ELUDO:
The University of Ottawa LOTOS ToolKit

A guided tour

by Bernard Stepien, Jean Tourrilhes and Jacques Sincennes

ELUDO (Environnement LOTOS de l'Université dOttawa) is the University of Ottawa LOTOS Toolkit. It is the descendant of the first interpreter of the LOTOS Language ever written. The tool has two interfaces: one for text terminals and one for the X window system (this is called XELUDO). The goal of this paper is to show how easy it is to use the toolki with the various features it includes. The example used is a very simple POTS (Plain Old Telephone System) specification.

About LOTOS and the specification

LOTOS is a specification language standardized by the ISO (IS 8807, 1989). It was designed to describe telecommunication systems. The LOTOS language uses operators to specify the temporal orderings between individual actions and/or larger entities, which describe various alternative sequences of actions.

LOTOS expressions are called *behavior expressions.* We represent behavior expressions by capital letters such as **B** while *actions* are represented with lowercase letters such as **a**. For now, we take 'actions' and 'gates' to mean the same thing. Later, we shall see that gates are parts of actions.

The following are the main LOTOS operators:

a; B: the action prefix operator ";" means that action **a** precedes behavior **B**.

B1 [] B2: the choice operator **[]** means that an action can be obtained either from behavior expression **B1** or from **B2** exclusively.

B1 || B2: the full synchronization parallelism operator **||** means that a common action from behavior expressions **B1** and **B2** has to exist in order for the system to proceed. If the action exists, it is executed and the next common action is obtained and so on.

B1 ||| B2: the interleaving operator **|||** means that at any moment one can obtain actions from behavior expression **B1** and later obtain actions from **B2** and vice versa.

B1 |[g1, g2,... gn]| B2: the general parallel operator **|[g1, g2,... gn]|** is a generalization of the full synchronization and interleave operators. It means that on gates g1,...,gn **B1** and **B2** must synchronize; any other gate is treated as interleaving.

B1 [> B2: the disable operator "[>" means that any time during the execution of **B1**, **B2** can take over, thus terminating **B1**.

B1 >> B2: the enable operator ">>" means that provided that **B1** has completed successfully, **B2** can be activated.

A typical example of a behavior expression would be:

a_1; (a_2; stop [] a_3; stop)

meaning that action $\mathbf{a_1}$ precedes behavior expression **(a_2; stop [] a_3; stop)** and that this behavior expression is composed of an alternative of two actions $\mathbf{a_2}$ and $\mathbf{a_3}$ followed by inaction.

In telephony the fact that after an off hook a user can either dial or hang up would be represented as:

off_hook; (dial; stop [] hang_up; stop)

The LOTOS behavior expression by itself does not necessarily show the possible sequences of actions explicitly. These are the result of sometimes lengthy evaluations of these expressions using inference rules. The tool has an integrated environment that allows one to apply the inference rules on LOTOS behavior expressions and obtain sequences of actions.

The concept of behavior expression provides a powerful natural structuring mechanism. Behavior expressions can be labelled with names, becoming processes which can have their own generic parameters, and thus are re-usable.

The simple POTS specification:

So far, we have seen only simple actions, which consisted only of a gate name. More complex actions start by a gate name, which denotes the interaction point where the action takes place, and continue identifying a set of values exchanged as a result of the execution of the action. This specification uses two gates: the user gate **u** for interactions between a user and his phone set and the gate **n** for interactions between the phone set and the network. The actions specified are abstractions of the real world, of which they provide only a concise description. An action is described by providing the name of the gate (point of interaction) and the values offered (experiments):

The action `u !offhook !NUMBER` means that, at gate **u**, two values are offered: the constant **offhook** and the content of the variable **NUMBER**.

The overall structure of the POTS specification is as follows:

```
behavior


( phone[u,n](num1) ||| phone[u,n](num2) )

|[n]|

network[n]
```

This specification indicates that two instances of phones with their respective numbers (assigned to the parameters **num1** and **num2**) are in interleave, both running in selective synchronization with the network; i.e. they synchronize with the network only for those actions whose gate is **n**.

The phone process description

The phone process is specified as a alternative between a call initiator and a call responder behavior, which are both straightforward sequences of actions.

```
process phone[u,n](NUMBER:phone_number):noexit:=

        u ! offhook ! NUMBER; (* call initiator role *)
        n ! conreq ! NUMBER ? CALLED_NUMBER:phone_number;
        n ! connect ! NUMBER;
        u ! talk ! NUMBER;
        stop
[]

        n ! ring ! NUMBER; (* call responder role *)
        u ! answer ! NUMBER;
        n ! connect ! NUMBER;
        u ! talk ! NUMBER;
        stop

endproc
```

The network process description

The network is specified as a simple sequence of actions that depict the end-to-end order of the combined sequence of actions of the two phones relative to network events.

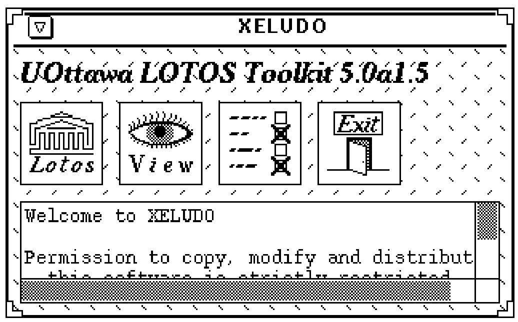
```
process network[n]:noexit:=

        n ! conreq ? CALLER:phone_number ? CALLED:phone_number;
        n ! ring ! CALLED;
        n ! connect ! CALLED;
        n ! connect ! CALLER;
        stop

endproc
```

The Toolkit X window environment

The XELUDO toolkit allows a user to execute a specification, in order to verify that it describes the desired behavior, or to explore the behavior to detect design flaws. The environment also lets you manipulate the specification in many other ways, allowing you to test data types and search occurrences of a specific goal.

In order for a specification to be executed, a file containing that specification must of course exist. If this file does not exist yet, copy the specification given in appendix (without lines number) in a text file named *simple_phone.l*. The X environment must be up and running. We are now ready to start the tool, by typing `XELUDO` on the command line.

The following window appears on the screen:

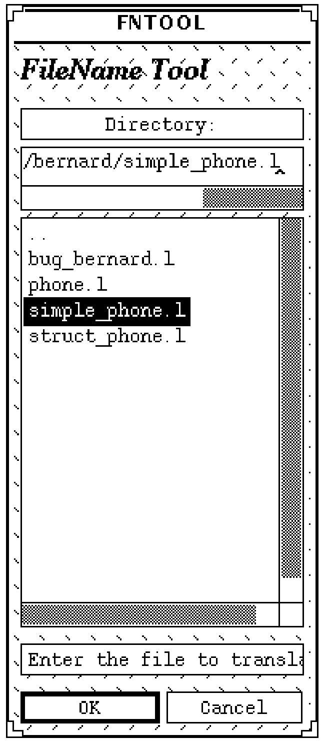


This is the toolkit **main window** or **XELUDO window**. To exit the whole application, just hit the *exit* button of this window.

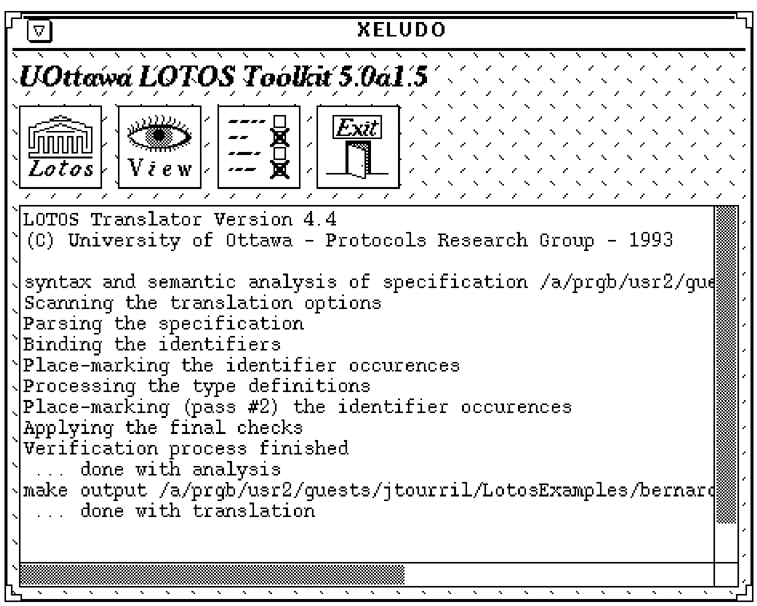
Translating a specification

In order to be executed, the specification must be syntactically and semantically checked, and translated into an appropriate Prolog database, a format understandable by the toolkit. This task may be performed from the UNIX prompt by calling the tool *lotos;* but the second alternative is chosen here, by choosing the **TRANSLATE** item of the LOTOS menu:

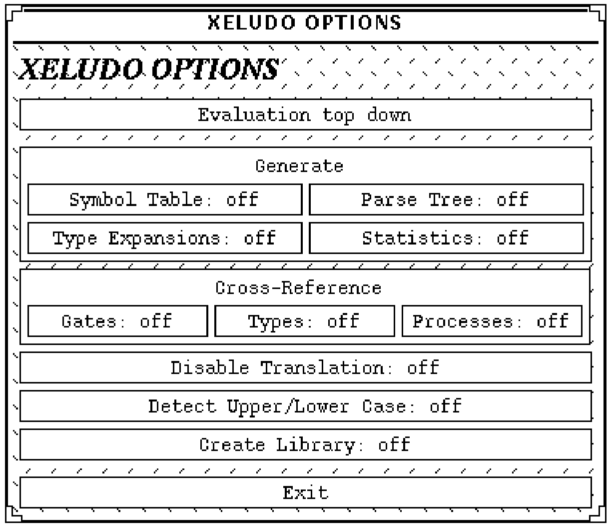
The following window then appears. Select your specification text file (simple_phone.l) and click on the **DONE** button:



Notice that when the toolkit is performing an operation, buttons become insensitive. The specification is being checked and translated. After a moment, the result of the translation appears in the **XELUDO window**:

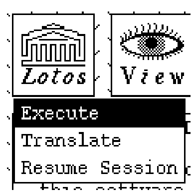


Had the specification contained any errors, this would have been displayed, and the translation process would not have completed its task.So the Prolog database would not have been created. Only correct specifications get to be translated completely. Therefore, when errors are detected, one must correct them and the modified text file must be submitted to the translator again. The translator has many useful features, whose control is gained in the **option window** of the XELUDO window:

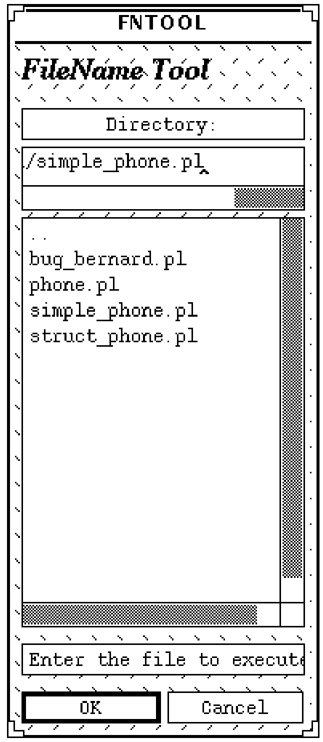


Executing a specification in the step by step (ISLA)

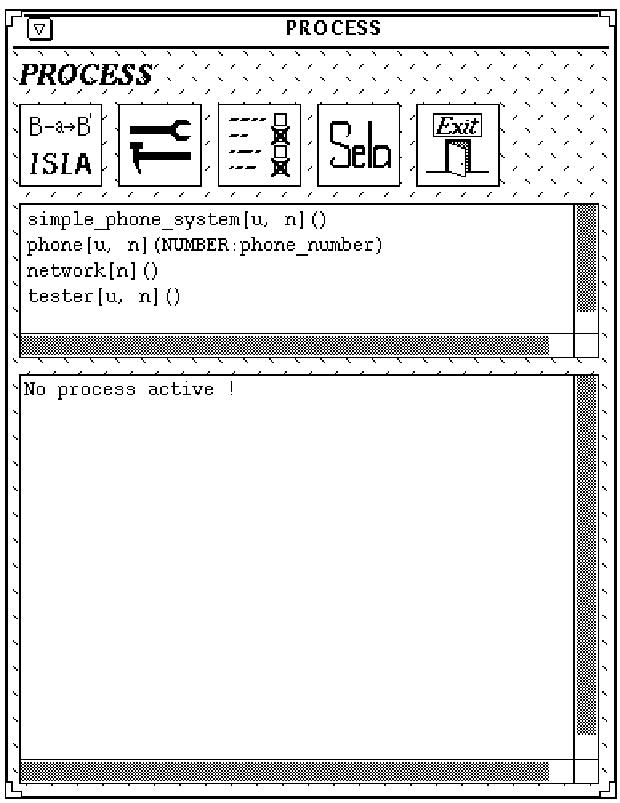
Assuming that the translation was successful, it is now wished to execute the specification, and discover how it works. The (translated) specification can be loaded into the system (making it also readily executable) with the EXECUTE item of the LOTOS menu:



The window named **FNTool** (standing for “File Name Tool”) appears again. This window will appear every time a filename needs to be specified, be it for input (reading) or output (writing). The tool “pre-selects” only files with the expected name extension (when such a requirement exists). Directories are always listed, allowing you to navigate your way into the desired location. In many cases,the tool will also suggest a filename, based on many factor, such as the last directory selected and the name of the loaded specification:

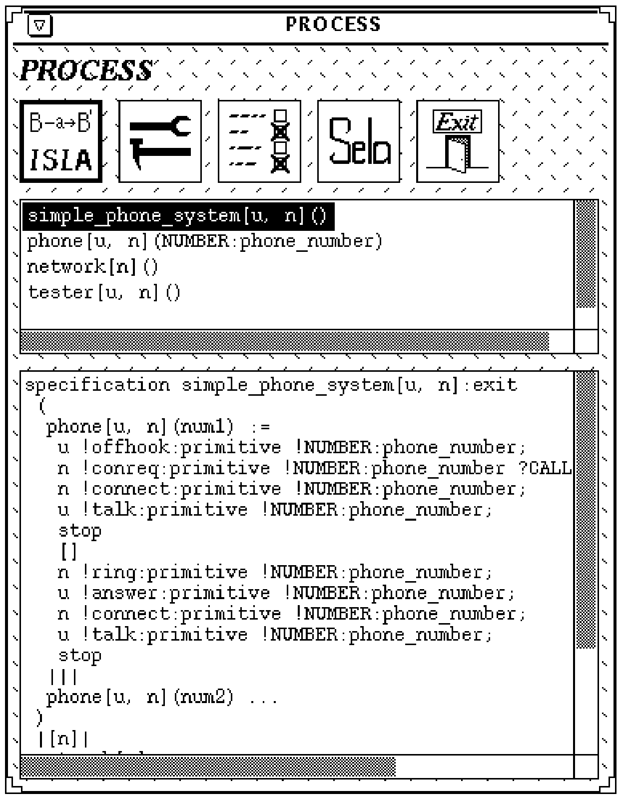


Once the proper file is selected (above "simple_phone.pl" is highlighted for having been clicked on already), click now on the **DONE** button for loading the selected specification. The PROCESS window appears on the screen:



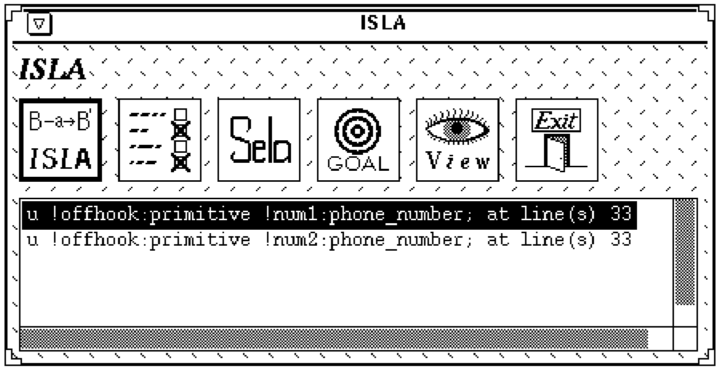
The first part of the window permits to select one of the behaviors defined in the loaded specification. The main behaviour (the first item) bears the name of the specification; all the other ones are the processes defined within. This gives the choice of testing either the global (main) behaviour, or the behaviour of any of the processes independently.

Whenever a process is selected in the upper part of the PROCESS window, the behaviour of that process is displayed in the lower part of the window. The two scrollbars (one for the vertical motion and one for the horizontal motion) can be used to see the whole process behaviour. If necessary, the PROCESS window can be enlarged.



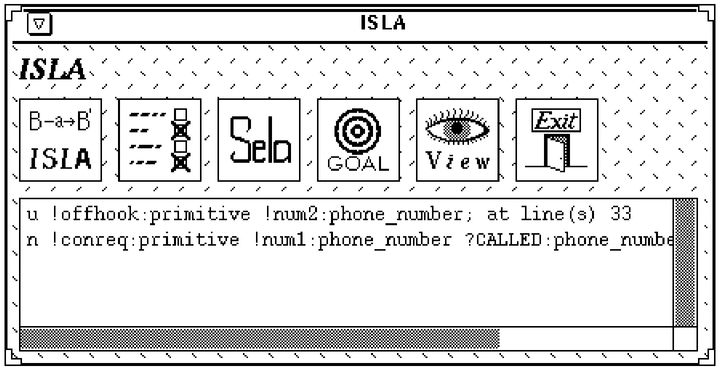
Selecting the first process (i.e. the main behaviour) and pushing the **ISLA** (Interactive System for LOTOS Applications) button starts the step-by-step execution mode.

Once the **ISLA** button is pushed, two more windows appear. The first one is called the ISLA window. It displays the list of available actions for the current behaviour (upon starting a process, the current behaviour is just the behaviour of the selected process). It allows one to select and execute one of the listed actions, thus going to the next behavior, and causing the list of actions to be updated accordingly. The ISLA window also provides access to other tools that may process the current behaviour.



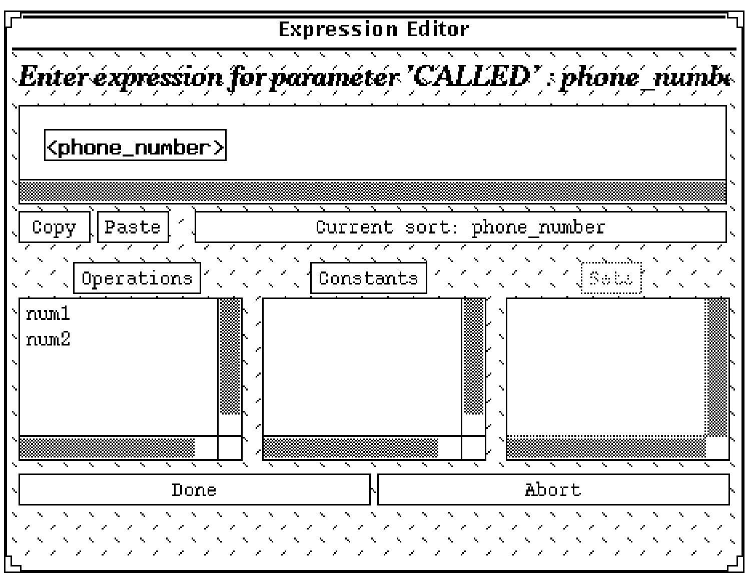
We are about to execute a sequence of events, where the first user calls the second one, and the communication gets established. First, the first user must get the phone; so the action labelled `u!offhook!1` is executed. After executing this action, the second window, called the HISTORY window, becomes:

The executed event has been added to the history tree.



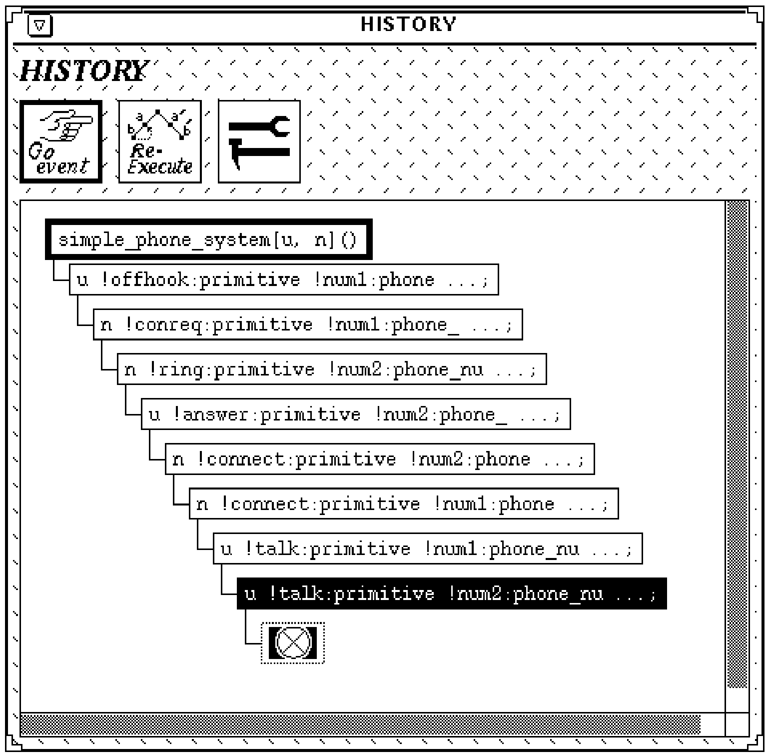
The HISTORY window manipulates historical data, by organizing the executed actions in a graphical tree display (i.e. a trace of execution), and permitting to return to a previous state of execution.

The ISLA window now displays the actions available in the new behaviour, i.e. in the behaviour deriving from the previous execution. The first phone could ask for connection with action `n!conreq!num1...` Note that this action has an “accept” experiment (`?CALLED:-phone`), implying that a value from the environment is expected Since it is desired to have full control over the execution, the tool will prompt you to enter such a value, each time one is required. When executing this action, the Expression Editor window appears, letting you enter a value:

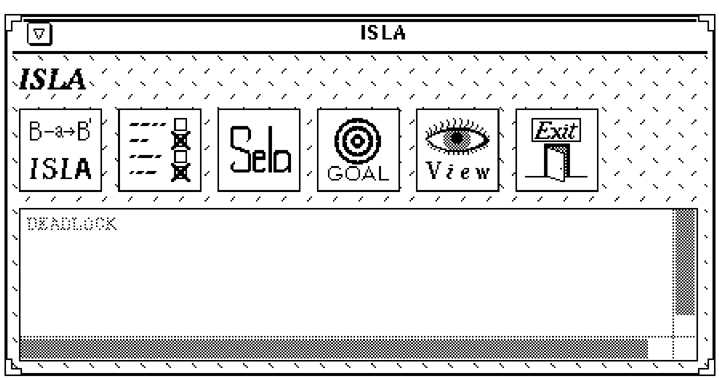


This window is called the **expression editor** because it allows you to construct value expressions according to the data types definitions contained in your specification. It checks the validity of the expression while you compose it. For now, what is required is that the number of the called party be entered. By selecting “num2” from the list of available operators, and clicking on the **DONE** button, the Expression Editor will have this value known to ISLA. After that, since no more values are requested, the action is executed normally; the ISLA and HISTORY windows being updated similarly as after executing the first action.

The end of the connection sequence between our two phones is then executed. Here is what the HISTORY window should display afterwards:



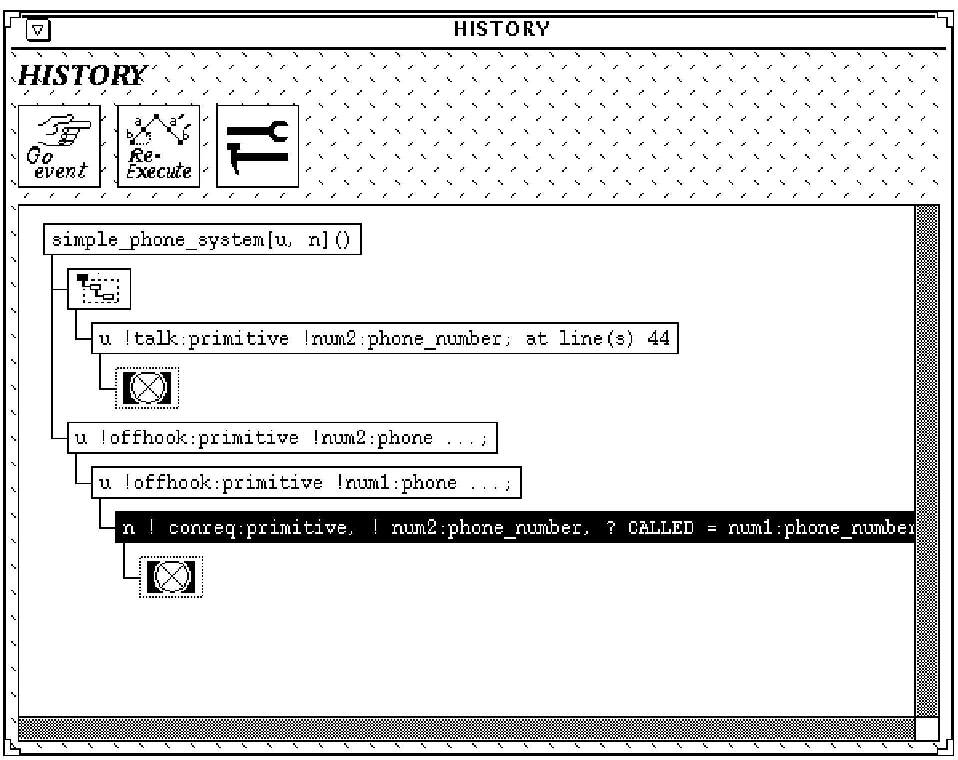
The symbol at the end of this sequence, in the HISTORY window, indicates DEADLOCK, i.e. the specification has no more actions available. The ISLA window reflects the fact that there is no action available for execution by displaying the word DEADLOCK:



The work done so far in this session could be saved in order to be re-used in a later session. The current state of ISLA may be saved with the **SAVE SESSION** item of the **UTILITY** menu:

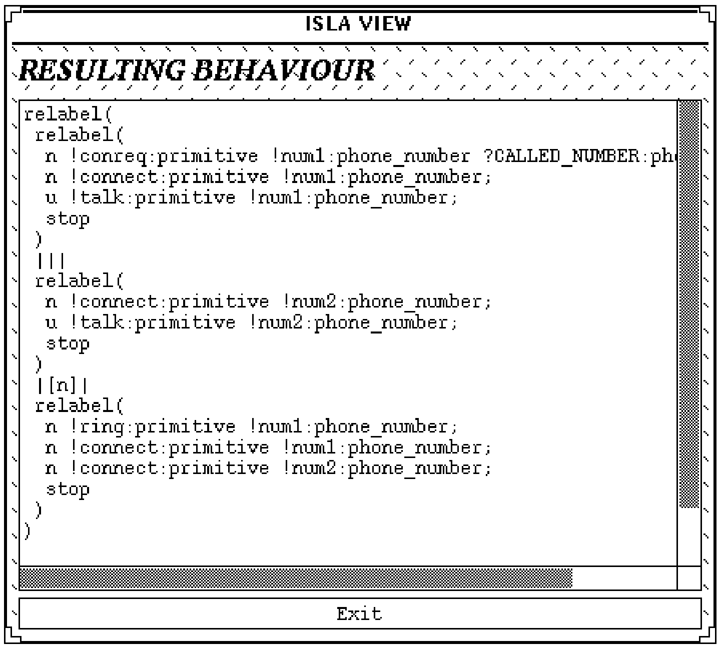
This allows one to reset this state of ISLA by using the **RESUME SESSION** item in the **LOTOS** menu, instead of the **EXECUTE** item. The current history tree could also be saved as a text file with the **SAVE AS TREE** item.

At this point, there is nothing else to do but to try a new sequence. Before doing so, this branch of the tree will be hidden to lighten the display of the HISTORY window. With the MIDDLE button of the mouse, click on the `u!offhook...` event. Now, with the LEFT button of the mouse, select the `simple_phone_system` event in the tree as shown, and push the **GO-EVENT** button in the history window. Note that the actions in i**sla window** are updated. So now, you may execute the following sequence:



Note that some actions in this tree show a long display of the action being executed instead of the short/truncated one. To toggle between "long" and "short" display formats, the RIGHT button of the mouse is used while pointing the action to alter the format.

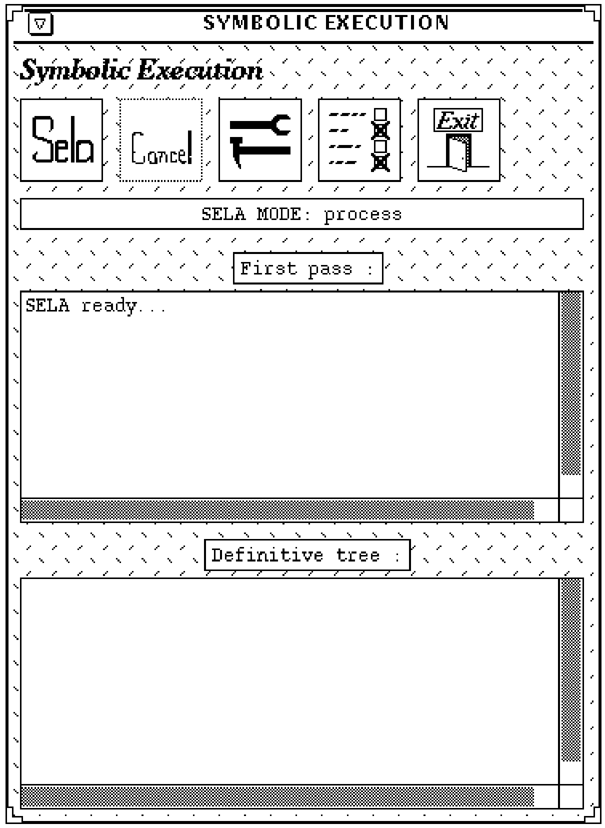
The state obtained after this new execution is also a DEADLOCK. But this deadlock is not explicitly specified in the specification (i.e. it does not, as in the first case, result from the specification of **stop**). It rather results from the two phone behaviours not being able to synchronize on their current behaviour. To see the current behaviour (in a textual LOTOS form) at this point, push the **VIEW** button in the **isla window**. The **view window** appears, showing the exact behaviour and the reason of the deadlock (as can seen in the behaviour, `n!conreq` or `n!connect` cannot synchronize with `n!ring`):



Exploring all possible sequences with SELA

SELA is a tool which generates a symbolic expansion of a behaviour. SELA can expand the behaviour of a given process, or the current behaviour of ISLA. The symbolic expansion is displayed as a tree of all possible actions in your specification. Different limits on execution options are controllable by the user.

To start SELA for a given process, select the desired process in the PROCESS window, and hit the SELA button from the same window. The SELA button in the ISLA window is used to have SELA expand the current behaviour of ISLA. In either case, the SELA window appears:



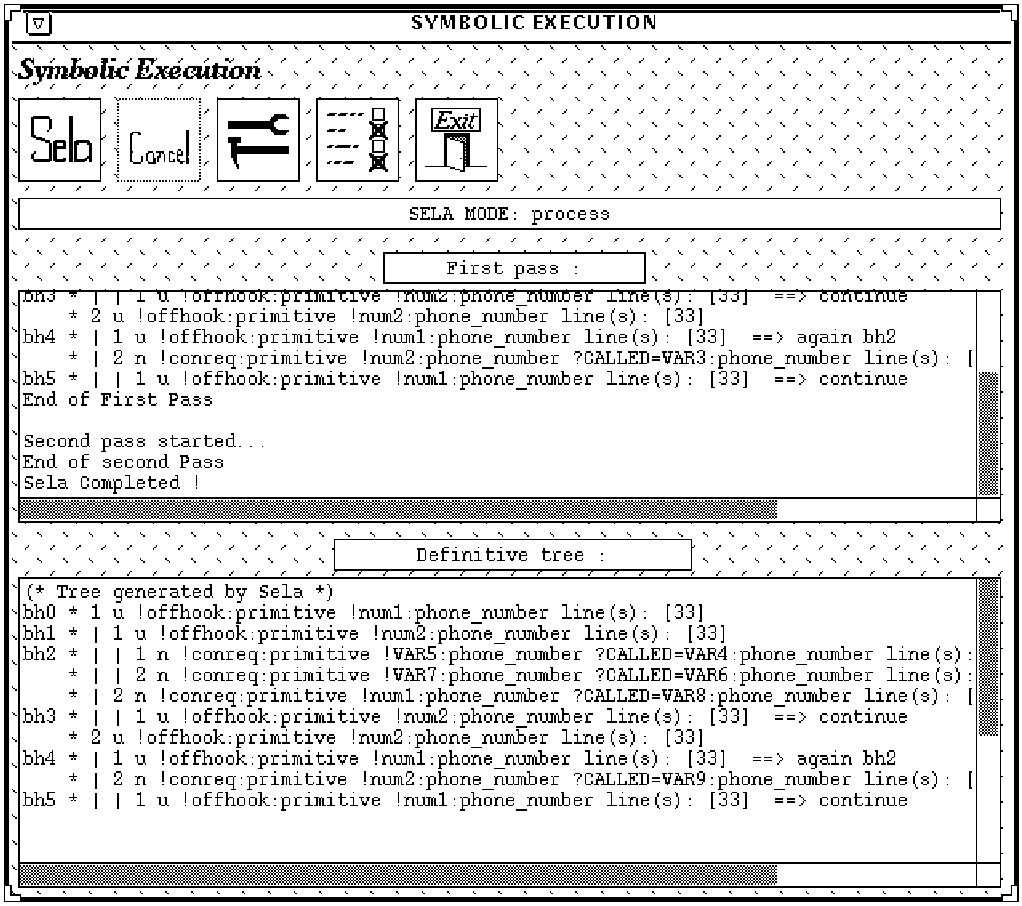
Note that SELA is in process mode (as shown in the toggle button in **sela window**). This reflects the fact that SELA was launched by the PROCESS window. Start SELA by pushing the **SELA** button in the SELA window. After that, the **CANCEL** button becomes active (sensitive) to let you interrupt SELA. SELA begins to fill the upper viewport (as a result of proceeding with the first expansion) then starts filling up the lower viewport (as a result of proceeding with the second expansion).

SELA generates two trees because it works in two passes. The second tree is the definitive one. Each line of the tree is composed of an optional label (to reference the behaviour), some indentation character to indicate the level of the action, the action itself (with all values replaced by symbolic values), and, at the end of some lines, some specials keyword:

`DEADLOCK` meaning that the action is followed by a deadlocked behaviour.

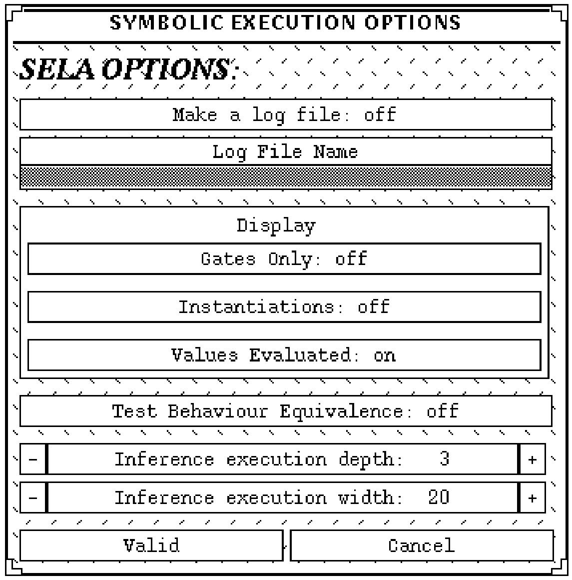
`==> again bh#` meaning that the execution at this point continues exactly like the subtree starting at node labelled `bh#`, the resulting behaviour at these two points being identical.

`==> continue` meaning that the execution continues deeper than what is actually shown, but stopped to respect the user-defined depth of execution.



SELA allows you to save the generated tree (the one in the second viewport). This can be achieved by selecting the **SAVE TREE** item in the **UTILITY** menu:

Options settings exist to control the generation of the symbolic expansion They are accessed by pushing the **OPTION** button:



One of the main options is the **Inference execution depth**. This value allows you to control how deep SELA will carry the expansion. For example, the previous tree contained some `==> continue`. Increasing this value enables to "see" deeper in the evaluation and perhaps eliminate all continues (if possible). But the deeper the tree, the longer it takes to generate it Always remember that the growth of the tree is usually exponential. Note also that options are validated only when exiting the OPTION window.

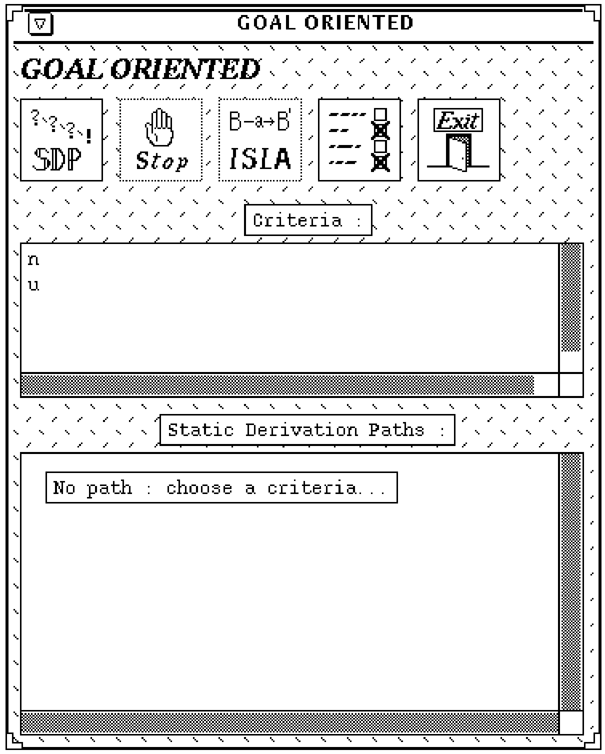
SELA may also be started from the ISLA window, to rather process the current behaviour, or switched later on the this mode (if SELA is already started) using the toggle button "SELA MODE" to make it display `current behaviour`. This permit to start the SELA tree on the current behaviour; i.e. you may first have a actions executed by ISLA, the have SELA generate a tree from the point you reached in ISLA. In cases where there is not too many recursive definitions, this reduces the size of the tree by focusing only on some execution cases. This strategy may be very useful when the initially executed actions reduce the number of behaviour choices or provide some important values.

Reaching and searching all occurrences of gates with GOAL ORIENTED

The GOAL ORIENTED tool is expected to become a third way of exercising a specification, but only a limited set of its capabilities have yet been implemented.This tool allows one to request having a specific gate reached. This allow you too test the reachability of some gates in your specification faster than it would be possible with ISLA, generating some sequences of actions normally much smaller than SELA symbolic trees. For now, GOAL ORIENTED searches only for gates, but this will be extended in the near future.

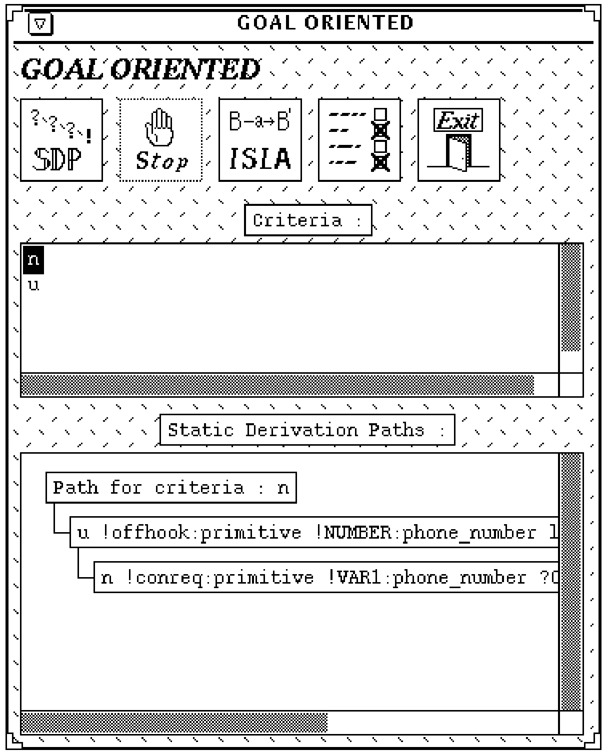
This tool works on the current behaviour of ISLA. It may be started from the ISLA window, with the GOAL button. In the previous example, a point was reached where there was a deadlock; let us return to the root event in the tree.

The following window appears on the screen after pushing the GOAL button in the **isla** window.

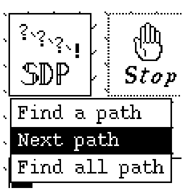


The possible search criteria consists in one of the behaviour external gate. Selecting the gate `n` and choosing the **FIND A PATH** item in the **SDP** menu:

has the GOAL tool to find a the first path leading to the gate `n` displayed in the second part of the **goal** window. Note that the label `no path` has been replaced by the goal of the search. Like in the history tree of ISLA, nodes may be selected or hidden using the buttons of the mouse:



Having an initial path displayed activates the NEXT PATH item of the SDP menu. This item permits to find the next sequence of actions:



Using the **NEXT PATH** item allows one to find the traces one by one. This is most convenient when the firsts paths are expected to be satisfactory. In instances where all the paths are preferable, the FIND ALL PATH item should be used instead. The search will continue until no more no paths are found; or will cease when the user hits the **STOP** button (hit again **FIND ALL PATH** to restart the process).

The tool searches for paths given the current behaviour of ISLA, and, as SELA, the lengths and quantity of paths will often get smaller as the execution in ISLA proceeds.

Testing strategies

As seen above, testing the design of a specification can be accomplished in a manual way by executing the specification step-by-step (ISLA) or by generating symbolic trees (SELA).

Another semi automatic way is to verify if an anticipated test sequence is accepted by the behaviour. Such a test sequence has been specified in our specification, and labelled as the process tester[u,n]. It offers two alternative behaviors: one of a correct call completion and another one where the answer action has been intentionally removed.

```
process tester[u,n]:noexit:=

        u ! offhook ! num1; (* a good test sequence *)
        n ! conreq ! num1 ! num2;
        n ! ring ! num2;
        u ! answer ! num2;
                n ! connect ! num2;
        n ! connect ! num1;
        u ! talk ! num1;
        u ! talk ! num2;
        stop

[]

        u ! offhook ! num1; (* a bad test sequence *)
        n ! conreq ! num1 ! num2;
        n ! ring ! num2;
                (* answer action intentionally missing *)
        n ! connect ! num2;
        n ! connect ! num1;
        u ! talk ! num1;
        u ! talk ! num2;
        stop

endproc
```

The tester is then composed in parallel with the POTS specification:

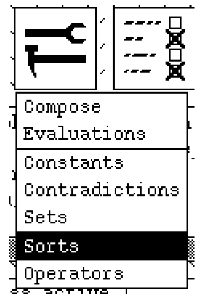
```
(
        ( phone[u,n](num1) ||| phone[u,n](num2) )

        |[n]|

        network[n]
)
||
tester[u,n]
```

The execution tree of the test is shown on the previous tree, where one can observe that the second alternative goes in deadlock after the **ring** action.

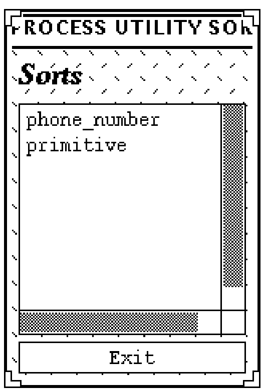
Abstract Data Types

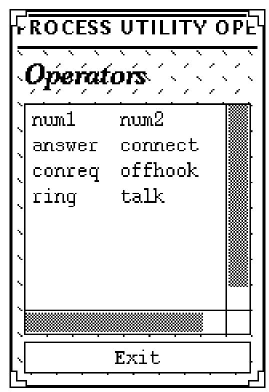
All aspects of the execution of a specification shown before dealt mainly with the behaviour parts of the specification. But XEludo is also a powerful tool for testing and evaluating the abstract data type (ADT) part of the specifications. The ADT part of a LOTOS specification is specified using the ACT-ONE ADT. XEludo provides ways of controlling the evaluation of value expressions and creating constants, among others. All those powerful features are mainly accessible from the **UTILITY** menu of the **process** window.

The first two tools related to the data types of a specification are concerned with looking for all sorts and all operators. Those would be, in the **UTILITY** menu of the **process** window, the **SORT** and the **OPERATOR** items respectively:



The two following windows are obtained, displaying the different sorts and sorts operators used in the currently loaded specification:

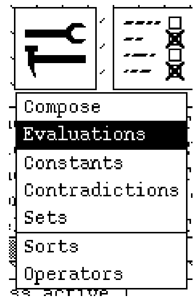




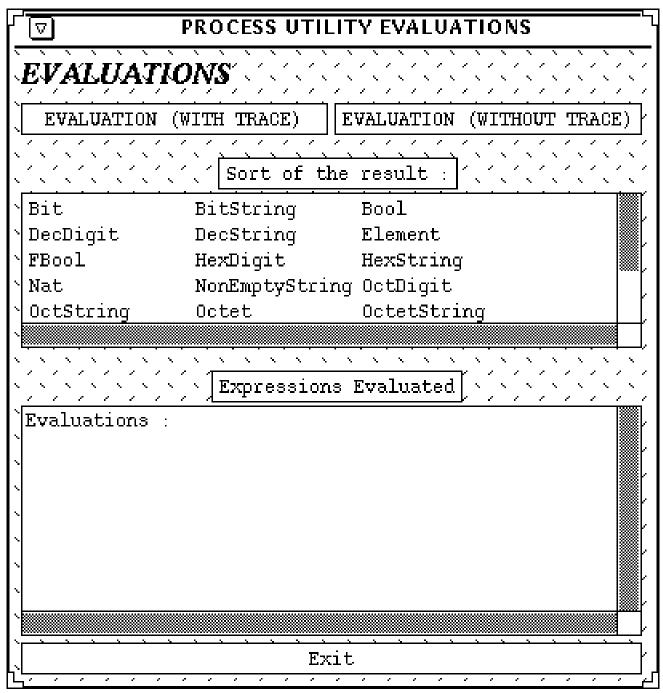
Evaluation of Abstract Data Types

Complex abstract data types can be difficult to verify by hand. The evaluator tool allows one to trace the evaluation of a value expression and evaluate some complex ACT ONE expressions very easily. Since our telephone specification as very few and rather simple data types, a more elaborated specification (in regard with ADTs) is used at this point.

The evaluator is called from the **UTILITY** menu of the **process** window:

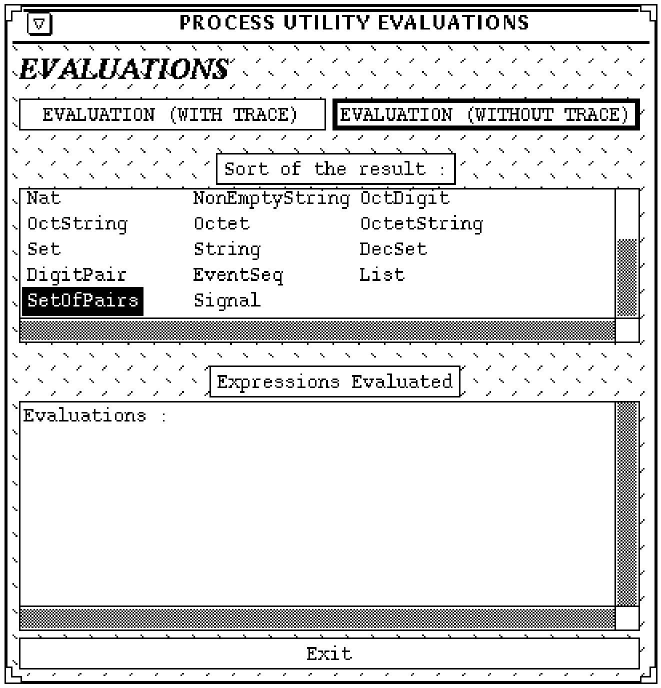


The **evaluator** window appear as:

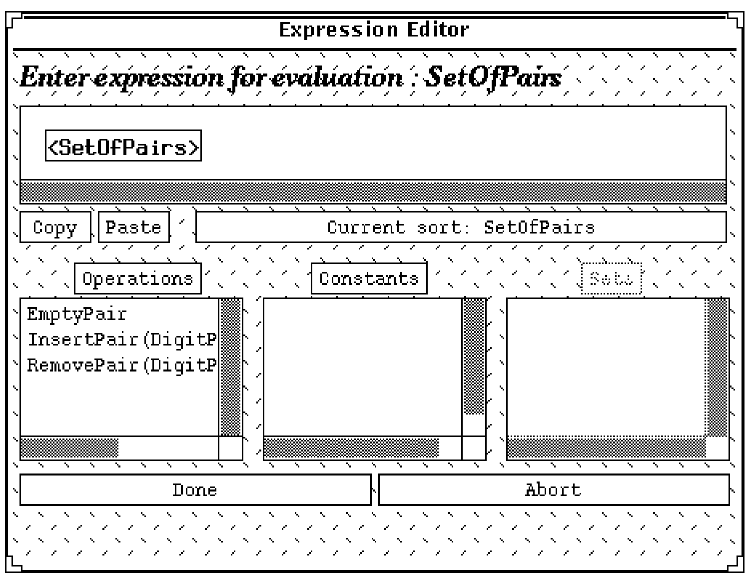


The first part of the window allows one to choose the resulting sort of the whole expression (or sort of the result), and the second part displays the evaluations made so far. All sorts refer to types as defined (or actualized/renamed) in your specification or imported from libraries.

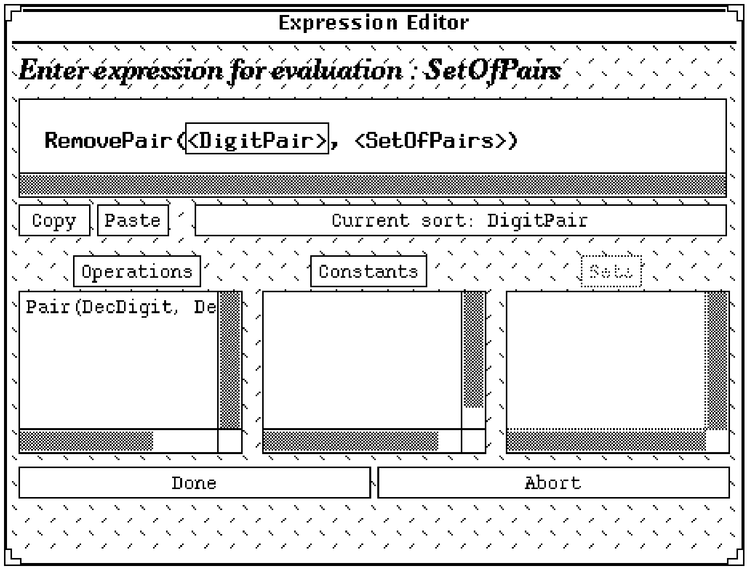
For our first example, let us construct some “set” and test some basic constructors on them. Selecting the `SetOfPairs` sort (using the scrollbar if needed), and pushing the **EVALUATIONS (WITHOUT TRACES)** button has the Expression Editor launched:



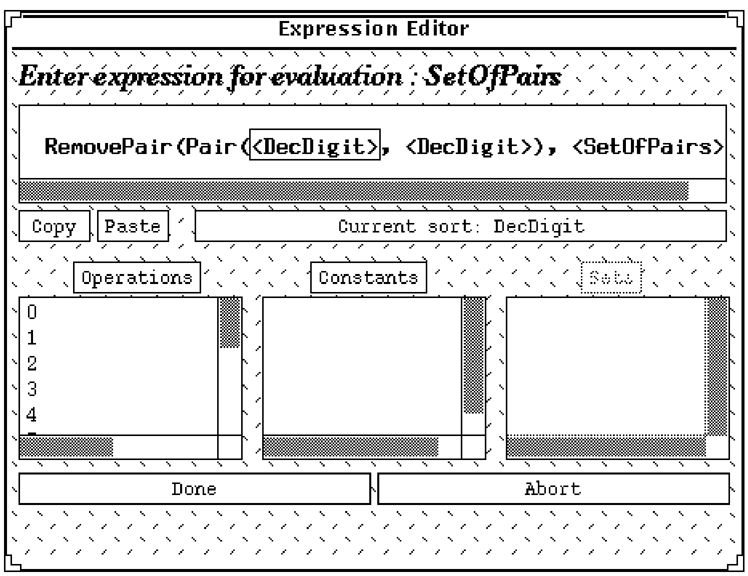
The Expression Editor window appears, displaying the current expression (empty for now), the current sort and operations available for the current sort:



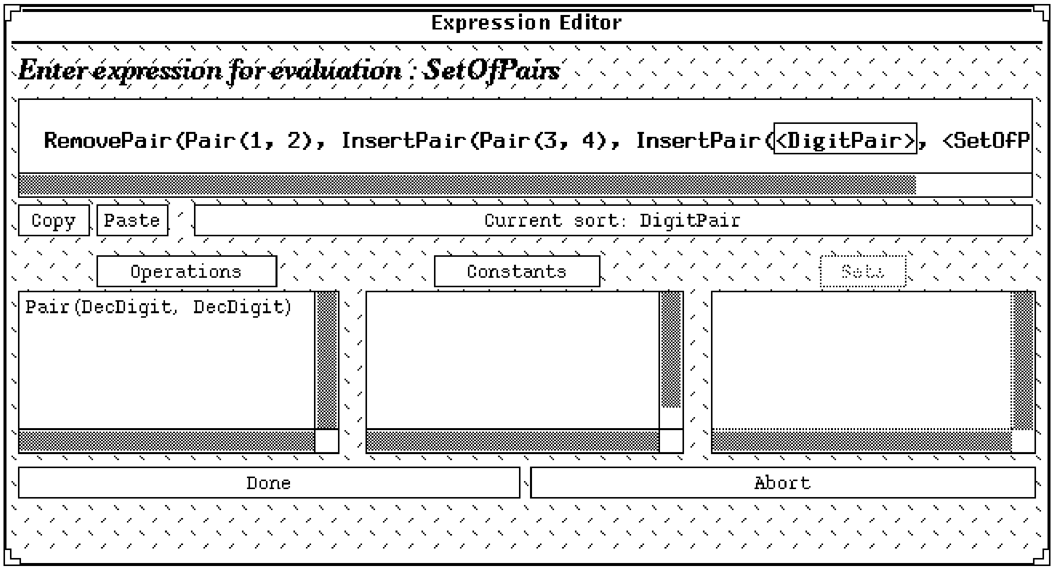
The chosen goal being to test the RemovePair operator, we first select. at the top level of the value expression, the operator `RemovePair`:



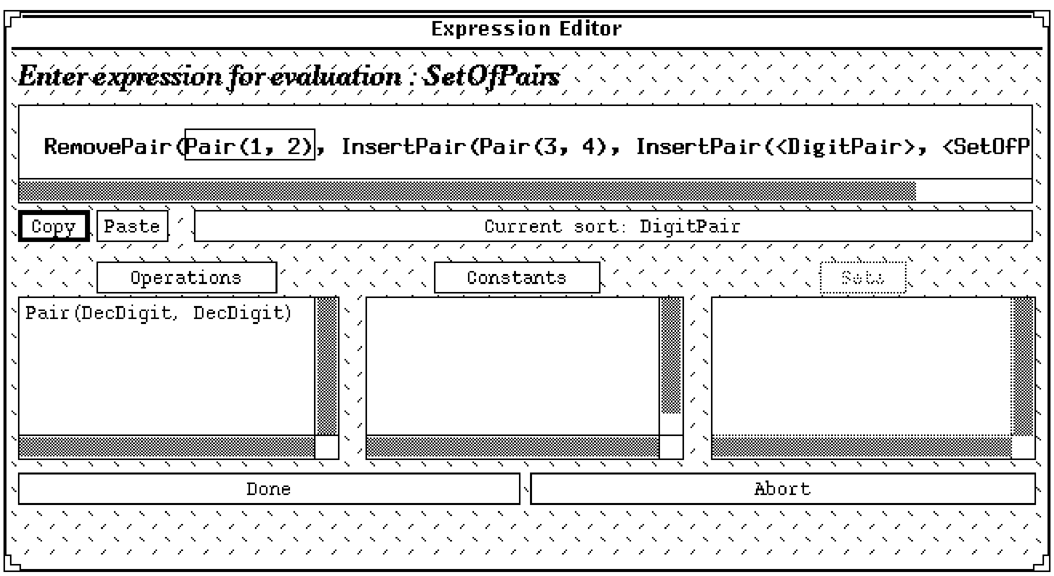
Note that as soon as the operator was chosen, it has been inserted in the expression; the selected expression (appearing in angle brackets) became the first uninitialized child of the RemovePair operator, and that both current sorts and the list of available operations were updated. The Pair to be removed from the set should now be constructed. Selecting the `Pair` operation:



the expression editor enables us to now enter the first digit of the pair. Carrying on with a few additions, the expression becomes:

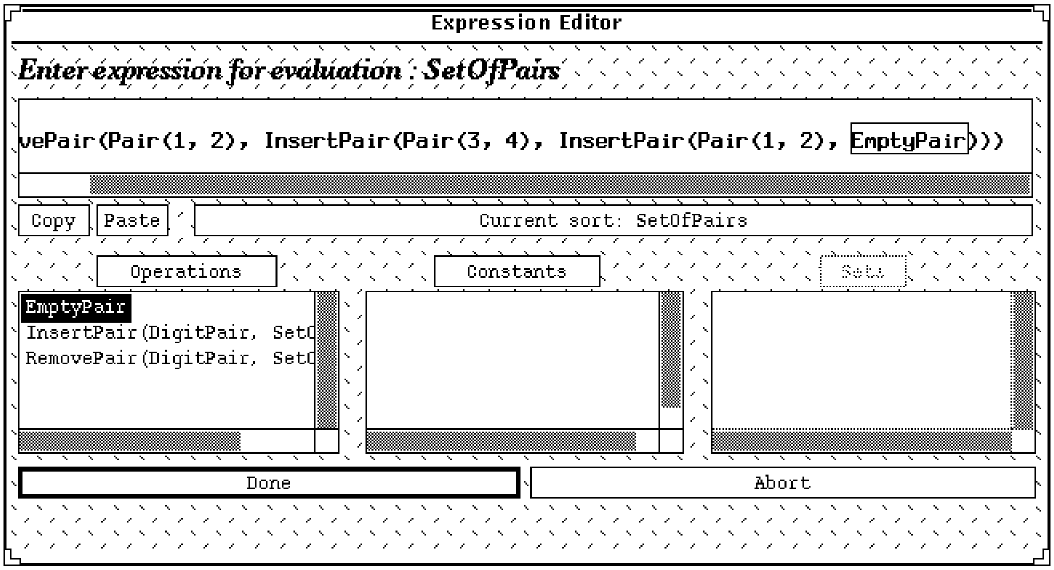


At this point, we want to insert `Pair(1, 2)` in the SetOfPair (to let the remove constructor remove afterwards). Instead of recreating the whole pair, it can be copied over from its first appearance: selecting it, pushing the copy button:

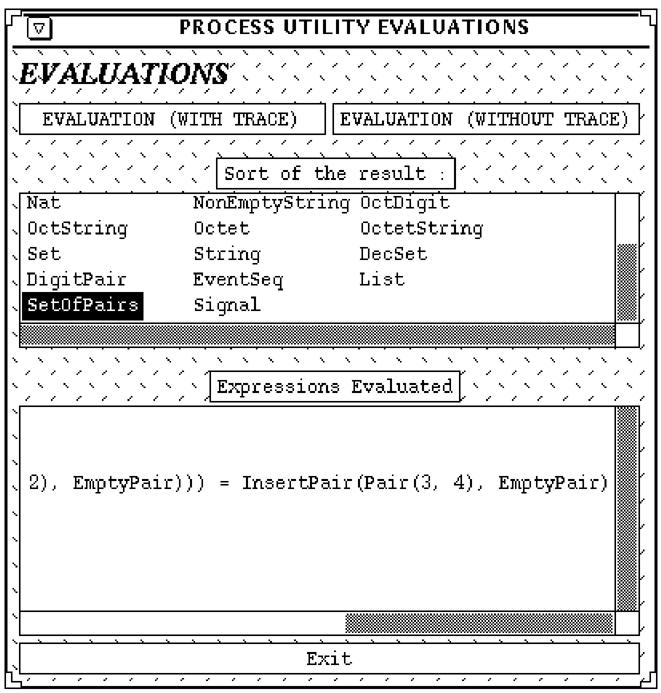


then resetting the `<DigitPair>` node and pushing the paste button.

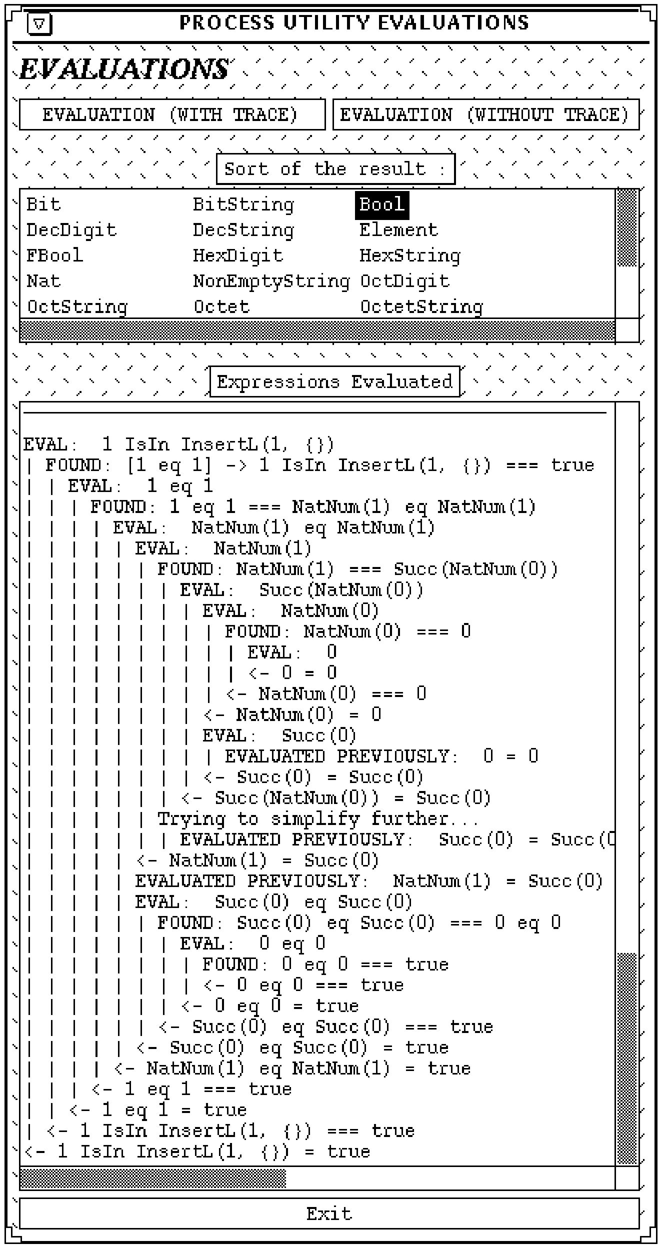
The last element may be filled with an `EmptyPair` operator, and the **DONE** button pushed. This example shows the simplicity of constructing some complex expression without any risk of making syntax (e.g. missing or extra parenthesis) or semantics (e.g. sort) errors. This eventually saves lots of time and frustration.



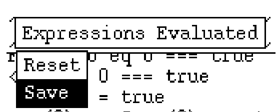
After pushing the **DONE** button of the expression editor, the **evaluation** window displays the result:



But the evaluator may show you more than just the result. It can generate the trace of evaluation, displaying the equations used (and rejected); i.e. all the evaluation steps leading to the construction of the final result. For instance, the expression `1 IsIn InsertL(1, {})` of sort `Bool` produces the following trace of evaluation:



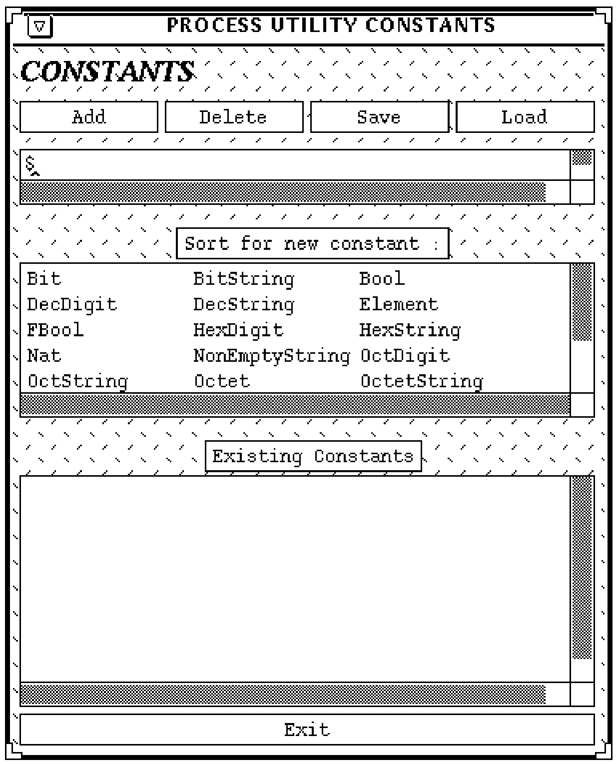
Evaluations may be saved with the **SAVE** item in the **EXPRESSION EVALUATED** menu:



The **RESET** item allows you to clear the Expression Evaluated viewport (because it may be useful to keep evaluation results, they are cumulated; but since some evaluations with trace may lead to an important quantity of text, the user may choose to clear it up).

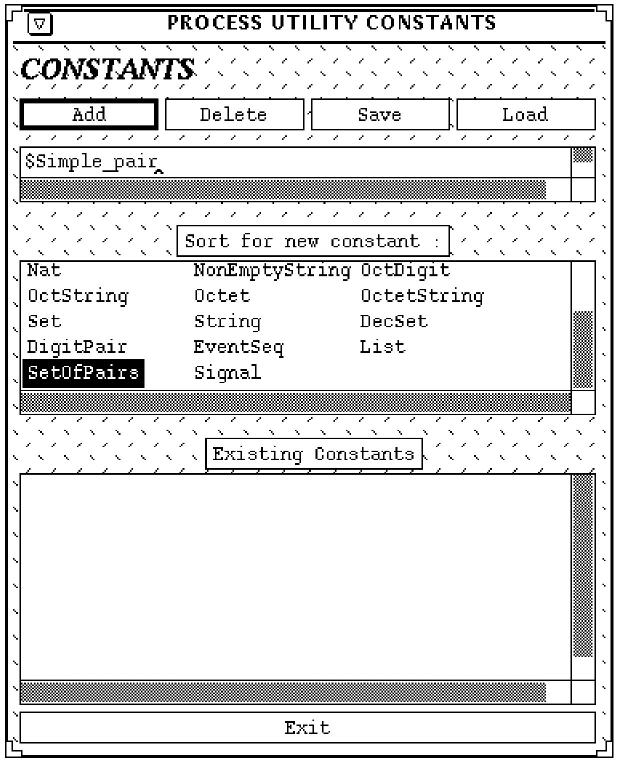
Abstract Data Type Constants

Another tool, called CONSTANTS, is found in the **UTILITY** menu of the **process** window. This tool permits to define some constants re-usable later on by the Expression Editor, to be substituted for a whole expression, or even just for a part of one. This make a value expression both easier to construct and to read. Upon entering the CONSTANTS tool, the following window appears:

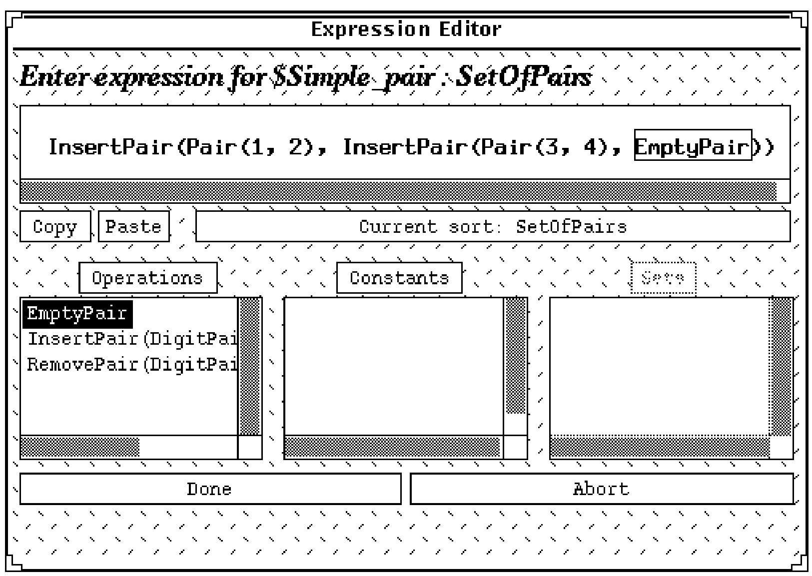


The first part of the window allows you to enter the constant name. This constant name must begin with a `$`, and then contain no `$` sign nor spaces and tabs. The window also displays a list of sorts, to be used to define the resulting sort of the constant. Once the resulting sort of the constant selected, a list of already existing constants is displayed. In the example, there is no constant defined.

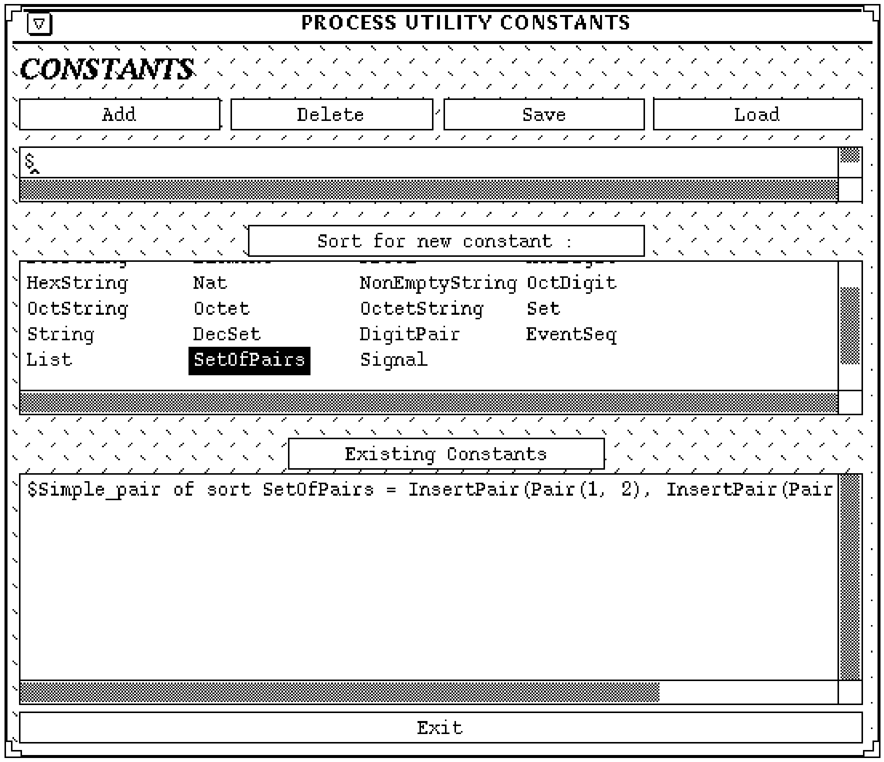
We will construct some `SetOfPair` expression to represent a typical set and name it Simple_pair. After selecting the desired sort, and typing the name to be associated to the constant, the ADD button may be pushed to have the Expression Editor launched:



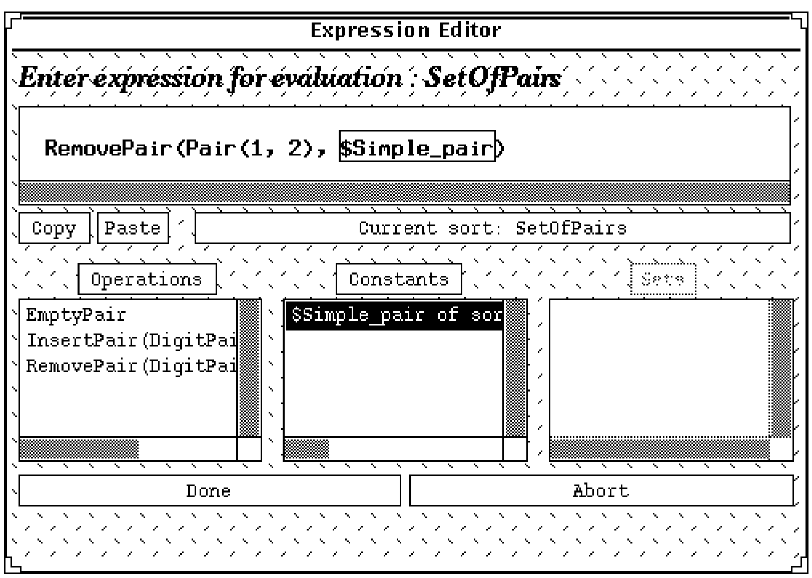
The desired value expression may then be entered:



Hitting the DONE button completes the creation of the new constant, as it subsequently appears in the CONSTANTS window:



Constants defined in the CONSTANTS tool are known by the Expression Editor (they appear in the second viewport, to the right of operations); thus they are available to construct values expressions later on, whether it is when making an evaluation, or defining another constant, or instantiating a process values, or executing actions with experiments requiring values! For example, evaluating an expression with the operator RemovePair, as done previously:



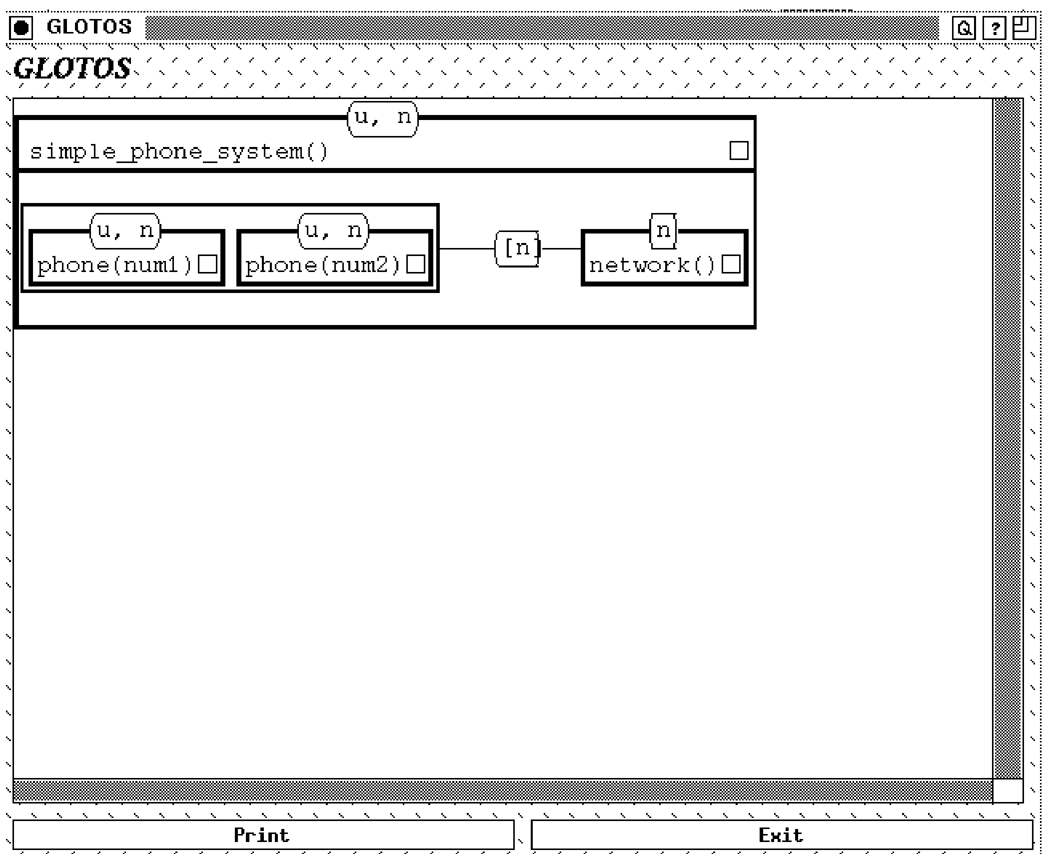
Graphic LOTOS

The Toolkit allows the automated translation of LOTOS behaviors into G-LOTOS graphic representations.

In the figure below, the graphical representation of the high level behavior of the telephone specification is shown:

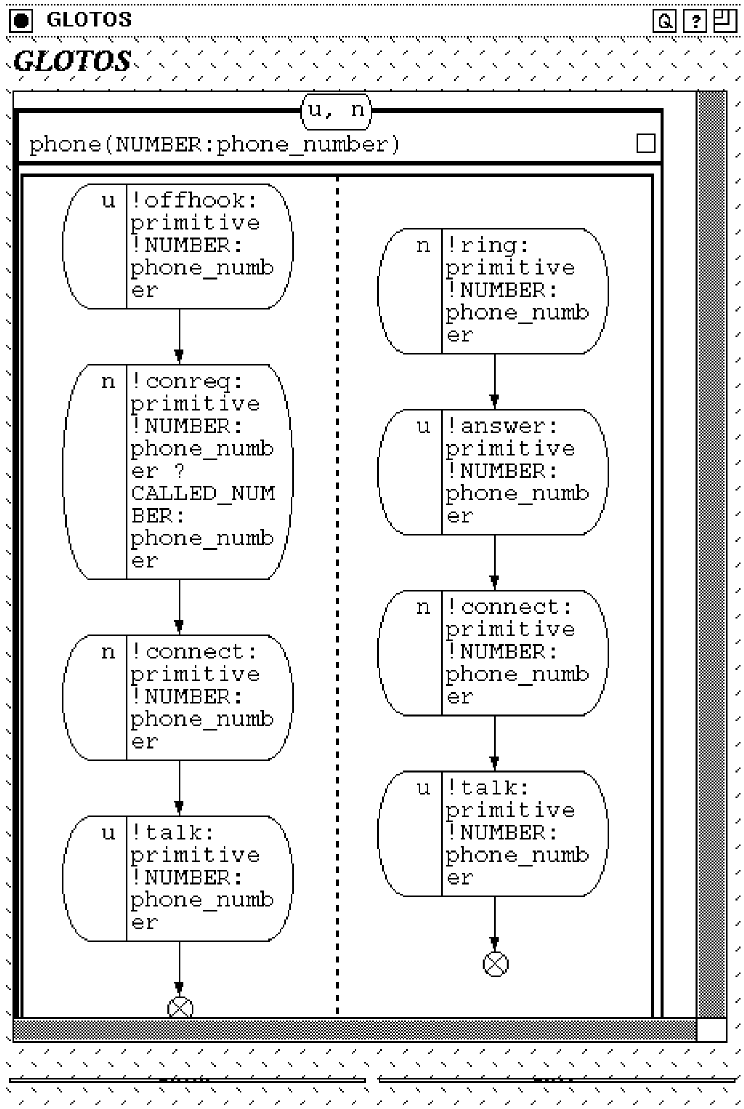
```
( phone[u,n](num1) ||| phone[u,n](num2) )

 |[n]|

 network[n]
```

It is obtained by clicking on the G-LOTOS icon after the process has been selected



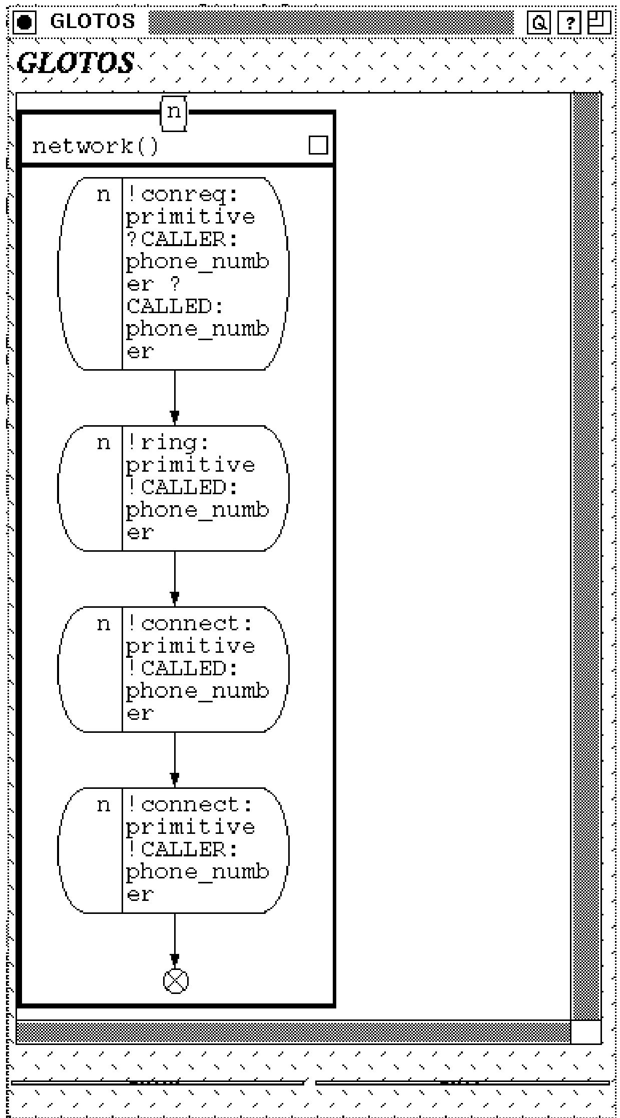
Any G-LOTOS screen can be used interactively to further display details of processes. By clicking in the check box of any of the instances of processes phone or network, their G-LOTOS representation will be displayed. As much information as possible is displayed in any given screen.

The G-LOTOS representation of the phone process under the G-LOTOS tool is the following:



Each action is depicted as a rounded rectangle where the interaction point (gate) is separated from the value offers. The arrows represent the action prefix symbol.

The G-LOTOS representation of the network process:

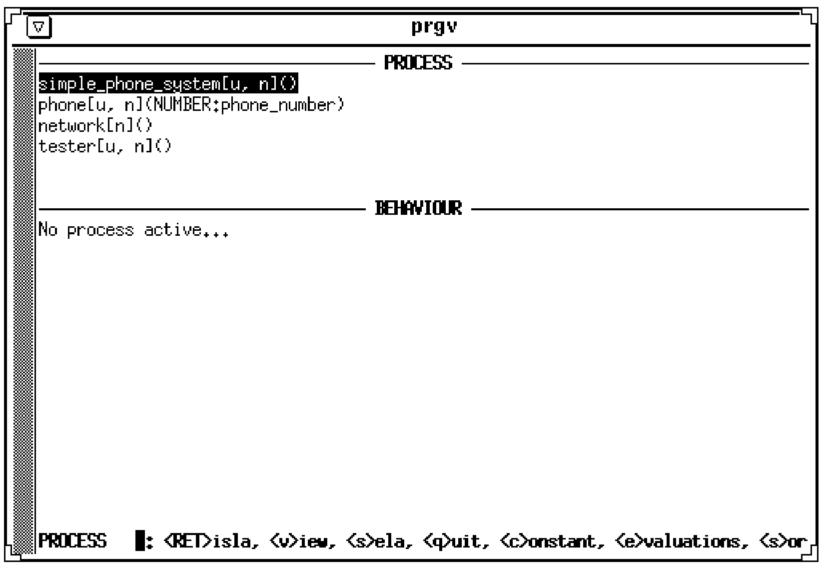


The Toolkit TTY interfaces

This section presents an overview on the TTY (curses) interface of the toolkit. This interface is provided for those users without the X window system, such as remote-login users. The reader is expected to have become familiar by now with the Eludo Toolkit functionality, and the focus will therefore be centered on the presentation and interactions of this interface.

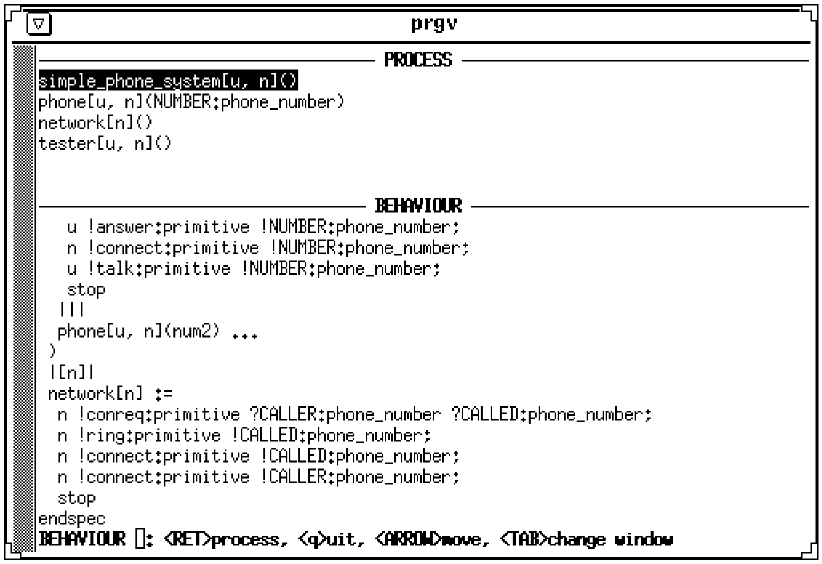
The TTY interface of the toolkit is called *eludo*, by opposition to *XELUDO* witch is the interface for the *X* window system. The main characteristic of *eludo* is that it provides a device independent terminal interface. It may be used with any text terminal whose capabilities are properly defined in the */etc/termcap* file.

A typical screen of *eludo* looks like the one presented here:



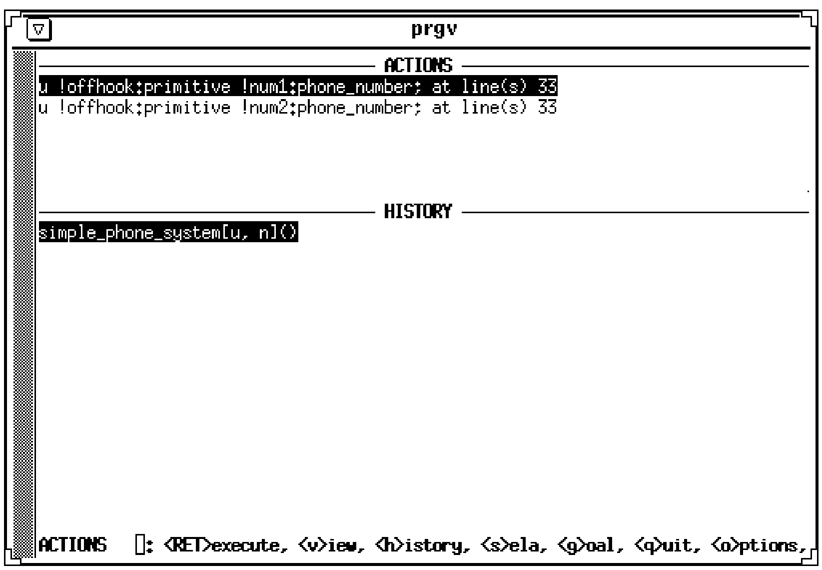
This screen corresponds to the PROCESS window. It is divided in two “viewports”. Each viewport is composed of a title bar (in bold characters, surrounded by two horizontal lines) and an area where the information can be scrolled using the arrow keys of the terminal. The last line of the screen displays the active “viewport” (here PROCESS) and the currently available commands (the TAB key is used to interchange the active and inactive viewports). The **q** key is always used to exit a tool (effectively returning to the previous one).

The last line indicates that the **v** key executes the **view** function. This function displays the behaviour of the highlighted process, in the second viewport (which then becomes the active viewport). When a behaviour is larger than the viewport, only the first (upper) part is shown; the arrow keys permitting to scroll up and down to the desired part of the text:

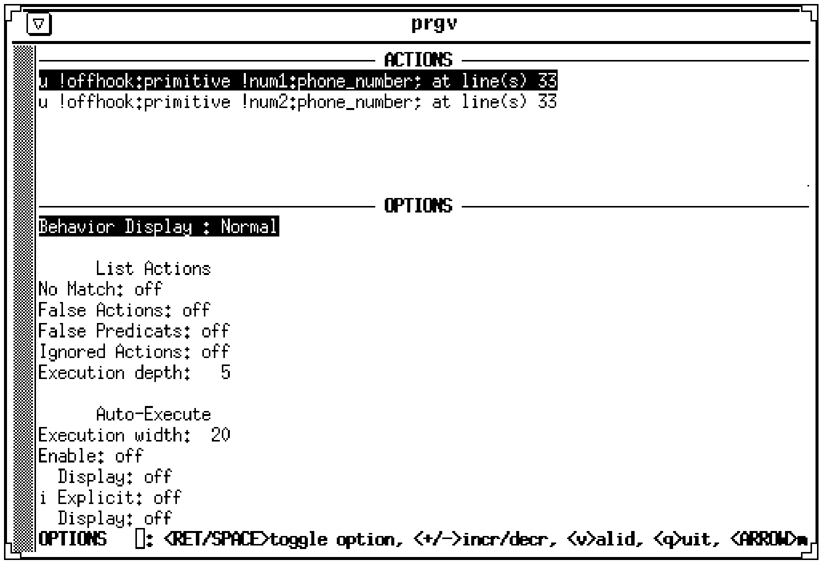


When the BEHAVIOUR viewport is active, the ISLA command is not available. The TAB key returns to the PROCESS viewport, from where the **RETURN** may be used to launch the ISLA tool on the selected process. The **arrow** in the PROCESS viewport are used to select a process; scrolling automatically happening if and when necessary.

The ISLA tool has the following display:

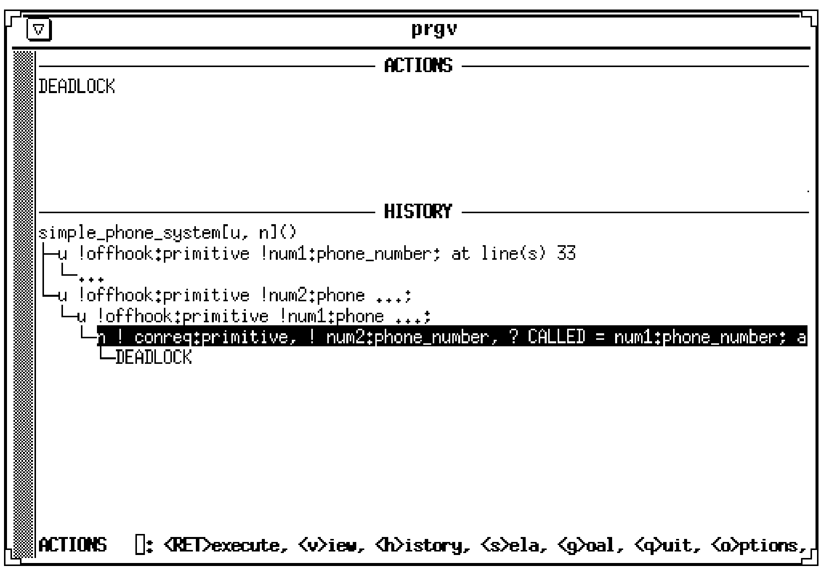


This screen contains again two viewports. The upper viewport is always ACTIONS; the second viewport may be used selectively for the display of HISTORY, CURRENT BEHAVIOUR or OPTIONS. The **o** key calls up the ISLA OPTION viewport. Options are set by selecting them using the arrow keys the pressing the **space** key to toggle their values:

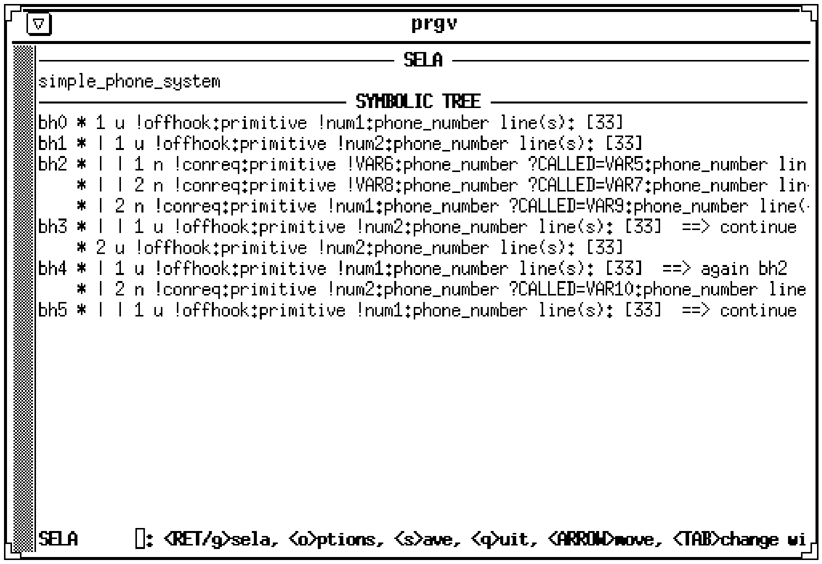


When the ACTIONS viewport is active, actions may be selected (with the UP and DOWN arrows), and executed (with the RETURN key) similarly as in the ISLA window of *XELUDO*.

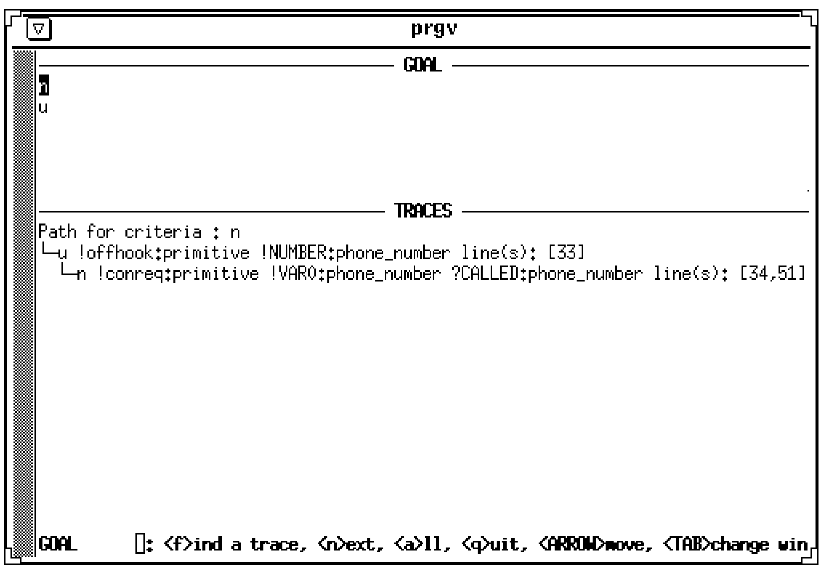
The HISTORY viewport (available with the **h** key) has the same functionality as its counterpart in *XELUDO*. Returning to an already visited state of execution is accomplished by moving up and down the HISTORY tree, then confirming the state to return to with the RETURN key:



The SELA tool follows the same rules. The screen contains only one viewport, which may either be the tree of the first pass or the final symbolic tree:

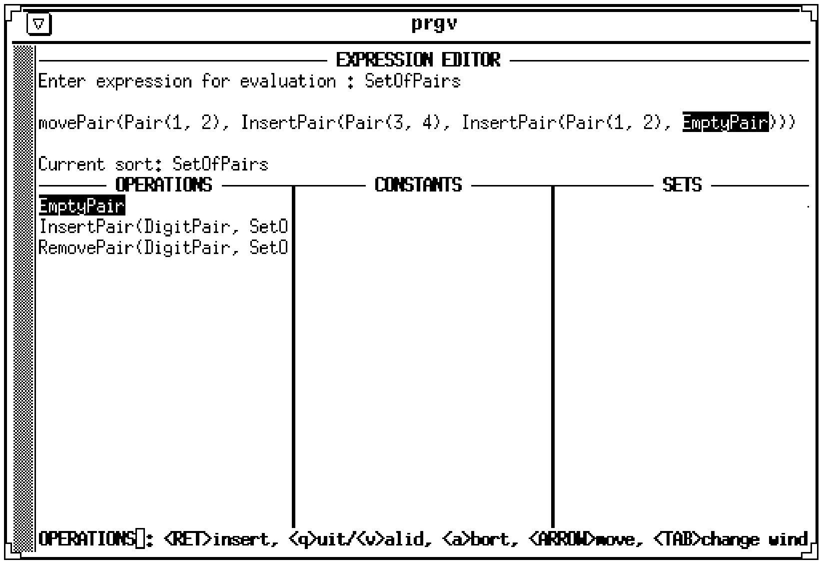


As in *XELUDO*, the GOAL ORIENTED tool is available from the ISLA screen. The search for the sequence of events is based on the current behaviour as well:



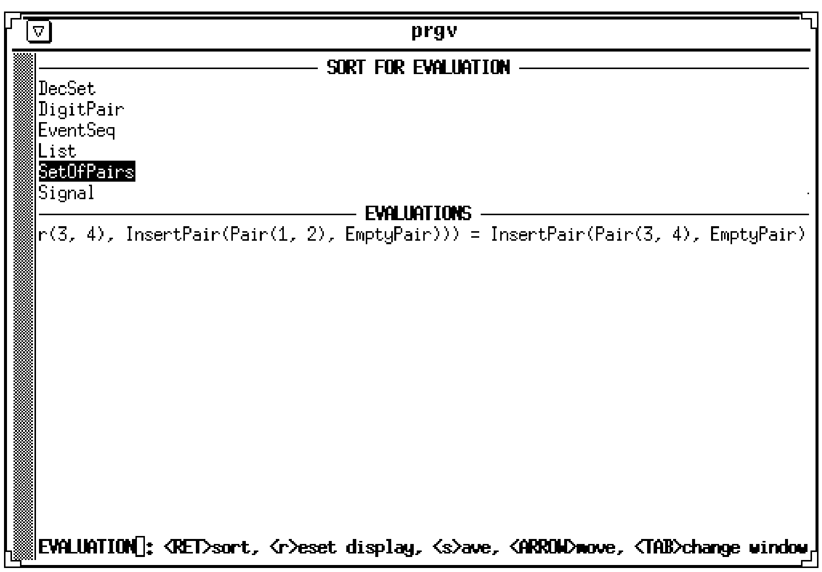
Despite the fact that *eludo* is only a text interface, it is still a very powerful tool and provides most[1] of the features carried by its graphical twin, by pressing only a few keys. A good example is the Expression Editor:

1. GLOTOS is not supported by *eludo.*



The selection of an operator is made from the OPERATIONS viewport. Each time the insertion of an operator takes place, the place-marker (in the value expression under edition) moves to the next uninitialized element. Moving within the edited value expression is accomplished with the four arrow keys, in the EXPRESSION viewport. Copy and paste functionality are supported too. Creating huge and complex expressions a quick and easy task.

All tools found in *XELUDO* (with the exception of the GLOTOS tool) may be found in *eludo*. For instance, results of evaluations may be shown by the EVALUATION tool, with or without the trace of evaluation:



Appendix: The complete specification of the POTS

```
specification simple_phone_system[u,n]:noexit
  type phone_number is
     sorts phone_number
     opns num1, num2 :-> phone_number
  endtype
  type primitives is
     sorts primitive
     opns offhook, conreq, ring, answer, connect, talk:-> primitive
  endtype
  behavior
  (
     ( phone[u,n](num1) ||| phone[u,n](num2) )
     |[n]|
      network[n]
  )
  ||
  tester[u,n]
  where
  process phone[u,n](NUMBER:phone_number):noexit:=
     u ! offhook ! NUMBER ; (* call initiator role *)
     n ! conreq ! NUMBER ? CALLED_NUMBER:phone_number ;
     n ! connect ! NUMBER ;
     u ! talk ! NUMBER ;
     stop
  []
     n ! ring ! NUMBER ; (* call responder role *)
     u ! answer ! NUMBER ;
     n ! connect ! NUMBER ;
     u ! talk ! NUMBER ;
```

```
   stop

endproc

process network[n]:noexit:=

   n ! conreq ? CALLER:phone_number ? CALLED:phone_number ;
   n ! ring ! CALLED ;
   n ! connect ! CALLED ;
   n ! connect ! CALLER ;
   stop

endproc

process tester[u,n]:noexit:=

   u ! offhook ! num1 ; (* a good test sequence *)
   n ! conreq ! num1 ! num2 ;
   n ! ring ! num2 ;
   u ! answer ! num2 ;
   n ! connect ! num2 ;
   n ! connect ! num1 ;
   u ! talk ! num1 ;
   u ! talk ! num2 ;
   stop

[]

   u ! offhook ! num1 ; (* a bad test sequence *)
   n ! conreq ! num1 ! num2 ;
   n ! ring ! num2 ;
        (* answer action intentionally missing *)
   n ! connect ! num2 ;
   n ! connect ! num1 ;
   u ! talk ! num1 ;
    u ! talk ! num2 ;
    stop

endproc

endspec
```

Appendix 2: Some ACT-ONE data types

```
specification ACT_ONE: noexit

library Set, Boolean, DecDigit, NaturalNumber, BasicNaturalNumber
endlib

type    UsersList is DecDigit, Boolean, BasicNaturalNumber

sorts   List

opns    {}                                        : -> List
        InsertL, RemoveL                         : DecDigit, List -
> List
        _IsIn_, _NotIn_                          : DecDigit, List -
> Bool
        _Union_, _Ints_, _MinusL_                 : List, List ->
List
        _eq_, _ne_, _Includes_, _IsSubsetOf_     : List, List ->
Bool
        Card                                      : List -> Nat
        FirstL                                   : List -> DecDigit

eqns    forall  x, y    : DecDigit,
                s, t    : List

        ofsort  List

                x IsIn InsertL(y,s) =>
       InsertL(x, InsertL(y,s))  = InsertL(y,s)                ;

        RemoveL(x, {}) = {}                                     ;
        RemoveL(x, InsertL(x,s))  = s                           ;
                x ne y =>
        RemoveL(x, InsertL(y,s))  = InsertL(y,
RemoveL(x,s))        ;

        {} Union s              = s                             ;
        InsertL(x,s) Union t    = InsertL(x,s Union t)          ;

        {} Ints s               = {}                            ;
                x IsIn t =>
        InsertL(x,s) Ints t     = InsertL(x,s Ints t)           ;
                x NotIn t =>
```

```
        InsertL(x,s) Ints t      = s Ints t                     ;

        s MinusL {}              = s                            ;
        s MinusL InsertL(x, t)    = RemoveL(x,s) MinusL
t           ;

        ofsort  Bool
        x IsIn {}                = false                        ;
                x eq y =>
        x IsIn InsertL(y,s)      = true                         ;
                x ne y =>
        x IsIn InsertL(y,s)      = x IsIn s                     ;

        x NotIn s                = not(x IsIn s)                ;

        s Includes {}            = true                         ;
        s Includes InsertL(x,t)  = (x IsIn s) and (s Includes t) ;

        s IsSubsetOf t           = t Includes s                 ;

        s eq t                 = (s Includes t) and (t Includes s);

        s ne t                   = not(s eq t)                  ;

        ofsort  DecDigit

        FirstL({})                 = 0                        ;
                x NotIn s =>
        FirstL(InsertL(x,s))       = x                          ;

        ofsort  Nat
        Card ({})                 = 0                           ;
                x NotIn s =>
        Card (InsertL(x,s))        = Suc-
c(Card(s))                    ;
endtype

type    EventSeq is NaturalNumber, Boolean
sorts   EventSeq
opns    0, 1, 2, 3, 4, 5, 6, 7, 8, 9            : -> EventSeq
        _eq_, _ne_, _lt_, _le_, _ge_, _gt_      : EventSeq, Event-
Seq -> Bool
        NatNum                                  : EventSeq -> Nat

eqns    forall  x, y, z : EventSeq
```

```
        ofsort  Nat

        NatNum(0)       = 0                     ;
        NatNum(1)       = Succ(NatNum(0))       ;
        NatNum(2)       = Succ(NatNum(1))       ;
        NatNum(3)       = Succ(NatNum(2))       ;
        NatNum(4)       = Succ(NatNum(3))       ;
        NatNum(5)       = Succ(NatNum(4))       ;
        NatNum(6)       = Succ(NatNum(5))       ;
        NatNum(7)       = Succ(NatNum(6))       ;
        NatNum(8)       = Succ(NatNum(7))       ;
        NatNum(9)       = Succ(NatNum(8))       ;

        ofsort  Bool

        x eq y  = NatNum(x) eq NatNum(y)        ;
        x ne y  = NatNum(x) ne NatNum(y)        ;
        x lt y  = NatNum(x) lt NatNum(y)        ;
        x le y  = NatNum(x) le NatNum(y)        ;
        x ge y  = NatNum(x) ge NatNum(y)        ;
        x gt y  = NatNum(x) gt NatNum(y)        ;

endtype

type DigitPair is DecDigit, Boolean

   sorts DigitPair
   opns
       None    : -> DecDigit
       None    : -> Nat
       Pair:DecDigit,DecDigit -> DigitPair
       _eq_: DigitPair, DigitPair -> Bool

   eqns

       forall f1,f2,s1,s2:DecDigit , f3:Nat

       ofsort Bool
          f1 eq None     = false;
          None     eq f1 = false;
          Pair(f1,s1) eq Pair(f2,s2) = (f1 eq f2) and (s1 eq s2);

       ofsort Bool
  f3 eq None     = false;
          None     eq f3 = false;
```

```
       ofsort Nat
  NatNum(None    ) = None
endtype

type SetOfPairs  is DigitPair
     sorts SetOfPairs
     opns
         EmptyPair: -> SetOfPairs
         InsertPair:   DigitPair,SetOfPairs  -> SetOfPairs
         RemovePair:   DigitPair,SetOfPairs  -> SetOfPairs
         IsPairIn : DigitPair,SetOfPairs  -> Bool
        SecondElement :DecDigit,SetOfPairs  -> DecDigit (* given
the first element, find the second element *)
        FirstElement :DecDigit,SetOfPairs  -> DecDigit  (* given
the second element, find the first element *)
         IsPairNotIn: DigitPair,SetOfPairs -> Bool

     eqns
        forall x,y:DigitPair, s:SetOfPairs , f1,f2,s1,s2:DecDigit

         ofsort SetOfPairs
            InsertPair(x,InsertPair(x,s)) = InsertPair(x,s);
            RemovePair(x, EmptyPair) = EmptyPair;
            RemovePair(x,InsertPair(x,s)) = s;
            not(x eq y) =>
            RemovePair(x,InsertPair(y,s)) = InsertPair(y,Remove-
Pair(x,s));

         ofsort Bool
            IsPairIn(x,EmptyPair) = false;
            IsPairIn(x,InsertPair(y,s)) = (x eq y) or (IsPair-
In(x,s));
            IsPairIn(x,RemovePair(x,s)) = false;
            not(x eq y) =>
            IsPairIn(x,RemovePair(y,s)) = IsPairIn(x,s);
            IsPairNotIn(x,s) = not(IsPairIn(x,s));

         ofsort DecDigit
            SecondElement (f1,EmptyPair) = None     ;
            SecondElement (f1,InsertPair(Pair(f1,s1),s)) = s1;
            f1 ne f2 => SecondElement (f1,InsertPair(-
Pair(f2,s2),s)) = SecondElement (f1,s);

            FirstElement (f1,EmptyPair) = None     ;     (* not
tested *)
            FirstElement (s1,InsertPair(Pair(f1,s1),s)) = f1;
```

```
            s1 ne s2 => FirstElement (s1,InsertPair(-
Pair(f2,s2),s)) = FirstElement (s1,s);
endtype

type DecSet0 is Set
     actualizedby DecDigit
     using
     sortnamesDecDigit for Element
              Bool for FBool
endtype

type DecSet is DecSet0
     renamedby
     sortnamesDecSet for Set
endtype

type Signals is Boolean, NaturalNumber
     sorts  Signal
     opns     OffHookToCall,
              OffHookToAns,
              OffHookToRemind,
              FlashHookOn,
              FlashHookOff,
              HangsUp,
              TalksTo,
              RingsFrom,
              RingsBackFrom,
              Dials,
              GetsTone,
              GetsCwTone,
              GetsBusySignal,
              Rering
                              :-> Signal

              _IsEq_          : Signal, Signal -> Bool
              _IsNe_          : Signal, Signal -> Bool
              Map             : Signal -> Nat

     eqns     forall x, y: Signal

              ofsort  Nat
                      Map   (OffHookToCall)    =  0;
                     Map   (OffHookToAns)     =  Succ(Map   (Off-
HookToCall));
                     Map   (OffHookToRemind)  =  Succ(Map   (Off-
HookToAns));
```

```
                    Map   (FlashHookOn)       =  Succ(Map   (Off-
HookToRemind));
                   Map   (FlashHookOff)      =  Succ(Map   (Flash-
HookOn));
                   Map   (HangsUp)           =  Succ(Map   (Flash-
HookOff));
                   Map   (TalksTo)           =  Succ(Map   (Hang-
sUp));
                    Map   (RingsFrom)         =  Succ(Map
(TalksTo));
                   Map   (RingsBackFrom)     =  Succ(Map   (Rings-
From));
                   Map   (Dials)             =  Succ(Map   (Rings-
BackFrom));
                    Map   (GetsTone)          =  Succ(Map
(Dials));
                    Map   (GetsCwTone)        =  Succ(Map   (Get-
sTone));
                    Map   (GetsBusySignal)    =  Succ(Map
(GetsCwTone));
                   Map   (Rering        )    =  Succ(Map   (Gets-
BusySignal));

              ofsort  Bool
                      x IsEq y =  Map   (x) eq Map   (y);
                      x IsNe y =  Map   (x) ne Map   (y);
endtype


behaviour
  stop
endspec
(* Minimal LOTOS Specification !!!! *)
```