Implementation of an interpreter for a parallel language in CENTAUR

Yves Bertot

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tel (1) 39 63 55 11

Rapports de Recherche

N° 1076

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

IMPLEMENTATION OF AN INTERPRETER FOR A PARALLEL LANGUAGE IN CENTAUR

Yves BERTOT

Août 1989

RR_1076

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 several modules providing a graphical input-output interface, a graphical execution observation tool, and a simple execution controller.

Développement d'un Interprète pour un Langage Parallèle dans Centaur

résumé

Cet article présente le développement d'un interprète pour le langage parallèle ESTEREL dans le système CENTAUR. La sémantique dynamique du langage est décrite et complétée par plusieurs outils pour une interface d'entrée sortie graphique, l'observation graphique de l'exécution et un contrôleur de l'exécution élémentaire.

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 several modules providing a graphical input-output interface, a graphical execution observation tool, and a simple execution controller.

1. The Logical Kernel of the Interpreter

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. The command system for an airplane or the man-machine interface for an interactive system like CENTAUR [gfxobj] are two examples of such reactive systems.

The interpreter that we are going to describe uses a semantic definition provided by the designers of ESTEREL. Here, we shall focus on the implementation of this semantic definition within CENTAUR [centaur], using the TYPOL formalism [typol]. TYPOL permits the coding of a system of inference rules into an executable form. The basic units of a TYPOL specification are inference rules grouped in modules called *sets*. Thus, the semantic definition corresponds to a collection of TYPOL sets of rules. During our discussion, we shall always call this semantic definition the *dynamic semantics* of ESTEREL. The interested reader should refer to [esterel] for a detailed description of the language.

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 and the relation between the execution of an ESTEREL program and the normalization of this program in the rewriting system.

1.1. *The* ESTEREL *Language*

In [esterel], 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 that constitutes the central and most difficult part of reactive systems. ESTEREL 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 encapsulated in abstract data type facilities.

The basic concepts of the language are the synchrony hypothesis, parallelism, and the broadcast signal communication. The synchrony hypothesis is based on the assumption that each reaction to an input is instantaneous; that is, the underlying execution machine takes no time to execute the operations involved in instruction sequencing, process handling, interprocess communication, and basic data handling. The reception of an input event and the emission of the corresponding reaction define an ESTEREL *instant*.

The parallelism is used to enhance the modularity of the program. It gives an opportunity to design complicated reactive systems by breaking them down into simpler ones. This notion of parallelism goes hand in hand with a way to communicate information between parallel processes: the broadcast signal communication. Metaphorically, when one of the processes in parallel wants to emit an output, it does not have to care about the receiver; it just emits a signal knowing that it will be simultaneously received and treated by every other process that might care.

A signal can carry two types of information. One is a presence information (a signal may be present or absent); the other is a carried value which can be of any declared type. Some signals, called pure signals, do not carry any value. Signals may also be either global or local. Global signals are used to communicate with the outer world; local signals are used for the communication between different subparts of a program. Signals can be emitted, tested for presence, and consulted for the carried value (consulting the value of a signal can be asynchronous with the emission, i.e., one reads the value which was carried by the signal when it was last emitted).

Semantically, 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 a value. This relation can also be easily represented by a finite state automaton, receiving an input event, changing its state, and emitting a reaction. The automaton is deterministic, in spite of the parallelism used in ESTEREL, which inherently introduces some non-determinism in the internal execution. The non-determinism disappears as soon as one only looks at the input and the output reaction.

1.2. *The Dynamic Semantics of* ESTEREL

We present here the main path of execution for an ESTEREL program in the dynamic semantics. The first step is to coerce the different data in an executable form. An ESTEREL program consists of two parts. One is a declaration part, which gives global types, signals, and external procedures or functions; the other is an execution part which contains the instructions describing the "behavior" of the program. The main features of the coercion are the translation of the execution part into a slightly extended version of the ESTEREL language, the creation of a memory which corresponds to the signals and variables found in the declaration part, and the building of communication facilities between the interpreter and the outer world for the global signals. The coercion of the execution part

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

is necessary because one needs, for example, to be able to attach the current value of local signals and variables to their declarations (see the operator $\square$ in [esterel]). This coercion is performed by the procedure defined in the TYPOL set `coerce`. The treatment of the declaration part which involves the creation of a memory and the building of communication facilities is described in the TYPOL set `decls`.

The data manipulated by the dynamic semantics are the (coerced) 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. At the beginning and at the end of each computation for an instant, communications with the outer world are performed through the input/output handler. This handler is a symbolic name, and we shall see in the section dealing with the communication facilities how this handler is actually used (see p. 8).

1.2.1. *The Main Loop*

The main body of the execution is a loop where each pass corresponds to an ESTEREL 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. expansion of the program to be ready for the next instant.

In most cases, this loop is infinite since it makes sense to write ESTEREL programs describing systems that work indefinitely (for example, an elevator controller should never stop). For the programs that do not work indefinitely, it is possible to detect that the execution of a program is finished at the end of the second phase. In such cases, 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*.

The phase of reception of a new event provokes only a modification of the memory. For this phase an interface with the outer world is necessary. In CENTAUR, this interface is implemented by the Lisp part of the system. A complete description of the tools permitting this communication is given below in the section dealing with the input/output of the ESTEREL program (p. 8).

During the second phase, the dynamic semantics perform a normalization of the executed program in a rewriting system, together with a modification of the memory environment. The modification of the memory environment corresponds to the elaboration of the reaction, while the modification of the program corresponds to the computation of the new state of the corresponding automaton. This second phase is called the *normalization* or *execution* phase. It is described in the set `normal`. We give a more precise description of this phase in the next section.

The next phase is the communication of the computed reaction to the outer world. Here again an interface with the outer world is necessary. The meaningful data to communicate are the values of the global signals that are kept in the memory environment. The details of this communication are explained below (p. 8).

The fourth phase prepares the program that appears in the result of a reaction for the next instant. Three things must be done:

1. Clean the values of local signals, so that they are not yet emitted for the following instant.
2. Prepare the different temporal guards that appear in ESTEREL constructs. This preparation corresponds to the introduction of conditionals triggered by the presence of signals in the next instant (see p. 5).
3. Perform some clean-up in the program, for example prune the parts where no execution can occur.

This can be done by a simple tree traversal that computes a new version of the program in which certain constructs have been removed or replaced by others. This step is called an expansion step. It is described in the TYPOL set `expanse`. The resulting program is ready for a new normalization and the computation for the next instant can proceed as soon as a new input event is received.

1.2.2. *The Normalization Phase*

The normalization phase uses a rewriting system to express the evolution of the memory and of the computations to be performed during the execution. Each 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 rewriting, when 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 are defined. There exist programs for which neither of these functions is defined. Such programs are incorrect; the corresponding error is called a *causal loop*.

Another class of wrong programs is the class of programs for which the normalization phase never ends. The execution of most of the instructions in a program makes the size of the program decrease. However, the `loop` construct provokes an increasing of the program size since it rewrites itself as the sequence of its body and a copy of itself, each time it is executed. If the copy of the loop can also be executed in the same normalization phase, the interpreter enters an infinite loop which prevents the normalization phase to end. This problem is called an *instantaneous loop* and it is solved by marking the copy of the `loop` construct when it is executed, and by detecting an error when one has to execute a marked `loop` construct. The marks are removed during the expansion phase so that the `loop` constructs are clean at the beginning of the next normalization phase.

Besides detecting the programs in normal form, the termination function returns a boolean value that expresses whether the object program satisfies the `termination` property or not. As we already explained, this property controls the termination of the main loop of execution: when the boolean value is `true`, the computation ends after the emission of the computed reaction; when the value is `false`, the computation continues for the next instant with a new unfolding of the main

loop. We also see later (p. 6) that the termination property helps to define the behavior of the sequence construct. The termination function is described in the set `terminated`.

1.2.3. *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. In the modified memory, one finds the modifications described by the executed elementary operation; in the rewritten program, the corresponding instruction is removed.

The execution function is described in the TYPOL set **exec** and is represented by judgements of the following form:

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

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`:

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

  Note that 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 conditional statement. Here, we give one of the rules for the `if` statement:

$$\frac{\overset{\textbf{eval}}{\rho, \sigma \vdash exp \rightarrow \text{“}true\text{”}}}{\overset{\textbf{exec}}{mem(\rho, \sigma) \vdash \texttt{if}\ exp\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2 \Rightarrow stat_1, mem(\rho, \sigma)}}$$

  If the condition of the statement evaluates to “*true*” 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 conditional statement alters the control flow of the execution.

- Execution of the `watching` statement. The watching statement is a watchdog, i.e., a construct that allows the apparition of a signal to limit the time taken by operations. Let us consider the rule that defines its behavior:

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

  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 behavior of this statement is described in the expansion phase (phase 4 of the main loop), by the rule:

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

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

- Execution of the **sequence**. The **sequence** construct has a behavior which ensures that the tail of a **sequence** is always executed after the beginning. One first expresses that executing a sequence is executing its beginning with the following rule:

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

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

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

- Execution of the **parallel**. In a way the execution of the **parallel** construct is very similar to the execution of the **sequence**. Only, 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**.

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

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

This two rules do not exclude each other. Each time a **parallel** construct is executed both rules can apply. This gives a kind of non-determinism in interleaving the elementary executions in both branches that exactly expresses parallelism.

Note that the execution of the **nothing** statement is not defined. On the contrary, the termination function is defined for such a statement. The same goes for the **halt** statement. The designers of the language say that these statements cannot *act*. To rewrite a statement in **nothing** corresponds virtually to remove this statement from the program.

The execution function uses two important auxiliary functions. The first one is the termination function, this function detects the parts verifying the termination property and that can be skipped

when embedded in a **sequence** construct. The second auxiliary function enforces a discipline concerning the read/write access to the signal memory. This function is called the *potential* function. The read/write access discipline is expressed in the following sentence: *One should never read a signal if there still can be an emission of that signal in the same instant* (i.e., the same normalization phase see [esterel]). The potential function gives an approximation of the signals which can still be emitted by an instruction. This function performs a simple traversal of the program, as defined in the TYPOL set **potential**.

1.2.4. *Memory Management*

There are two kinds of memory in the ESTEREL semantics. One is for classical variables like those found in usual imperative languages (PASCAL, C). This memory is implemented by a structure of type **env**, represented by the greek letter ρ in this paper, and three tools are defined to read, write, and free a cell of this memory (the allocation of a cell is implicit). These tools are described in the sets **getenv**, **update**, and **remove_link**.

An other kind of memory is provided for the signals. As we mentioned above, a special access discipline works on this second memory. This discipline is enforced by putting a tag on the memory cells to indicate whether they are readable or not. To set these tags, one needs to know which signals can still be emitted within the same instant. This information is computed by the potential function between the rewritings for the global signals and each time the execution enters the scope of the declaration for local signals. The signal memory is implemented by a structure of type **signal_env**, and it is represented by the greek letter σ in this paper. We designed three tools for read/write access and freeing of cells for this signal memory as we did for the classical memory. These tools are described in the sets **sig_presence**, **add_signal**, and **remove_s_link**.

The current value of local variables or signals is attached to their declarations. A local variable declaration is an ESTEREL construct linking a variable and a body which is an ESTEREL statement where the declared variable may occur free. The ESTEREL language has been slightly extended with closing constructs so that a memory can be attached to the statement to keep the new value of the local identifier after a rewriting. When the execution function is called with a memory and a closure corresponding to a local variable declaration, the memory found in the closure is added to the global memory, and one executes the body in this extended memory. This execution returns a rewritten body and a new memory. The part of the returned memory that corresponds to the local variable is removed from the global memory and a new closure is constructed with the rewritten body and a memory describing the value of the local variable. Thus, the memory corresponding to the local variables is allocated only when the execution enters the scope of these local variables. The big advantage of this method is that the treatement of local variables and global variables is exactly the same when one executes the body. This method corresponds to the use of the operator $\square$ in [esterel]. This is shown in the TYPOL rule for the local variable closure:

$$\frac{mem(\rho \cup \{x = val\}, \sigma) \overset{\textbf{exec}}{\vdash} stat \Rightarrow stat', mem(\rho' \cup \{x = val'\}, \sigma')}{mem(\rho, \sigma) \overset{\textbf{exec}}{\vdash} [x = val, \textbf{var } x \textbf{ in } stat \textbf{ end}] \Rightarrow [x = val', \textbf{var } x \textbf{ in } stat' \textbf{ end}], mem(\rho', \sigma')}$$

Here the variable memory ρ contains an extra cell for the variable x only during the execution

of the body *stat*. After the rewriting, the new value of the variable *val'* is kept in the returned variable declaration construct, instead of the global memory. The same phenomenon occurs for local signals.

2. Communication Facilities

The signals declared in the global declarations of an ESTEREL module define the communication with the outer world. The interpreter must provide communication facilities for these signals. These communication facilities are very implementation dependent, and it makes sense to free the semantic definition of the programming details involved in their management. We achieve this goal with an interface based on a precise definition of the objects involved in this communication. These objects are described below:

- *channels*. To each signal is associated a channel of communication. This channel is used for communicating the signal's value. A limited set of operations is provided to work on these channels. These are performed through a protocol procedure: `channel_operation`.
- a *handler*. This object is associated to the set of all channels for a given ESTEREL program. A specific handler is provided for every interpretation of an ESTEREL program. It provides protocols for opening new channels and communicating through these channels. It can also be used for some interface parts which are not directly related to the ESTEREL semantics (e.g., visualizing and controlling the execution). The basic operations are related to set manipulations on the set of channels and to operations performed on all the channels at a time.

From a practical point of view, these objects permit a separation between the task performed in the logical part of the CENTAUR system and its interface tools. They enable us to code the dynamic semantics without making any assumption on the interface provided for the interpreter. Conversely, the interface can be developed without caring about the logical details in the semantics of the language.

We shall now describe the way these objects are used both inside the dynamic semantics of the ESTEREL language and in the tools provided for the interface. We then show how the interface can easily be adapted to different uses of the interpreter.

2.1. *The* TYPOL *Part of the Interface*

Every language provides a protocol for input/output facilities. This protocol usually contains four parts: the initialization of the objects used in the communication, the input of values, the output, and the freeing of the resources.

In the dynamic semantics of ESTEREL, the initializing the communication channels is associated with the treatment of global signal declarations (recall that the local signals only describe communication between processes inside the program; they are not used for input/output), while receiving and emitting signals are performed in the main loop of the dynamic semantics. The closing of channels is not treated in ESTEREL, it is implicitly peformed by the interpreter when the

dynamic semantics detects that the program is finished.

2.1.1. *Declaration of the Channels*

The interpretation of an ESTEREL program begins with the elaboration of the global declarations. This elaboration has two purposes: one is to prepare the memory environment of the interpreter by allocating memory cells for all the global signals, the second is to initialize the communication channels. The information associated to a channel consist of three data:

- a mode, which can have three values: *input*, *output* and *inputoutput*.
- a symbolic name, which is the name of the signal.
- a type (in this simple version we only treat pure and boolean signals).

This information must be given when we create the channel. The initialization occurs in the set `decls` where one traverses the declaration tree, calling the procedure `add_channel` whenever a signal is declared. This procedure has four arguments which are the communication handler in which the channel will be opened, the name of the signal for which a channel is created (for later reference), the communication mode, and the type. The result of this call is not to create the channel but to collect data for the configuration of all channels. At the same time, a memory cell is being allocated for each signal. The actual creation of all channels is done once and for all at the end of the declaration tree traversal, with a call to `config_channel_set`. When communication occurs we find in the memory the names of the signals, this enables us to name the channels used in the communication.We return to this in the next section.

This two-step organization allows us to consider the communication tool as a single object, the *handler*, which is constructed only when all the information is available. Below, we see how to use this organization to provide a graphical interface (see p. 10).

2.1.2. *Reception and Emission of Events*

In the dynamic semantics, the basis of any communication is the memory and the handler. All individual operations on channels are performed through the `channel_operation` procedure. This procedure has four arguments: a symbolic name for the handler, a name of signal used as a symbolic name for the channel, a symbolic name for the operation to perform, and a variable or a value to communicate. When the operation is an emission, the fourth argument is the emitted value; when it is a reception, the fourth argument is a variable which receives the value.

Receiving events appears in the first phase of the dynamic semantics' main loop. The symbolic names for the operations performed during the reception of a signal are `ask-state` to know the signal's presence state or `ask-bool-val` to know its boolean value (remember that this simple interpreter only works with pure or boolean signals). The operation for the reception of the value is only performed when the signal is present.

Like receiving an input event, emitting an output event is performed in the main loop. It is not directly related to the execution of the `emit` statements appearing in the program. If this were the case, the ESTEREL language would not be deterministic, since one could distinguish two possible executions by the ordering of the individual emissions. On the contrary, the output event must

be taken as a collection of present signals with their values, without caring about the order of the emissions[2]. Emission is performed at the end of the instant, when all the corresponding information is collected. This is the third phase of the main loop. The emission corresponds to a mapping of the memory into the output channels. Here the operations are `send-state` to send the signals' presence state and `send-bool-val` to send the values of boolean signals. The communication of the boolean value is performed only if the signal is present.

2.2. *The User Interface Point of View*

The idea to have a handler for the input/output communication of the interpreted language is not specific to ESTEREL. In CENTAUR, the communication between the logical part, the Prolog[3] kernel, and the interface part of the system, implemented in LE-LISP, is already based on manipulations performed on a Lisp structure, which is named a `typol-object`. This Lisp structure is a record containing the tree on which the computation is done and an association list called a *property list*. It is possible to get or set the value associated to a symbolic key in this property list. Most of the communication operations use this property list to keep values or structures which are meaningful during the interpretation of a program. The Lisp functions that perform these operation always receive as argument the name of a variable whose value is the `typol-object` associated to the execution. This variable name is the symbolic name used to refer to the handler in the dynamic semantics.

2.2.1. *Construction of the graphical objects*

We must provide Lisp counterparts to the procedures used in the dynamic semantics to initialize the communication channels. These procedures are `add_channel` to collect the information relative to a signal, and `config_channel_set` to trigger the actual construction of an object grouping all the channels, once they have all been declared. The corresponding Lisp functions are `add-channel` and `config-channel-set`. When a signal is declared, we keep the corresponding information in a structure which contains its name, its type, and its mode (i.e., `input`, `output`, or `inputoutput`). We name such a structure a channel. In all channels, we add structures to display the value of these signals; in the channels used for `input` or `inputoutput` signals we add extra structures (check-boxes) to receive the values of these signals from the user. The channels corresponding to all the signals are kept in a list and stored as a property of the `typol-object`. This operation is performed by the Lisp function `add-channel`.

After all the signals have been declared, the information that has been collected is used to construct two graphical objects: one for the input interface, the other for the output. This operation is performed by the Lisp function `config-channels`. This function works on the list of channels constructed by `add-channel`. The input graphical object groups the signals' name with the check-boxes used to input their values. The output graphical object groups the structures to show the

[2] If a signal carries a value, and more than one emission of this signal in a single instant is allowed, the language requests an associative and commutative function to be provided to combine the different carried values. This is another way to protect the determinism of the execution.

[3] MU-prolog of Melbourne University.

values of the signals.

2.2.2. *The Management of Input Events*

The reception of an input event works in two stages. The first appears in the dynamic semantics and corresponds to reading the values of the signals, one by one. The second is implicit, the controller does the work. The effect is to read and save the values of the check boxes found in the channels for input or inputoutput signals. The check box is then cleared, ready for the next event. This operation is performed by the function `{interp}:flush-input`. Another operation performed by this function is to flush the input signals' values directly towards the output graphical object, since these values can not be altered by the execution. This function is called at the beginning of each instant by the control mechanism that we describe below (see p. 17).

The reading of the value of a signal is triggered in the dynamic semantics by a call to the procedure `channel_operation` with the operation symbolic names `ask-state` and `ask-bool-val`. These two names are actually the names of two Lisp functions. These functions find the channel associated to a signal name and return the value which has been saved by the procedure `{interp}:flush-input`.

2.2.3. *The Display of the Reaction*

The meaningful data for the output is found in the global memory after the computation of an instant. A natural way to display the reaction is to take a snapshot of this memory. This method has the drawback, that it freezes the output. One would like to be able to attach actions, e. g., in the form of procedure calls, to the output. Therefore, we use the channel machinery to provide a more abstract and more versatile protocol for output emission. This solution is symmetric to the solution used for the input. Here again we use operation symbolic names of the `channel_operation` procedure to define two Lisp function: `send-state` and `send-bool-val`. These functions write in the channels the values of the signals sent from the dynamic semantics. After all the channels have been updated, the values are actually put in the graphical structures used for the display. The modification is flushed to the screen by a call of the function `{interp}:flush-output` (see below the section on the execution control, p. 17).

Notice that the reception and the emission of signals are exactly symmetrical. The functions `{interp}:flush-input`, `ask-state`, and `ask-bool-val` correspond respectively to the functions `{interp}:flush-output`, `send-state`, and `send-bool-val`.

2.3. *Other solutions*

The use of an abstract protocol based on channels makes it easy to change the input/output interface of the interpreter. We give in this section the example of an interface using only file input/output. It is likewise possible to use procedure calls for communicating with other ESTEREL debugging tools.

To provide a file input/output machinery to the interpreter we only have to redefine the two functions `flush-input` and `flush-output`. Let us suppose that the input events are written in a file in the form of lists whose elements are the name of signal or the pair of a name and a boolean value for boolean signals. The `{interp}:flush-input` function only has to read such a list in the

file and update the value field in the channel structures. The {interp}:flush-output function only has to traverse the list of channels associated to the execution and to write in a file the name of the signal when it is emitted, with its value when necessary. The other functions which actually perform the communication with the dynamic semantics need not be changed.

3. Tools for Visualizing Execution

The purpose of an execution visualization tool is to show in the original program the parts which are involved in the execution. Such a tool helps to understand the parallelism and the sequencing along the different parallel branches, these concepts are easier to grasp than the more abstract notions of normalization and termination. Thus, visualizing helps debugging by providing the programmer facilities to detect places where the execution behaves differently from expected. Such a tool refers to an intuitive semantics of the language that needs to be precised. Unlike the communication facilities, it is not absolutely necessary to the execution of the dynamic semantics; it only provides a way to observe the execution.

The visualization tool consists of three parts. The first part is a post-processor for the TYPOL compiler that provides a way to designate sub-expressions in a tree. This part is not described here, but we explain in the next section the features that are provided by this tool among them structures we call *multi-occurrences*. The second part consists of observation tools written in TYPOL and integrated in the dynamic semantics. These programs observe the data manipulated by the dynamic semantics and return multi-occurrences designating sub-expressions of the executed program and describing where the execution is currently taking place. The third part is a collection of LE-LISP functions that display a meaningful animation of the executed program.

3.1. *Subject Tracking*

We use *multi-occurrences* to designate sub-expressions of a tree. These multi-occurrences are structures describing navigation paths. The sub-expressions are found by applying the multi-occurrence on the studied tree.

In a rule, the variable *subject* has an special meaning. Its value is a multi-occurrence designating the sub-expression represented by the subject of the rule. The subject of a rule is the first parameter to the right of the turnstile in the rule's conclusion (see [natural semantics]). Using this feature, we can define a function `subject` that returns the multi-occurrence designating the sub-expression corresponding to its first argument. This function is defined by the following axiom:

subject
$\vdash exp \rightarrow subject$

The observation tools written in TYPOL and integrated in the dynamic semantics are based on manipulating the values obtained through this variable *subject*.

3.2. *Execution Observation*

Observation is based on two tools. The first one works on the resumption between the rewritings, during the normalization phase. The second one works on the program manipulated in the expansion phase. In each tool we want to detect sub-expressions related to important phenomena

in the ESTEREL execution. Since this execution is entirely described in the dynamic semantics of ESTEREL, the observation is deduced from this semantics. In the following sections we show that this deduction is, up to a point, systematic. We first introduce a general notion of TYPOL execution which we then use to define precisely the phenomena that we observe in both tools.

3.2.1. *Generalized Axiom*

We introduce here a class of TYPOL rules which are useful to detect the elementary steps of a computation. This class is the class of *generalized axioms*. In TYPOL, every set of rules defines a function or a property. A generalized axiom is a rule which expresses the function or property on a construct without any recursive call to the same property on subterms of this construct. The following rule is a generalized axiom:

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

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 the body of the `loop` construct to execute the entire construct:

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

The generalized axioms are the rules that express the value of the defined function or property on the constructs which are elementary relative to this function or property.

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. The notion of an elementary step is very intuitive. We give here a precise definition to this notion. We also find a symmetric notion of *execution suspension point* which is very useful in understanding the execution, too. We add a third notion that complements the two previous ones by expressing the blockings that appear due to the signal memory access discipline.

These three notions correspond to criteria to be applied on the dynamic semantics of ESTEREL to derive a TYPOL function that computes the corresponding sets of sub-expressions. This function is named the *front* function, it is described by judgements of the following type:

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

The first parameter, *sigs*, is the set of all the signals which can still be emitted in the same instant —this set has been computed by the potential function, it helps to detect the read access blockings. The subject of the judgement, *stat*, is the statement in which the execution is analyzed.

The triple contains three sets of multi-occurrences designating sub-expressions of the program. The first component set_1 designates the instructions where an elementary execution can be performed next, set_2 designates the signals on which a forbidden read access is attempted, set_3 designates the points of execution suspension. As we see later, the multi-occurrences in set_2 only designate sub-expressions of expressions designated in set_1 since the memory access discipline adds a constraint on the execution steps. Also, the two sets set_1 and set_3 are disjoint.

To define the notion of elementary execution, we use the notion of generalized axioms that we defined above. We associate the notion of *elementary execution step* to the application of the generalized axioms found in the set **exec**. Our method is to designate the sub-expressions on which these axioms are applied. The following two rules are axioms and thus describe elementary executions:

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

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

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

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

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

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

To limit the execution observation to properties deduced from the execution function only gives a partial result, since this execution function is a partial function. A complete observation tool should also contain computations deduced from the termination function, since this function describes the complement of the execution function's domain. More intuitively, the termination property is important to observe since it expresses whether execution can progress in a part of the program or not. Here again, we are interested in associating sets of sub-expressions to the possible applications of generalized axioms. However, our analysis must be more precise. When the termination function is defined on a part of the program, it can have two meanings:

1. Execution is finished and one can skip to what follows. This means that execution behaves as if this part of the program did not exist (this part verifies the termination property). For such an expression, the termination function returns the boolean value **true**.
2. Nothing can be executed because a suspending statement is reached. For such an expression, the termination function returns the boolean value **false**.

We decide to designate only expressions corresponding to the second case. We call a *point of execution suspension* a sub-expression of the program where an axiom of the set **terminated** can be applied and returns the value **false**. The following rule, taken from the set **terminated**, is an axiom describing the termination property for the **halt** statement:

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

The corresponding **front** axiom is as follows:

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

Here the corresponding multi-occurrence is kept in the triple's third set. Thus we define an extension of the notion of elementary execution that we name an *execution point*. An execution point is either an elementary execution or a point of execution suspension. The sets designating the elementary execution steps and the points of execution suspension are disjoint since the execution function and the termination function have disjoint domains.

Because of the symmetry between execution and termination, the recursive rules of the set **exec** correspond exactly to the recursive rules of the set **terminated**. We provide such recursive rules for the observation function too. These rules express that all the execution points found in a non-elementary construct are the execution points found in those of its subparts which could be executed. For example, we have two rules for the **parallel** construct in the set **exec**, which express that the execution can proceed either in the first or the second branch:

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

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

The front function groups all the possibilities by studying both cases:

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

As we mentioned in the part on the normalization phase (p. 4), there are some parts where neither the execution function nor the termination function are defined. These correspond to execution points where the execution cannot proceed although the execution is not suspended. These points are blocked by the signal memory access discipline. A full observation of the execution must also detect these points. The enforcement of signal memory access discipline is performed through the calls of the `sig_presence` function. The value returned by this function contains the

value of the signal (presence and carried data) and the status of the memory. The status of the memory can have four different values: $+$, $-$, $\dagger$ and $\perp$. Only $+$, and $-$ correspond to cells that are readable, $\dagger$ and $\perp$ correspond to forbidden cells. We see how these values are used in the following two rules that describe an elementary execution testing for the presence of a signal:

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

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

There are no rules for the cases when the cell is not labelled by $+$ or $-$ but by $\dagger$ or $\perp$ In the front function, we provide the detection of this case with the two following rules:

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

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

We do not simulate the complete memory operations for the visualization tool, we check directly from the set of signals computed by the potential function. The designated expression for the read access blocking is the signal itself. This expression is a sub-expression of an the instruction designated as an elementary execution step in the same rule.

3.2.3. *Observation of the Expansion Step*

The expansion step cannot be viewed like a normalization and we are not interested in the same phenomena. Here the rules of interest are not especially related to the rules which contain a recursive call to the same function. The interesting rules are the rules where a **present** construct has been introduced to be executed in the next instant. The rule for the **watching** construct is one such rule:

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

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:

$$\frac{\overset{\texttt{show\_xpans}}{\vdash} stat \rightarrow set \qquad \overset{\texttt{subject}}{\vdash} S \rightarrow position}{\overset{\texttt{show\_xpans}}{\vdash} \texttt{do}\ stat\ \texttt{watching}\ S \rightarrow set \cup \{position\}}$$

The introduction of a **present** construct corresponds to the raising of a temporal guard. We prefer to designate the signal on which the guard is raised rather than the complete instruction, since it is more atomic.

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 correspondent in the set **show_xpans** that contain a recursive call to the expansion observation. This can also be done systematically.

3.3. *The Program Animation*

The data computed by the TYPOL part of the visualization tools are multi-occurrences which designate expressions in the original program. We show these expressions in different colors, using the selection machinery provided by the CENTAUR system. The TYPOL multi-occurrences must be translated into path structures, as defined in the CENTAUR manual[paths]. In this translation, a major problem occurs because it often happens that an executed instruction does not correspond directly to any sub-tree in the original program. Let us take the example of a **watching** construct:

```
do halt watching S
```

during the expansion phase it is transformed in:

```
present S else do halt waching S end
```

So that in the next normalization phase one has to execute a **present** statement. The question we have to solve is the following one: *Which subtree should we designate?* We consider that the test of presence on the signal is a part the **watching** construct's behavior, and we decide to designate this construct.

The multi-occurrence that designates the **present** statement expresses that this statement contains the sub-expression S and the sub-expression **do halt watching** S. Both expressions are sub-expressions of the watching construct, and the watching construct is the smaller instruction containing these expression. This property is very general, and can be verified for all the expansions of temporal guards. We decide to always show the smallest instruction containing all the parts designated by a multi-occurrence when this multi-occurrence represents an execution point (all the execution points should be instructions). However, when the given multi-occurrence does not forcibly represent an instruction (as it is the case for the blocking points) we prefer to show the expressions directly designated by the multi-occurrence. These two variants of translation are performed by two different Lisp functions.

The computed paths are then sent to structures indicating the color used for their display[4] and the program is redisplayed. The main functions, which are called directly by the TYPOL program, are **select-instruction** and **select-exp**. These functions work with the handler which is also used for the input/output communication.

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

[4] The selection structures which are provided in the CENTAUR system.

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 calls to `{interp}:flush-input` and `{interp}:flush-output` are 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:

- Go to the next instant, and stop.
- Go to the next elementary execution, and stop.

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

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

Except for the two last commands (stop and abort), all these commands are prompted to the user only when the execution is stopped by the controller. The two last commands are always available, they enable the user to interrupt the execution at any moment. In the graphical interface, the tool which provides these commands is a box where buttons appear or disappear depending when the commands are available.

5. Further Developements

This interpreter is bare. For an efficient debugger, one needs more than just visualization and this simple control. Other tools would allow to set break-points in the original program so that the execution stops when reaching such a point and to access the values of local variables and signals even between two elementary executions. But these tools generalize the existing ones, and use the same basic ideas. A tool for the manipulation of break-points can easily be constructed using the subject tracking, already developed for visualizing, and the customizable controller. A tool providing the possibility to access the value of the memory during the execution can be constructed with the subject tracking system. Thus, the four points on which we focused in this paper are the basic blocks with which a powerful debugger can be constructed.

The CENTAUR system proves to be a good choice of a tool box for the developement 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[5]. The integration can also be done with tools outside the CENTAUR system, since we provide a well defined input/output protocol for the interpreter.

[5] This work has been done by J. B. Saint from ENSMP-CMA at Sophia-Antipolis.

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*).

[esterel] G. BERRY, G. GONTHIER "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation", *INRIA Research Report no. 842 (1988).*

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

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

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

[natural semantics] G. KAHN "Natural Semantics", *INRIA Research Report no. 601 (1987).*

Appendix

We present here the main sets of ESTEREL's dynamic semantics. These dynamic semantics differ from the TYPOL code that we actually use in the interpreter only by syntactic conventions. As in the paper, the parts of the rules that represent ESTEREL terms are written in the concrete syntax of the language. Inside a set the name of the defined judgement is implicit. Some rules that appear in the code are not presented here. Some removed rules correspond to ESTEREL constructs that are not basic, others to the treatement of errors that normally do not appear in the semantics (the semantics only express the behavior of correct programs).

The set **instant** describes the main loop of the execution. The rule "loop" corresponds to the normal case. The premise **new_event** expresses the reception of an input event. The premise **normal** expresses the normalization phase and the emission of the output. The premise **expanse** expresses the expansion phase. The last premise expresses that the loop continues. This rule is selected if the boolean returned by the last normalization expresses that the studied program does not verify the *termination property*. The rule "end" expresses that the execution is finished when the studied program verifies the termination property.

set instant **is**

loop: $$\frac{\textbf{new\_event}(Handler, \sigma, \sigma') \quad < \rho \times \sigma', Handler >^{\textbf{normal}} \vdash stat \Rightarrow stat', B, Mem' \quad \vdash stat' \xrightarrow{\textbf{expanse}} stat'' \quad < Mem', Handler > \vdash stat'', B}{< \rho \times \sigma, Handler > \vdash stat, \text{false}}$$

end: $world \vdash stat$, true

end instant

The set normal describes the normalization phase. The rule "loop" expresses that the execution proceeds when it is possible. The premise **new_potentials** expresses that the signal memory must be updated with the signals' potentials, to enforce the read access discipline. The premise **exec** expresses that a rewriting is performed. The last premise expresses that the loop continues. The rule "end" expresses that the loop stops when the termination function is defined for the studied program. The boolean value returned by the termination function is returned for use in the rules from the previous set.

set normal **is**

loop: $$\frac{\overset{\textbf{new\_potentials}}{\sigma \vdash stat : \sigma'} \qquad \overset{\textbf{exec}}{\rho \times \sigma' \vdash stat \Rightarrow stat', \rho' \times \sigma''} \qquad < \rho' \times \sigma'', Handler > \vdash stat' \Rightarrow stat'', state}{< \rho \times \sigma, Handler > \vdash stat \Rightarrow stat'', state}$$

end: $$\frac{\overset{\textbf{terminated}}{\vdash stat \rightarrow B, \_}}{< \rho \times \sigma, Handler > \vdash stat \Rightarrow stat, B, \rho \times \sigma}$$

end normal

The set **exec** describes the rewritings performed in the normalization phase. The judgement **eval** expresses the evaluation of an expression, The judgement **update** expresses the modification of the memory for a variable. The judgement **add_signal** expresses the modification of the memory for a signal. The judgement **sig_presence** expresses the read access in the signal memory. The value + (resp. −) expresses that the signal can be read and is present (resp. absent). The judgement **mk_env** expresses the construction of a local memory corresponding to declarations of variables. The judgement **local_values** expresses the extraction of the local memory corresponding to a closure from the global memory. The judgement **unclose** expresses the merging of a local signal memory and a global memory, taking into account the potential of the local signals.

set exec is

$$\frac{\rho, \sigma \vdash exp \xrightarrow{\text{eval}} val \qquad \text{update}(\rho, x, val, \rho')}{\rho \times \sigma \vdash x := exp \Rightarrow \text{nothing}, \rho' \times \sigma}$$

$$\frac{\rho, \sigma \vdash expression \xrightarrow{\text{eval}} val \qquad \sigma \vdash S, val : \sigma' \ \text{add\_signal}}{\rho \times \sigma \vdash \text{emit } S(expression) \Rightarrow \text{nothing}, \rho \times \sigma'}$$

$$\frac{\vdash stat_1 \xrightarrow{\text{terminated}} \text{true}, \emptyset \qquad mem \vdash stat_2 \Rightarrow stat_2', mem'}{mem \vdash stat_1; stat_2 \Rightarrow stat_2', mem'}$$

$$\frac{mem \vdash stat_1 \Rightarrow stat_1', mem'}{mem \vdash stat_1; stat_2 \Rightarrow stat_1'; stat_2, mem'}$$

$$\frac{mem \vdash stat \Rightarrow stat', mem'}{mem \vdash \text{loop } stat \text{ end} \Rightarrow stat'; \text{loop } stat \text{ end}, mem'}$$

$$\frac{\rho, \sigma \vdash exp \xrightarrow{\text{eval}} \text{bool true}}{\rho \times \sigma \vdash \text{if } exp \text{ then } stat_1 \text{ else } stat_2 \text{ end} \Rightarrow stat_1, \rho \times \sigma}$$

$$\frac{\rho, \sigma \vdash exp \xrightarrow{\text{eval}} \text{bool false}}{\rho \times \sigma \vdash \text{if } exp \text{ then } stat_1 \text{ else } stat_2 \text{ end} \Rightarrow stat_2, \rho \times \sigma}$$

$$\frac{\text{sig\_presence}(\sigma, S, -, \_)}{\rho \times \sigma \vdash \text{present } S \text{ then } stat_1 \text{ else } stat_2 \text{ end} \Rightarrow stat_2, \rho \times \sigma}$$

$$\frac{\text{sig\_presence}(\sigma, S, +, \_)}{\rho \times \sigma \vdash \text{present } S \text{ then } stat_1 \text{ else } stat_2 \text{ end} \Rightarrow stat_1, \rho \times \sigma}$$

$$\frac{mem \vdash stat_1 \Rightarrow stat_1', mem'}{mem \vdash \text{do } stat_1 \text{ watching } S \Rightarrow \text{do } stat_1' \text{ watching } S, mem'}$$

$$\frac{mem \vdash stat_1 \Rightarrow stat_1', mem'}{mem \vdash stat_1 \parallel stat_2 \Rightarrow stat_1' \parallel stat_2, mem'}$$

$$\frac{mem \vdash stat_2 \Rightarrow stat_2', mem'}{mem \vdash stat_1 \parallel stat_2 \Rightarrow stat_1 \parallel stat_2', mem'}$$

$$\frac{mem \vdash stat \Rightarrow stat', mem'}{mem \vdash \text{trap } T \text{ in } stat \text{ end} \Rightarrow \text{trap } T \text{ in } stat' \text{ end}, mem'}$$

$$\frac{\emptyset, \rho, \sigma \vdash Declarations : \rho_l \xrightarrow{\text{mk\_env}} \qquad (\rho \cup \rho_l) \times \sigma \vdash stat \Rightarrow stat', \rho' \times \sigma' \qquad \rho' \vdash \rho_l : \rho_l', \rho'' \xrightarrow{\text{local\_values}}}{\rho \times \sigma \vdash \text{var } Declarations \text{ in } stat \text{ end} \Rightarrow [\rho_l', stat'], \rho'' \times \sigma'}$$

$$\frac{\rho \cup \rho_l \times \sigma \vdash stat \Rightarrow stat', \rho' \times \sigma' \qquad \rho' \vdash \rho_l : \rho_l', \rho'' \xrightarrow{\text{local\_values}}}{\rho \times \sigma \vdash [\rho_l, stat] \Rightarrow [\rho_l', stat'], \rho'' \times \sigma'}$$

$$\frac{\vdash stat : sigs, \_, \_ \xrightarrow{\text{potential}} \qquad \sigma, sigs \vdash \sigma_l : \sigma' \xrightarrow{\text{unclose}} \qquad \rho \times \sigma' \vdash stat \Rightarrow stat', \rho' \times \sigma'' \qquad \sigma'' \vdash \sigma_l : \sigma_l', \sigma''' \xrightarrow{\text{local\_values}}}{\rho \times \sigma \vdash [[\sigma_l, stat]] \Rightarrow [[\sigma_l', stat']], \rho' \times \sigma'''}$$

end exec

The set `potential` describes the computation of the signals' potentials. The returned values are the set of the signals that can still be emitted, a boolean value expressing whether a derivation of the studied program can verify termination property, and a set of exceptions that can be raised. The set `rm_declared` expresses that the signals declared in a local declaration hide the global signals having the same name in the scope of this declaration.

set potential **is**

$\vdash$ `nothing` : $\emptyset$, true, $\emptyset$

$\vdash$ `halt` : $\emptyset$, false, $\emptyset$

$\vdash X := exp$: $\emptyset$, true, $\emptyset$

$$\frac{\vdash stat_1 : S, \text{false}, T}{\vdash stat_1; stat_2 : S, \text{false}, T}$$

$$\frac{\vdash stat_1 : S_1, \text{true}, T_1 \qquad \vdash stat_2 : S_2, \beta, T_2}{\vdash stat_1; stat_2 : S_1 \cup S_2, \beta, T_1 \cup T_2}$$

$$\frac{\vdash stat : S, \beta, T}{\vdash \texttt{loop}\ stat\ \texttt{end} : S, \text{false}, T}$$

$$\frac{\vdash stat_1 : S_1, \beta_1, T_1 \qquad \vdash stat_2 : S_2, \beta_2, T_2}{\vdash \texttt{if}\ exp\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} : S_1 \cup S_2, \beta_1 \vee \beta_2, T_1 \cup T_2}$$

$$\frac{\vdash stat_1 : S_1, \beta_1, T_1 \qquad \vdash stat_2 : S_2, \beta_2, T_2}{\vdash \texttt{present}\ Sig\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end} : S_1 \cup S_2, \beta_1 \vee \beta_2, T_1 \cup T_2}$$

$$\frac{\vdash stat_1 : S_1, \beta_1, T_1 \qquad \vdash stat_2 : S_2, \beta_2, T_2}{\vdash stat_1 \| stat_2 : S_1 \cup S_2, \beta_1 \wedge \beta_2, T_1 \cup T_2}$$

$$\frac{\vdash stat : S, \text{true}, T}{\vdash \texttt{trap}\ tag\ \texttt{in}\ stat\ \texttt{end} : S, \text{true}, T - \{tag\}}$$

$$\frac{\vdash stat : S, \text{false}, T \qquad tag \in T}{\vdash \texttt{trap}\ tag\ \texttt{in}\ stat\ \texttt{end} : S, \text{true}, T - \{tag\}}$$

$$\frac{\vdash stat : S, \text{false}, T \qquad tag \notin T}{\vdash \texttt{trap}\ tag\ \texttt{in}\ stat\ \texttt{end} : S, \text{false}, T}$$

$\vdash$ `exit` T : $\emptyset$, false, $\{T\}$

$$\frac{\vdash stat : S, \beta, T}{\vdash \texttt{var}\ X\ \texttt{in}\ stat\ \texttt{end} : S, \beta, T}$$

$$\frac{\vdash stat : S, \beta, T}{\vdash [\rho_l, stat] : S, \beta, T}$$

$$\frac{\vdash stat : S, \beta, T \qquad \texttt{rm\_declared}(S, \sigma_l, S')}{\vdash [[\sigma_l, stat]] : S', \beta, T}$$

$\vdash$ `emit` Sig : $\{Sig\}$, true, $\emptyset$

$$\frac{\vdash stat : S, \beta, T}{\vdash \texttt{do}\ stat\ \texttt{watching}\ Sig : S, \beta, T}$$

end potential

The set **terminated** describes the termination function. The returned values are a boolean expressing the termination property and a set of exceptions raised in the studied program. This set of exceptions is used to decide the termination of the **trap** construct.

set terminated **is**

$\vdash$ **nothing** $\rightarrow$ true, $\emptyset$

$\vdash$ **halt** $\rightarrow$ false, $\emptyset$

$$\frac{\vdash stat_1 \rightarrow \text{false}, traps}{\vdash stat_1; stat_2 \rightarrow \text{false}, traps}$$

$$\frac{\vdash stat_1 \rightarrow \text{true}, \emptyset \qquad \vdash stat_2 \rightarrow \beta, traps}{\vdash stat_1; stat_2 \rightarrow \beta, traps}$$

$$\frac{\vdash stat \rightarrow \text{false}, traps}{\vdash \texttt{loop}\ stat\ \texttt{end} \rightarrow \text{false}, traps}$$

$$\frac{\vdash stat \rightarrow \beta, traps}{\vdash \texttt{do}\ stat\ \texttt{watching}\ Sig \rightarrow \beta, traps}$$

$$\frac{\vdash stat_1 \rightarrow \beta_1, traps_1 \qquad \vdash stat_2 \rightarrow \beta_2, traps_2}{\vdash stat_1 \parallel stat_2 \rightarrow \beta_1 \wedge \beta_2, traps_1 \cup traps_2}$$

$$\frac{\vdash stat \rightarrow \beta, \{T\}}{\vdash \texttt{trap}\ T\ \texttt{in}\ stat\ \texttt{end} \rightarrow \text{true}, \emptyset}$$

$$\frac{\vdash stat \rightarrow \text{true}, \emptyset}{\vdash \texttt{trap}\ T\ \texttt{in}\ stat\ \texttt{end} \rightarrow \text{true}, \emptyset}$$

$$\frac{\vdash stat \rightarrow \text{false}, traps \qquad traps \neq \{T\}}{\vdash \texttt{trap}\ T\ \texttt{in}\ stat\ \texttt{end} \rightarrow \text{false}, traps - \{T\}}$$

$\vdash$ **exit** $T \rightarrow$ false, $\{T\}$

$$\frac{\vdash stat \rightarrow \beta, traps}{\vdash [\rho_l, stat] \rightarrow \beta, traps}$$

$$\frac{\vdash stat \rightarrow \beta, traps}{\vdash \texttt{var}\ X\ \texttt{in}\ stat\ \texttt{end} \rightarrow \beta, traps}$$

$$\frac{\vdash stat \rightarrow \beta, traps}{\vdash [[\sigma_l, stat]] \rightarrow \beta, traps}$$

end terminated

The set **expanse** describes the expansion phase. This phase mainly expresses that the **watching** construct must be expanded. The judgement **bottom** expresses that the local signals must be marked as not yet emitted for the next normalization phase.

set expanse is

$\vdash$ halt $\rightarrow$ halt

$\vdash$ nothing $\rightarrow$ nothing

$\vdash X := exp \rightarrow X := exp$

$\vdash$ emit $S \rightarrow$ emit S

$\vdash$ exit $Tag \rightarrow$ exit Tag

$$\frac{\vdash stat_1 \rightarrow stat_1'}{\vdash stat_1; stat_2 \rightarrow stat_1'; stat_2'} \quad \textbf{provided} \quad \overset{\textbf{terminated}}{\vdash stat_1 \rightarrow \text{false}, \_}$$

$$\frac{\vdash stat_2 \rightarrow stat_2'}{\vdash stat_1; stat_2 \rightarrow stat_2'} \quad \textbf{provided} \quad \overset{\textbf{terminated}}{\vdash stat_1 \rightarrow \text{true}, \_}$$

$$\frac{\vdash stat \rightarrow stat'}{\vdash \texttt{loop } stat \texttt{ end} \rightarrow stat'; \texttt{loop } stat \texttt{ end}}$$

$\vdash$ if exp then $Stat_1$ else $Stat_2$ end $\rightarrow$ if exp then $Stat_1$ else $Stat_2$ end

$\vdash$ present exp then $Stat_1$ else $Stat_2$ end $\rightarrow$ present exp then $Stat_1$ else $Stat_2$ end

$$\frac{\vdash stat_1 \rightarrow stat_1' \qquad \vdash stat_2 \rightarrow stat_2'}{\vdash stat_1 \| stat_2 \rightarrow stat_1' \| stat_2'}$$

$$\frac{\vdash stat \rightarrow stat'}{\vdash \texttt{trap } excepts \texttt{ in } stat \texttt{ end} \rightarrow \texttt{trap } excepts \texttt{ in } stat' \texttt{ end}}$$

$$\frac{\vdash stat \rightarrow stat'}{\vdash \texttt{do } stat \texttt{ watching } S \rightarrow \texttt{present } S \texttt{ then nothing else do } stat' \texttt{ watching } S} \quad \textbf{provided} \quad \overset{\textbf{terminated}}{\vdash stat \rightarrow \text{false}, \_}$$

$\vdash$ do $stat$ watching $S \rightarrow$ nothing $\quad \textbf{provided} \quad \overset{\textbf{terminated}}{\vdash stat \rightarrow \text{true}, \_}$

$$\frac{\vdash stat \rightarrow stat'}{\vdash \texttt{var } decls \texttt{ in } stat \texttt{ end} \rightarrow \texttt{var } decls \texttt{ in } stat' \texttt{ end}}$$

$$\frac{\vdash stat \rightarrow stat'}{\vdash [\rho_l, stat] \rightarrow [\rho_l, stat']}$$

$$\frac{\vdash stat \rightarrow stat' \qquad \overset{\textbf{bottom}}{\vdash \sigma_l \rightarrow \sigma_l'}}{\vdash [[\sigma, stat]] \rightarrow [[\sigma_l', stat']]}$$

end expanse

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique