \to \sim i, \text{Env}) = (C \to \sim c, \text{Env}') \] or \( \text{STB}(I \to \sim i, \text{Env}) = (\text{NIL}, \text{Env}') \).

(2.2.1) If \( \text{STB}(I \to \sim i, \text{Env}) = (C \to \sim c, \text{Env}') \), there is a finite sequence of calculations
\[ (I \to \sim i, \text{Env}) \xrightarrow{\text{RED}} (C \to \sim c_1, \text{Env}_1) \xrightarrow{\text{RED}} (C \to \sim c_2, \text{Env}_2) \cdots \xrightarrow{\text{RED}} (C \to \sim c, \text{Env'}) \].
At its first step evaluation gives \((C \rightarrow R/I \rightarrow \sim r/\sim i, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c_1, Env_1 \oplus (\sim r/\sim i))\).

The foreign events lemma applies to \(Env_1 - V = Env_1 \oplus (\sim r/\sim i)\):

\[
(C \rightarrow R/I \rightarrow \sim r/\sim i, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c_1, Env_1 - V) \xrightarrow{\text{RED}} (C \rightarrow \sim c_2, Env_2 - V) \xrightarrow{\text{RED}} \cdots
\]

\[
(C \rightarrow \sim c_2, Env_2 - V) \xrightarrow{\text{RED}} (C \rightarrow \sim c_3, Env_3 - V),
\]

hence \(\text{STB}(C \rightarrow R/I \rightarrow \sim r/\sim i, Env) = (C \rightarrow \sim c_3, Env' - V)\).

(2.2.2) If \(\text{STB}(I \rightarrow \sim i, Env) = (\text{NIL}, Env')\) from the sequence

\[
(I \rightarrow \sim i, Env) \xrightarrow{\text{RED}} (C \rightarrow \sim c_1, Env_1) \xrightarrow{\text{RED}} (C \rightarrow \sim c_2, Env_2) \cdots
\]

\[
(C \rightarrow \sim c_k, Env_k) \xrightarrow{\text{RED}} (\text{NIL}, Env')
\]

the same work gives \(\text{STB}(C \rightarrow R/I \rightarrow \sim r/i, Env) = (\text{NIL}, Env' - V)\).

5. Conclusion

The language ELECTRE was born from users requests in such an accurate way that compiler and interpreter were first built step by step from experiment. Abstract work on the semantics of a kernel of the language was initiated in order to get rigorous definitions of the operators. Fortunately the language syntax is so simple that formal semantics only uses straightforward attribute evaluation.

The first step is an attribute grammar for the kernel of the language, we are concerned with. This checks semantical correctness, calculates consumptions and incorporates them into the program by translation into the target-language.

The second step is an attribute grammar for the target-language, depending on a parameter Env, which calculates transitions from any given configuration. The result is a deterministic transition system which is complete “apart from the dead states”. It proves that any event occurrence turns any current state of the controlled real time application, which is not its terminal end, into a new one, in a deterministic way. Another point to focus on is to deduce semantical properties of programs from properties of this transition system.

When expanding the transition system to the whole language, we shall have a few more nonterminals and productions within the grammars, describing parallelism and disjunction, without threatening in any way our straightforward attribute evaluations.
References


