In: *European Symposium on Programming (ESOP)*, Copenhague, Danemark, May, 1990

Implementation of an Interpreter for a Parallel Language in Centaur

Yves Bertot
INRIA, Sophia Antipolis
Route des Lucioles, 06565 Valbonne Cedex, France

Abstract

This paper presents the implementation of an interpreter for the parallel language ESTEREL in the CENTAUR system. The dynamic semantics of the language is described and completed with two modules providing a graphical visualization of the execution and a graphical execution controller. The problems of implementing a parallel language using natural semantics and of providing a visualization for a parallel language are especially addressed.

1. The Logical Kernel of the Interpreter

ESTEREL [esterel] is a language involving parallelism and broadcast signal communication used for the description of reactive systems, i.e., systems that react to successions of events [reactive]. The command system for an airplane or the man-machine interface [gfxobj] for an interactive system like CENTAUR are two examples of such reactive systems.

The interpreter described in this paper is based on a description of the *dynamic semantics* of ESTEREL written in natural semantics by the designers of the language [design]. We focus on the implementation of this dynamic semantics within CENTAUR [centaur], using the TYPOL formalism [typol].

We first give a short description of the language and the constructs it contains. Then we give an overview of the semantics' organization. Finally, we concentrate on the key points of the semantics: the use of a rewriting system to express parallelism in the execution of an ESTEREL program.

1.1. *The* ESTEREL *Language*

In [design], Berry and Gonthier give a schematic presentation of reactive systems. Such systems are composed of three layers:

- An *interface* with the environment, in charge of receiving input and producing output.
- A *reactive kernel* that contains the logic of the system. It decides the computations and outputs that must be generated in answer to inputs.
- A *data handling* layer that performs classical computations requested by the logical kernel.

ESTEREL is used to program the reactive kernel. It is not a full-fledged programming language but rather a program generator used to produce a reactive system written in Ada, C, or Le_Lisp[1]. The interface and data handling layer are specified in the host language. The data handling is integrated using abstract data type facilities.

[1] Ada is a trademark of the U. S. DoD, Le_Lisp is a trademark of INRIA.

The basic data manipulated in ESTEREL are signals. they correspond to stimuli that can be emitted and listened in the different parallel processes. Signals can carry additional values. Their communication is conceptually instantaneous and broadcasted. The basic constructs are the following instructions:

- `emit S` or `emit S(exp)`: emit the signal `S` or emit the signal `S` with the value of `exp`.
- `present S then` $\texttt{stat}_1$ `else` $\texttt{stat}_2$ `end`: execute $\texttt{stat}_1$ if `S` is present or $\texttt{stat}_2$ otherwise.
- `do stat watching S`: execute `stat` until the signal `S` appears.
- $\texttt{stat}_1$`;` $\texttt{stat}_2$: execute $\texttt{stat}_1$ and $\texttt{stat}_2$ in sequence, i.e, execute $\texttt{stat}_2$ only when $\texttt{stat}_1$ is terminated.
- $\texttt{stat}_1$`||` $\texttt{stat}_2$: execute $\texttt{stat}_1$ and $\texttt{stat}_2$ in parallel.
- `X := exp`: assign the value of `exp` to the variable `X`. variables can be used in expressions as in any imperative language. They cannot be shared between parallel processes.

The basic concepts of the language are the synchrony hypothesis, parallelism, and broadcast signal communication. The synchrony hypothesis is based on the assumption that each reaction to an input is conceptually instantaneous. The reception of an input event and the emission of the corresponding reaction take place at the same time and define an ESTEREL *instant*.

The parallelism permits to enhance the modularity of the program. It gives an opportunity to design complicated reactive systems by breaking them down into simpler ones that communicate through broadcasted signals. When one of the processes wants to communicate a value, it only emits a signal. This signal is simultaneously received by whichever process is currently listening this channel.

Every correct ESTEREL program describes a relation between an infinite sequence of input events and an infinite sequence of reactions, each event being a collection of present signals, optionally carrying values. This relation can also be easily represented by a deterministic finite state automaton, receiving an input event, changing its state, and emitting the reaction. Naturally, the transition of an automaton is instantaneous. This justifies the synchrony hypothesis.

2. Natural Semantics

We use *natural semantics* to present the different aspects of a language in a unified manner. A natural semantics definition is an unordered collection of rules. A rule has two parts, a numerator and a denominator. Variables may occur in the numerator and the denominator. These variables allow a rule to be instantiated.

The numerator of a rule is again an unordered collection of formulae, the *premises*. The denominator is a single formula, the *conclusion*. Intuitively, if all premises hold, then the denominator holds. Formally, from proof-trees yielding the premises, we can derive a proof-tree yielding the conclusion.

The formulae may have several form depending on the meaning they are given by the programmer. A very frequent form is the *sequent* form. A sequent has two parts, and *antecedent* (on the left) and a consequent (on the right), and we use the turnstile symbol $\vdash$ to separate these parts. The consequent is a predicate. Predicates come in several forms indicated by various infix symbols. These infix symbols have no reserved meaning, they just help in memorizing what is being defined. The antecedent usually contains information on results that are assumed, whereas the consequent represents the information that is being described. For example, the formula:

$$\rho \vdash exp : \tau$$

expresses that in the context ρ (giving e.g. the types of identifiers) the expression exp has the type τ.

Some structure is introduced in the collection of rules. To this end, rules may be grouped into *sets*, with a given *name*. Formulae that are provable by a specific set of rules are usually denoted by placing the set's name on top of the turnstile ($\vdash$), as in the following example:

$$\texttt{eval}$$
$$\rho \vdash exp \rightarrow value$$

Natural semantics define a formalism that enables us to use a computer to reason about the semantics of a language. Typically, our unknown will be type values, execution states or generated code. There are many approaches to turn semantics definition into executable code. The one we use for this implementation of an interpreter is to compile rules into Prolog code, taking advantage of similarity of Prolog variables and variables in inference rules. Roughly speaking, the conclusion of a rule maps to a clause head, and the premises to the body. Distinct forms of formulae map to distinct Prolog predicates. An equation is turned into a Prolog goal. Given an ESTEREL program and ESTEREL's dynamic semantics, the interpretation of the program can be obtained by executing the corresponding Prolog goal. The compiler is provided by the CENTAUR system. The fact that we use Prolog to execute the dynamic semantics will appear in section 3 dealing with the control of the execution.

2.1. *The Dynamic Semantics of* ESTEREL

The data manipulated by the dynamic semantics are the executed program, the memory environment, and an input/output handler. During the interpretation, the program and the memory evolve to take into account the effects of execution. Any communication with the outer world is performed through the input/output handler. This handler permits a symbolic linking of the interpreter with an interface.

2.1.1. *The Main Loop*

Due to the synchrony hypothesis, any reaction to an input event is supposed to be instantaneous, thus defining an ESTEREL *instant*. The main body of the execution is a loop where each pass corresponds to one instant. This loop consists of four phases:

1. reception of an event.
2. computation of a reaction to the event.
3. emission of the computed reaction.
4. preparation of the program for the next instant.

In most cases, this loop is infinite since ESTEREL programs usually describe systems that run indefinitely (they define a relation between infinite sequences of inputs and outputs). However, some programs do not run indefinitely, they are detected in the second phase. For such programs, the interpreter stops at the end of the third phase, after having emitted the last reaction. A program whose execution is finished satisfies a property called *termination*.

Receiving a new event only provokes a modification of the memory. For this phase, the interpreter provides a communication with the outer world using an interface implemented in the Lisp part of the system. This interface uses graphical objects.

During the second phase, the dynamic semantics performs a normalization of the executed program and the memory in a rewriting system. The modification of the memory is the elaboration of the output reaction, while the modification of the program is the computation of a new state. This second phase is called the *normalization* or *execution* phase. We describe this phase more precisely in the next section.

The next phase is the communication of the computed reaction to the outer world. The meaningful data are the values of the global signals found in the memory. The interpreter also provides an interface using graphical objects for this phase.

The fourth phase finishes the computation of the new state. Three operations must be done:

1. Clean the values of local signals, so that they are not yet emitted for the following instant.
2. Set up the different temporal guards that appear in ESTEREL constructs. Conditionals triggered by the presence of signals in the coming instant are introduced in the program.
3. Perform some clean-up in the rewritten program, for example prune the parts where execution can no longer occur.

These operations are done by a simple tree traversal that performs yet another rewriting. This step is called an expansion step. The resulting program is ready for a new normalization. The computation for the next instant can proceed as soon as a new input event arrives.

2.1.2. *The Normalization Phase*

The normalization phase uses a rewriting system to express the evolution of the memory and the computation performed during the execution. Each elementary rewriting corresponds to the execution of an elementary operation. After a rewriting, the computation continues with the resulting object program, also called the *resumption*, until no further rewriting is possible.

Thus, the normalization function is based on two partial functions, the *execution* function and the *termination* function. The execution function performs the rewritings. It takes as arguments the memory and the program and returns the memory and the program modified by one elementary rewriting, when such a rewriting is possible. The *termination* function detects programs in normal form, it takes as argument only a program and returns a value only when no further rewriting is possible. Thus, there exist no program for which both the execution function and the termination function have a value. However, there exist programs for which neither of these functions is defined. Such programs are erroneous; the corresponding error is called a *causal loop*.

Besides detecting the programs in normal form, the termination function computes whether the object program satisfies the `termination` property or not. As we already explained, this property controls the termination of the main loop of the interpreter. We also see later that the termination property helps to define the behavior of the `sequence` construct. The termination function is defined in the set `terminated`.

2.1.3. *The Execution Function*

The execution function is one of the two partial functions used in the normalization phase. This function expresses how the memory is modified, how the control is performed, and how the executed instruction is removed when an elementary operation is performed. This function is defined in the TYPOL set `exec` and is represented by judgements of the following form:

$$mem \vdash stat \overset{\texttt{exec}}{\Rightarrow} stat', mem'$$

The terms *stat* and *mem* are given as arguments, *stat* is an ESTEREL program to execute and *mem* is the memory describing the values of the free variables appearing in *stat*. The terms $stat'$ and mem' are returned by the function, they are the rewritten program and the modified memory.

We take a closer look at some rules from this set.

– Execution of an `assignment` statement. Let us consider the rule for the `assignment` statement:

$$\frac{mem(\rho,\sigma) \overset{\texttt{eval}}{\vdash} exp \rightarrow val \qquad \texttt{update}(\rho, x, val, \rho')}{mem(\rho,\sigma) \overset{\texttt{exec}}{\vdash} x := exp \Rightarrow \texttt{nothing}, mem(\rho', \sigma)}$$

The memory is divided in two parts for the variable memory and the signal memory. The execution of an `assignment` statement provokes a modification of the variable memory ρ into ρ'. The resumption is the blank statement `nothing`.

– Execution of the `present` statement. Here, we give one of the rules for the `present` statement:

$$\frac{\texttt{sig_presence}(\sigma, S, +_{,)}}{mem(\rho,\sigma) \overset{\texttt{exec}}{\vdash} \texttt{if}\ exp\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2 \Rightarrow stat_1, mem(\rho,\sigma)}$$

If the signal S is present (denoted by $+$), then the execution of this conditional corresponds to the execution of the first branch, with the same memory. This rule (in fact, not only this one) shows how a `present` statement alters the control flow of the execution.

– Execution of the `watching` statement. The watching statement is one of the ESTEREL constructs that implement temporal guards, i.e., constructs that allow the apparition of a signal to limit the time taken by operations. Let us consider the rule that defines its behavior:

$$\frac{mem \overset{\texttt{exec}}{\vdash} stat \Rightarrow stat', mem'}{mem \overset{\texttt{exec}}{\vdash} \texttt{do}\ stat\ \texttt{watching}\ S \Rightarrow \texttt{do}\ stat'\ \texttt{watching}\ S, mem'}$$

This rule shows that the "one step" execution of some ESTEREL constructs can be expressed directly from the execution of a subpart. It appears that the `watching` construct has no effect on the execution within an instant. In fact, the real behavior of this statement is described in the expansion phase (phase 4 of the main loop), by the rule:

$$\frac{\overset{\texttt{expanse}}{\vdash} stat \rightarrow stat'}{\overset{\texttt{expanse}}{\vdash} \texttt{do}\ stat\ \texttt{watching}\ S \rightarrow \texttt{present}\ S\ \texttt{else do}\ stat'\ \texttt{watching}\ S}$$

This means that in the coming instant, the expansion of the instruction *stat* will be executed only if S is not present.

– Execution of the `sequence`. The `sequence` construct has a behavior that ensures that the tail of a `sequence` is always executed after the beginning

is finished. One first expresses that executing a sequence is executing its beginning with the following rule:

$$\frac{\overset{\texttt{exec}}{mem \vdash stat_1 \Rightarrow stat'_1, mem'}}{\overset{\texttt{exec}}{mem \vdash stat_1; stat_2 \Rightarrow stat'_1; stat_2, mem'}}$$

Then one expresses that the tail can be executed if the head verifies the termination property:

$$\frac{\overset{\texttt{terminated}}{\vdash stat_1 \rightarrow \texttt{true}, \emptyset_{traps}} \qquad \overset{\texttt{exec}}{mem \vdash stat_2 \Rightarrow stat'_2, mem'}}{\overset{\texttt{exec}}{mem \vdash stat_1; stat_2 \Rightarrow stat'_2, mem'}}$$

– Execution of the `parallel`. In a way the execution of the `parallel` construct is very similar to the execution of the `sequence`. However, it is not necessary to wait until the head has been completely executed to execute the tail. No preference is given to either way and the two rules for the head and the tail are symmetric, and similar to the first rule of the `sequence`.

$$\frac{\overset{\texttt{exec}}{mem \vdash stat_1 \Rightarrow stat'_1, mem'}}{\overset{\texttt{exec}}{mem \vdash stat_1 \| stat_2 \Rightarrow stat'_1 \| stat_2, mem'}}$$

$$\frac{\overset{\texttt{exec}}{mem \vdash stat_2 \Rightarrow stat'_2, mem'}}{\overset{\texttt{exec}}{mem \vdash stat_1 \| stat_2 \Rightarrow stat_1 \| stat'_2, mem'}}$$

These two rules do not exclude each other. Each time a `parallel` construct is executed both rules can apply. Thus, the execution is not deterministic. The parallelism in the ESTEREL language comes directly from this nondeterminism in the interleaving of the elementary steps. The parallelism found in ESTEREL has the same properties as the parallelism that one can find in β-reduction for the λ-calculus.

Note that the execution of statements like `nothing` or `halt` is not defined. On the contrary, the termination function is defined for such statements. Rewriting a statement in `nothing` is virtually removing this statement from the program.

3. Tools for Visualizing Execution

The purpose of an execution visualization tool is to animate the program during the interpretation, using different colors or typefaces to express the current state of execution as in figure 1. Visualizing enhances debugging by helping the programmer to detect places where the execution behaves differently from expected.

Visualizing contains three problems. The first problem is to track correspondences between the resumptions of the rewritings and the original program, which is actually displayed on the screen. To solve this problem we use *multi-occurrences* as described in the next section. The second problem is to detect in the resumption the expression that are worth emphasizing in the program for the current state of execution. This problem can be solved systematically from the dynamic semantics. The third problem is to transform the computed data in an actual display of the

program. This problem is [illegible] of CENTAUR [paths] and will not be de[illegible]

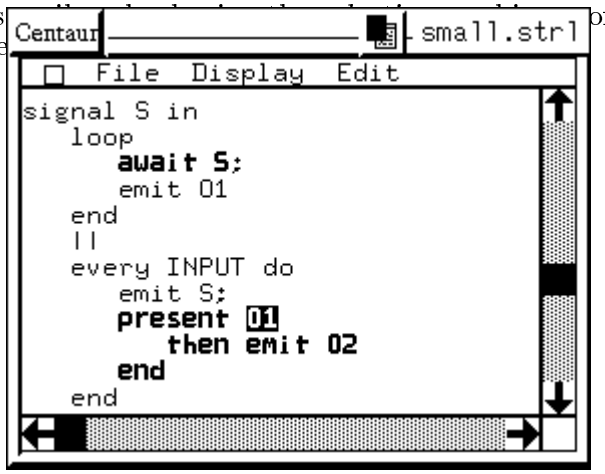

Figure 1 , Examples of execution points (in bold face) and of a signal blocked in read access (in reverse video).

3.1. *Subject Tracking*

We use *occurrences* and *multi-occurrences* to designate sub-expressions of a tree. Occurrences are strings of navigation commands that enable us to express the position of an expression in the tree. Multi-occurrences are used when one want to express that an expression is not a sub-expression of the tree, but that it shares sub-expressions with this tree. The expressions that are emphasized in figure 1 are designated with multi-occurrences. During the rewritings in the dynamic semantics the expressions are given multi-occurrences. When a term t_1 with the multi-occurrence m_1 is rewritten in a term t_2, one computes a multi-occurrence m_2 to go with t_2 that expresses what sub-expressions of t_2 come from t_1.

example: if t_1 is rewritten in t_2 where these terms have the following values:

$$t_1 = \texttt{present S then emit O end; emit P}$$
$$t_2 = \texttt{emit O; emit P}$$

then the multi-occurrence m_2 associated with t_2 will have the following value:

$$m_2 = u[s(s(m_1, 1), 2), s(m_1, 2)]$$

where m_1 is the multi-occurrence associated with t_1, to express that the first son of t_2 is a sub-expression of m_1 and give its place in m_1 and do the same for its second son.

In a rule, the variable *subject* gives the multi-occurrence associated to the expression that appears to the right of the turnstile ($\vdash$) in the conclusion of the rule. Using this feature we can define a `subject` function that returns the multi-occurrence associated to any expression. This rule is defined using the following axiom:

$$\texttt{subject} \vdash exp \rightarrow subject$$

3.2. *Execution Observation*

Now, we describe how we detect the interesting expressions (and the corresponding multi-occurrences) in the resumption of the rewritings. This work is performed at two moments. The first one is between each call to the execution function, in the normalization phase. The second one is during the expansion phase. The observation is performed by two functions that return multi-occurrences. We

show that the description of these functions can be systematically derived from the dynamic semantics.

3.2.1. *Generalized Axioms*

In TYPOL, every set of rules defines a function or a property. A *generalized axiom* is a rule which expresses this function or property on a construct without any recursive call for the same property on subterms of this construct. The following rule is a generalized axiom:

$$\frac{\rho, \sigma \overset{\texttt{eval}}{\vdash} exp \rightarrow val \qquad \texttt{update}(\rho, x, val, \rho')}{mem(\rho, \sigma) \overset{\texttt{exec}}{\vdash} x := exp \Rightarrow \texttt{nothing}, mem(\rho', \sigma)}$$

Although it has premises, none of these premises state that the execution function is recursively called on a subterm.

The following rule is not a generalized axiom. The premise states that one has to execute an elementary rewriting in the body of the `loop` construct to execute an elementary rewriting in the entire construct:

$$\frac{mem \overset{\texttt{exec}}{\vdash} stat \Rightarrow stat', mem'}{mem \overset{\texttt{exec}}{\vdash} \texttt{loop}\ stat\ \texttt{end} \Rightarrow stat'; \texttt{loop}\ stat\ \texttt{end}, mem'}$$

The generalized axioms are the rules that express a property on the constructs that are elementary relative to this property. We shall say that the rules that are not generalized axioms are *recursive* rules.

3.2.2. *Observation of the Normalization Phase*

We have explained above that every rewriting that appears in the normalization phase corresponds to an elementary execution step. When stepping the execution we want to show the exact situation of the instructions that will be reduced in all the possible elementary steps. These points correspond to the expressions where a generalized axiom could be applied in any possible application of the execution function.

We also want to show the exact situation of all the points where no rewriting is possible. this gives a symmetric notion of *elementary execution suspensions*. These points correspond to the expression where a generalized axiom of the termination function expresses that the execution is suspended. At last, we want to express the temporary blocking of execution that come from a synchronization discipline on the access to signals.

With these three notions, we have a criterion to apply on the dynamic semantics that enables us to derive a TYPOL function that computes the corresponding sets of multi-occurrences. This resulting function is named the *front* function, it is described by judgements of the following type:

$$sigs \overset{\texttt{front}}{\vdash} stat \rightarrow triple(set_1, set_2, set_3)$$

The first parameter, $sigs$, is the set of all the signals which can still be emitted in the same instant — this set is used to detect the temporary blockings coming from the synchronization. The expression $stat$ is the statement that describes the state of execution. The returned triple contains three sets of multi-occurrences designating sub-expressions of the program. The first component set_1 designates

the instructions where an elementary step of execution can occur next, set_2 designates the expressions where the execution is blocked on synchronization, set_3 designates the points where the execution is suspended.

If we find a place where a generalized axiom from the set `exec` can be applied, the corresponding expression should be designated as an elementary execution step. The following rules are two generalized axioms from this set:

$$\frac{\overset{\texttt{eval}}{\rho, \sigma \vdash exp \rightarrow val} \qquad \texttt{update}(\rho, x, val, \rho')}{\overset{\texttt{exec}}{mem(\rho, \sigma) \vdash x := exp \Rightarrow \texttt{nothing}, mem(\rho', \sigma)}}$$

$$\frac{\overset{\texttt{eval}}{\rho, \sigma \vdash exp \rightarrow \texttt{true}}}{\overset{\texttt{exec}}{mem(\rho, \sigma) \vdash \texttt{if}\ exp\ \texttt{then}\ stat1\ \texttt{else}\ stat2\ \texttt{end} \Rightarrow stat1, mem(\rho, \sigma)}}$$

To these two rules correspond two axioms in the set `front`:

$$\overset{\texttt{front}}{sigs \vdash x := exp \rightarrow triple(\{subject\}, \emptyset, \emptyset)}$$

$$\overset{\texttt{front}}{sigs \vdash \texttt{if}\ exp\ \texttt{then}\ stat1\ \texttt{else}\ stat2\ \texttt{end} \rightarrow triple(\{subject\}, \emptyset, \emptyset)}$$

These rules state that the expressions affected by the elementary executions are designated by multi-occurrences appearing in the triple's first set.

If we find a place where a generalized axiom from the set defining the termination function can be applied and expresses that the execution is suspended, we must express this in the front function. The following rule from the set `terminated` is an example of this case:

$$\overset{\texttt{terminated}}{\vdash \texttt{halt} \rightarrow \texttt{false}, \emptyset_{traps}}$$

The corresponding `front` axiom is as follows:

$$\overset{\texttt{front}}{sigs \vdash \texttt{halt} \rightarrow triple(\emptyset, \emptyset, \{subject\})}$$

Here the corresponding multi-occurrence is kept in the triple's third set.

Execution and termination are symmetric; recursive rules of the set `exec` correspond to recursive rules of the set `terminated`. We provide corresponding recursive rules for the set `front` too. These rules express that all the interesting expression found in a construct where a recursive of the execution function applies are the execution points that can be found in the subparts where a recursive call is possible. For example, we have two rules for the `parallel` construct in the set `exec`, they express that the execution can proceed in either branch:

$$\frac{\overset{\texttt{exec}}{mem \vdash stat_1 \Rightarrow stat_1', mem'}}{\overset{\texttt{exec}}{mem \vdash stat_1 \| stat_2 \Rightarrow stat_1' \| stat_2, mem'}}$$

$$\frac{mem \vdash stat_2 \overset{\texttt{exec}}{\Rightarrow} stat_2', mem'}{mem \vdash stat_1 \| stat_2 \overset{\texttt{exec}}{\Rightarrow} stat_1 \| stat_2', mem'}$$

The front function replaces the non-determinism of the execution function by an actual representation of the parallelism, by showing all the instructions that could be executed:

$$\frac{sigs \vdash stat_1 \overset{\texttt{front}}{\rightarrow} triple(set_1, set_1', set_1'') \qquad sigs \vdash stat_2 \overset{\texttt{front}}{\rightarrow} triple(set_2, set_2', set_2'')}{sigs \vdash stat_1 \| stat_2 \overset{\texttt{front}}{\rightarrow} triple(set_1 \cup set_2, set_1' \cup set_2', set_1'' \cup set_2'')}$$

The synchronization discipline in ESTEREL expresses that all reading access to a signal (corresponding, e.g., to the instruction `present`) must be performed after all writing access (corresponding to the instruction `emit`). To see why a program fails to execute, we need to see when this discipline alters the execution. There exists a function, the *potential* function that approximates the signals that can still possibly be emitted from the current execution state in the current transition. The signal memory cells are marked using this information, thus permitting to forbid any reading access when necessary. The function that enforces this discipline is the function that permits to read in the signal memory: `sig_presence`. A systematic way to detect the places where the access discipline alters the execution is to detect the execution rules that perform a call to this function:

$$\frac{\texttt{sig_presence}(\sigma, s, +, Value)}{mem(\rho, \sigma) \overset{\texttt{exec}}{\vdash} \texttt{present}\ s\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \Rightarrow stat_1, mem(\rho, \sigma)}$$

$$\frac{\texttt{sig_presence}(\sigma, s, -, Value)}{mem(\rho, \sigma) \overset{\texttt{exec}}{\vdash} \texttt{present}\ s\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \Rightarrow stat_2, mem(\rho, \sigma)}$$

In the front function, we use directly the result of the potential function to know whether the `sig_presence` function will block the execution or not.

$$\frac{s \in sigs \qquad \vdash s \overset{\texttt{subject}}{\rightarrow} s_subject}{sigs \vdash \texttt{present}\ s\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \overset{\texttt{front}}{\rightarrow} triple(\{subject\}, \{s_subject\}, \emptyset)}$$

$$\frac{s \notin sigs}{sigs \vdash \texttt{present}\ s\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} \overset{\texttt{front}}{\rightarrow} triple(\{subject\}, \emptyset, \emptyset)}$$

The call to `sig_presence` are always part of an generalized axiom for the execution. Thus, the expression that are detected as blocking the execution are always subparts of an expression detected as a possible elementary execution step.

3.2.3. *Observation of the Expansion Step*

The expansion step is not a normalization and we are not interested in the same phenomena. Here the rules of interest are not generalized axioms. The

interesting rules are the rules where a `present` construct has been introduced for a temporal guard. The rule for the `watching` construct is one such rule:

$$\frac{\texttt{expanse} \vdash \mathit{stat} \rightarrow \mathit{stat}'}{\texttt{expanse} \vdash \texttt{do}\ \mathit{stat}\ \texttt{watching}\ S \rightarrow \texttt{present}\ S\ \texttt{else do}\ \mathit{stat}'\ \texttt{watching}\ S}$$

The tool for the observation of the expansion step is defined in the set `show_xpans`. This set contains the following rule for the `watching` construct:

$$\frac{\texttt{show_xpans} \vdash \mathit{stat} \rightarrow \mathit{set} \qquad \texttt{subject} \vdash S \rightarrow \mathit{position}}{\texttt{show_xpans} \vdash \texttt{do}\ \mathit{stat}\ \texttt{watching}\ S \rightarrow \mathit{set} \cup \{\mathit{position}\}}$$

The introduction of a `present` construct corresponds to the raising of a temporal guard. We designate the signal on which the guard is raised.

The set `show_xpans` is designed to traverse exactly the part of the tree which is traversed by the set `expanse`. Thus, any rule from the set `expanse` containing a recursive call to the expansion function has a corresponding one in the set `show_xpans` that contains a recursive call to the expansion observation. This can also be done systematically.

4. Tools for Execution Control

A good debugger must also provide a way to execute slowly a program so that the programmer can observe precisely the key parts of his program. Ideally the programmer must be able to command the speed of execution at any time. A generic control is already given for the execution of TYPOL itself. It is possible to customize this generic control tool to give a control better suited to the ESTEREL execution model.

The controller is actually a finite state automaton, written in the ESTEREL language itself. It receives messages from all parts of the system, such as *this rule has been applied*, or *this button has been depressed*. The generic tool provides facilities to design a specific automaton for a language.

The basic events attached to the application of rules are of four kinds:

1. *Try.* A rule is tried in the computation.
2. *Prove.* The application of a rule has been proved.
3. *Back.* A new try is done for a rule.
4. *Fail.* The application of a rule has failed, i.e., this rule does not apply.

These events describe the computation as it is done in the Prolog interpreter. When a rule is applied it is possible to know the applied rule and the multi-occurrence designating the data it is applied on. This information helps to control the execution. For example, the multi-occurrence designating the subject data can be used to detect break-points in the program, although it is not done in this version of the interpreter. The output of this generic debugger is a collection of messages, such as *make this button appear* sent to the interface part of the system, or *continue the execution* sent to the logical kernel, i.e., the Prolog interpreter.

With this controller, we attach operations to certain points of the execution. For example, one says *When this rule is applied, flush the input event* (the external part of the input/output communication is performed this way); one can even have conditional operations, like *at this point, if there is a breakpoint on the subject of the rule, prompt the user for a command.* The control of the execution is designed

on top of the dynamic semantics, whose design is completely independent. One only needs to choose in the dynamic semantics the points where a control has to be added and to design the operations attached to this control, using all the data available in the computation.

For the ESTEREL interpreter we selected two points in the execution:

- The end of an instant.
- The execution of one rewriting in the normalization phase, i.e., the execution of an elementary instruction.

The first point is attached to the event *Prove* for the rule of the set `normal` that expresses the end of the normalization phase. The second point is attached to the event *Try* for the rule of the same set that expresses that a rewriting will be performed. The execution can then be broken down into steps, going from one of these points to another one. The interpreter provides a tool to express different commands such as:

- Execute the next elementary instruction and stop (command `Instruction`).
- Execute the next instant and stop (command `Instant`).

The generic controller provides other commands, that we keep in our controller:

- Abort as soon as possible (command `Abort`).
- Stop as soon as possible (command `Break`).
- Go without caring about instants or elementary executions or any similar event (command `Go`).

All these commands are grouped in a command box where some options appear only when the controller prompts the user for an order. The options *Break* and *Abort* are always available.

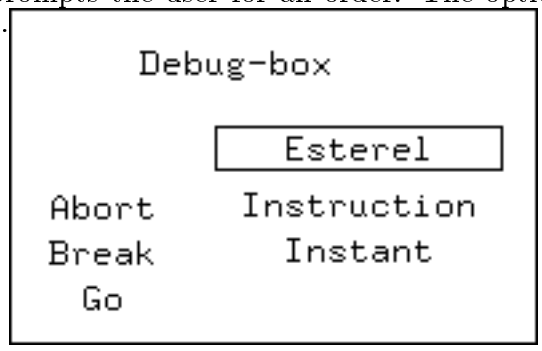

Figure 2, The controller's command box

5. Further Developments

This interpreter is a first step toward a complete debugger for the parallel language ESTEREL. Earlier experiments like [ml] only dealt with sequential languages. The treatment of parallelism introduces a new style of specification, making extensive use of rewriting to describe dynamic semantics. We have shown that this new style of specification iss still within the scope of natural semantics.

We have also shown that visualizing the execution state requires non-trivial computations. We have sketched a methodology to extract from the dynamic semantics a tool that helps visualizing execution. However, this methodology was a first attempt at solving this kind of problem and we have only provided an *ad hoc* treatment for this particular language. Regardless, the existence of a stated and well understood criterion of observation entitles us to claim that the shown information is relevant.

Visualizing is one of many debugging tools that track a correspondence between the executed program and the term that represents the execution state. Other such tools would, for example, allow to set *breakpoints* in the program so

that the execution stops when *reaching* such points, or to access the value of local variables during the execution. In these examples, one must find the expressions that inherit the breakpoints in the current execution state or the expressions that represent the local variables.

The CENTAUR system proves to be a good choice of a tool box for the development of an application like this interpreter. The semantics definition is kept in a pure form, free of implementation details. It is therefore easy to maintain and to check for correctness. The design of the man-machine interface is eased by the graphical tools which are already provided by the system. The result is an application which is easy to integrate in a more complete environment, including a type-checker and a compiler, since such tools can also be developed in the system.

Bibliography

[centaur] P. BORRAS ET AL., "CENTAUR: the system", *Proceedings of the ACM SIGSOFT'88: Third Symposium on Software Development Environments*, November 1988, Boston, USA. (Also appears as *INRIA Research Report no. 777.*)

[design] G. BERRY, G. GONTHIER, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation", *INRIA Research Report no. 842 (1988).* To appear in *Science of Computer Programming.*

[esterel] G. BERRY, L. COSSERAT, "The synchronous Programming Language ESTEREL and its Mathematical Semantics", *Seminar on Concurrency*, Springer Verlag LNCS 197, (1984).

[gfxobj] D. CLÉMENT, J. INCERPI, "Specifying the Behavior of Graphical Objects Using ESTEREL", *Proceedings of TAPSOFT'89 Colloquium on Current Issues in Programming Languages*, March 1989, Barcelona. (Also appears as *INRIA Research Report no. 836.*)

[ml] D. CLEMENT, J. DESPEYROUX, T. DESPEYROUX, G. KAHN, "A Simple Applicative Language: Mini-ML", *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986, Cambridge Massachusetts.

[natural semantics] G. KAHN, "Natural Semantics", *Programming of Future Generation Computers*, K. Fuchi, M. Nivat (Editors), Elsevier Science Publishers B.V. (North-Holland), 1988. (Also appears as *INRIA Research Report no. 601 (1987).*)

[paths] D. CLÉMENT, L. HASCOËT, "Centaur Paths: a structure to designate subtrees" *in Centaur 0.9 Documentation, Vol II, June 1989.*

[reactive] D. HAREL, A PNUELI, "On the development of Reactive Systems: Logic and Models of Concurrent Systems", *Proceedings of NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, NATO ASI Series F, vol. 13, Springer Verlag, (1985)

[typol] T. DESPEYROUX, "Typol, a formalism to implement Natural Semantics", *INRIA Technical Report no. 94, (1988).*