PC LOTOS

A hands-on LOTOS tutorial

Bernard Stepien

LOTOS (Language Of Temporal Ordering Specification) is an ISO standard (ISO 8807) for the purpose of protocol specification. The language itself is based on a limited set of operators to specify behaviours and on the Abstract Data Type (ADT) language ACT ONE to specify data abstraction. LOTOS is taught in various academic institutions around the world to an audience of both university students and Industry. The motivation behind this hands-on tutorial came as a result of teaching LOTOS to Industry for the Protocol Research Group of the University of OTTAWA. Using the various LOTOS tools requires an in depth knowledge of the language, and even small specifications including limited data types can become a major hurdle to the first time user. This is an unfortunate situation, because LOTOS in itself is a relatively simple language using few basic operators. Consequently, the need for a tool that uses only a subset of the language seemed to be a prime requirement for the widespread use of this language.

The need for such a subset is not new. Basic LOTOS (derived from CCS and CSP) represent the "control" component of LOTOS. ACT ONE was added to this kernel to gain the ability of specifying data abstraction.Unfortunately, the use of basic LOTOS is very limited because there are no protocols without data. Also, many aspects of LOTOS, for example the concept of reusability of processes, representing addresses, etc., cannot be demonstrated without some elementary data.

Another consideration in the development of this subset of LOTOS was the lack of tools for unsophisticated platforms. Most of the currently available tools can be operated only on workstations requiring large amounts of memory.

A PC based LOTOS tool allows exposure of LOTOS to a much wider audience.

There is a good selection of tutorials available in the literature, but their use by novices is sometimes limited by the lack of opportunity for hands-on experience. This tutorial is very simple and comes with a good selection of examples that can be run with the PCLOTOS tool setting a platform for more adventurous exploration of this language.

Finally, a substantial motivation for the development of the PC LOTOS tool was to have a tool to work on LOTOS specifications while travelling. As a matter of fact, most of the development of PCLOTOS was achieved on a laptop

computer on a sailboat sailing between Gibraltar and the Azores in the middle of the Atlantic ocean.

Contents

PART I
PCLOTOS DOCUMENTATION

Chapter 1 PCLOTOS OPERATION

Copy the PCLOTOS system diskette on your hard drive or make a duplicate copy on a floppy diskette. Store your original diskette safely away and work exclusively on your copy.

Section 1.1 system start-up

type **PCLOTOS**

The following popup menu will appear

EDIT A LOTOS SPECIFICATION

EXECUTE A LOTOS SPECIFICATION

EXIT

Using the arrows, move the green cursor bar to the appropriate selection and press <enter> to activate your selection

1.1.1 editing a specification

Editing an already existing LOTOS specification

A directory of LOTOS specification files will appear. Files containing examples corresponding to the ones described in this documentation have been provided to avoid retyping.

Pick one of the files using the arrows and page up/down and press enter to load.

Creating a new specification

when the directory of specifications appears, press Escape and a skeleton specification will appear. You are now in an editor that has the same functions as the turbo Pascal or turbo Prolog editors. As a reminder, all the editing functions are similar to Wordstar functions.

Editor functions:

arrows: move up or down or left or right one character.

CTRL-T: delete word

CTRL-Y: delete line

copy block:

CTRL-K/B: begin block

CTRL-K/K: end block

CTRL-K/C: paste block at current cursor location

CTRL-K/V: move block at current cursor location

F7 : import text from another ASCII file. The imported text will be inserted at the current cursor location. A new window will appear prompting for a file name. Enter the file name. A new window containing the beginning of the imported file will appear. Browse through it using the same above described editor functions. The **F7** key is used again to delimit the start and the end of the selected block of text to import.

ESC: terminate editing and compile.

Error messages will appear if relevant. If your specification is syntactically correct, you will be returned to the main menu.

1.1.2 list of example files

This hands-on tutorial features a number of ready to use examples to explain various LOTOS operators and concepts. All the examples listed in this tutorial are available on your PCLOTOS system diskette and should be copied on your work disk. To use them, follow this sequence of actions:

— select “editing a LOTOS specification” from the main menu

— select one of the files from the popup menu.

— once you see a specification, press Escape to trigger its compilation. A file name prompt will appear. To re-save your specification in the same file, press enter or enter a new file name if you want to preserve the original file.

— select “LOTOS specification execution” from the main menu

— select the same file name from the popup menu.

— select a process name from the process popup menu.

— after execution of an action or display of a behaviour, press <enter> to continue execution.

FILE

LEX1_1.L : action_prefix example

LEX1_2.L : non deterministic choice example

LEX1_3.L : independent parallelisms example (interleave operator)

LEX1_4.L : dependent parallelisms example (synchronization operator)

LEX1_5.L : mixed parallelisms example (interleave and synchronization)

LEX1_6.L : disable operator example

LEX1_7.L : enable with successful termination example

LEX1_8.L : enable with unsuccessful termination example

LEX1_9.L : resolving non determinism example

LEX2_1.L : usage of data example

LEX2_2.L : data and deadlock in parallelism example

LEX2_3.L : value passing in parallelism example

LEX2_4.L: guard operator example

LEX2_5.L : predicates in resolving non determinism example

LEX3_1.L : process definitions example

LEX3_2.L : telephone process recursion example

LEX3_3.L: transport service recursion example
LEX3_4.L: relabelling vs substitution of gates example
LEX4_1.L: using abstract data types to represent databases
LEX4_2.L: State oriented specification example
LEX4_3.L: Constraint oriented specification example
LEX4_4.L: Busy state representation example

1.1.3 executing a specification

PCLOTOS has been organized as a learning tool. It will show you all the possible paths of actions that you can obtain from a LOTOS specification by building an execution tree showing all the alternatives that your behaviour expression can generate.

behaviour expression: is a state in which the system is at a given time.

There are two screens:

the LOTOS execution tree screen:

shows the alternate sequences of actions step by step. When this screen appears, you can look at it until you decide to either proceed to the next available action by pressing **<enter>** or look at the behaviour screen by pressing the **right arrow key.** The tree is produced with a depth first order.

the LOTOS behaviour expression screen:

It shows two different kinds of behaviours:

- the LOTOS behaviour that has produced the action that appeared in the LOTOS execution tree screen.

- the LOTOS behaviour expression that results from the execution of an action.

The system pauses after displaying each type of behaviour expressions. Press **<enter>** to proceed, or press the **left arrow** to view the execution tree screen.

Interrupting an execution: press the **<ESC>** key when looking at the execution tree screen only.

Chapter 2 BASIC LOTOS

LOTOS is based on an algebra. It has operators that perform operations on actions or behaviour expressions to express the ordering of events in time.

syntax:

a LOTOS specification is composed of:

the specification name, parameters and functionality

the key word "behaviour"

a LOTOS behaviour expression

the key word "endspec"

This is the basic skeleton that will appear in the editor when attempting to create a new specification.

specification [<gate-list>]:noexit

behaviour

....

endspec

Section 2.1 sequence operator: action prefix ";"

This is the most basic LOTOS operator depicting the fact that the event on the left side of the operator will precede in time all other events involved in the behaviour expression on the right side of the operator.

NOTE: do not confuse this sequence operator ";" with a statement delimiter like in PASCAL or PLI for example. There is no such thing as statements in LOTOS.

The following specification is contained in file "LEX1_1.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification action_prefix_ex[wake_up,
                               breakfast,work,lunch,work,
                                relax,dinner,tv,sleep ]
                                                    :noexit
      behaviour
       wake_up ; breakfast ; work ; lunch ;
       work ; relax ; dinner ; tv ; sleep ; stop
endspec
```

This example is self explanatory.

REMARK: note the list of actions between square brackets after the specification name "action_prefix_ex". If you modify this specification by adding more actions, do not forget to include them in the action declaration list also called "gate-list" by LOTOS experts.

You can execute this specification and see a relative linear execution tree.

```
wake_up ;
 |     breakfast ;
 |     |    work ;
 |     |    |    lunch ;
 |     |    |    |     work ;
 |     |    |    |     |    relax ;
 |     |    |    |     |    |    dinner ;
 |     |    |    |     |    |    |    tv ;
 |     |    |    |     |    |    |    |    sleep
```

Section 2.2 non deterministic choice operator "[]"

This is an operator that depicts a choice between two behaviour expressions. It is non deterministic because the system itself cannot decide which path it will follow. The environment can however decide which way it will go. When executing the specification with PCLOTOS, the user is playing the role of the environment.

Important note: When selecting one of the two alternative behaviour expressions offered by the choice operator, you will be committed to the branch that has been chosen. Consequently, only the remaining events or actions of the selected branch are considered as next possible actions. You no longer can switch to the other branch. It has the same function as a railroad switch.

example 1.2 is showing two alternatives of someone's personal life:

- a typical work day sequence of events

- a typical holiday sequence of events with some further choice of how to spend that holiday.

The following specification is contained in file "LEX1_2.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification choice_ex[wake_up,breakfast,work,lunch,
                        work, relax,dinner,tv,sleep,
                      golf,brunch,tennis,movie]:noexit
   behaviour
     wake_up ; breakfast ; work ; lunch ;
     work ; relax ; dinner ; tv ; sleep ; stop
     []
     wake_up ; brunch ;
     (
       golf ; movie ; sleep ; stop
       []
       tennis ; tv ; sleep ; stop
     )
endspec
```

Execution tree

The execution tree shows all possible paths of actions one can draw from the above specification. We can expect two main branches with further sub-branches for the second branch.

This tree is obtained using the PCLOTOS interpreter, which shows both the tree in the making and the various states or behaviour expressions resulting from the execution of an action.

```
wake_up
|   breakfast
|   |   work
|   |   |   lunch
|   |   |   |   work
|   |   |   |   |   relax
|   |   |   |   |   |   dinner
|   |   |   |   |   |   |   tv
|   |   |   |   |   |   |   |   sleep
wake_up
|   brunch
|   |   golf
|   |   |   movie
|   |   |   |   sleep
|   |   tennis
|   |   |   tv
|   |   |   |   sleep
```

The vertical lines help to understand the various alternatives. First we have an alternative between a wake up for a week day and a wake up for a holiday. The week day branch is composed of a linear sequence of actions typical of a routine work day. The holiday branch is more diversified and one can see a vertical alignment for golf and tennis meaning that these are alternate activities.

One can also see that if one plays tennis, the specification dictates that he should watch television as the next action, preventing him from going to a movie. This illustrates the concept of commitment to a branch that is the characteristic of the non deterministic choice operator.

This commitment can be easily observed on the resulting behaviour expression following the execution of an action. For example if one picks a holiday wake up,

the week day branch will disappear from the resulting behaviour expression. This is easily understandable, because if one decides he is in a holiday, he definitely is not in a work day. (This, before cellular phones and laptop computers were invented!)

The following behaviours can be observed in the behaviours window when executing the above specification. You can switch back and forth between the execution tree window and the behaviour window using the left and right arrow. press <enter> to proceed to the next available action

```
resulting behaviour:
performed action: wake_up ---->
 brunch ;
 ( golf ;  movie ;  sleep ;  stop
[]
 tennis ;  tv ;  sleep ;  stop
 )
-----------------------------------------------------
infering behaviour: (extracting the next possible action from
                     the above resulting behaviour)
 brunch ;
 ( golf ;  movie ;  sleep ;  stop
[]
 tennis ;  tv ;  sleep ;  stop
 )
=====================================================
resulting behaviour:
performed action: brunch ---->
 ( golf ;  movie ;  sleep ;  stop
[]
 tennis ;  tv ;  sleep ;  stop
 )
-----------------------------------------------------
infering behaviour:
 ( golf ;  movie ;  sleep ;  stop
[]
 tennis ;  tv ;  sleep ;  stop
```

```
 )
=====================================================
resulting behaviour:
performed action: golf ---->
 movie ;  sleep ;  stop
-----------------------------------------------------
infering behaviour:
 movie ;  sleep ;  stop
```

at this point there is only a linear sequence of actions left for the evening program.

Section 2.3 Parallelism or concurrency

One of the most important difference between a traditional computer language and the specification language LOTOS is that LOTOS can handle parallel processing much more easily.

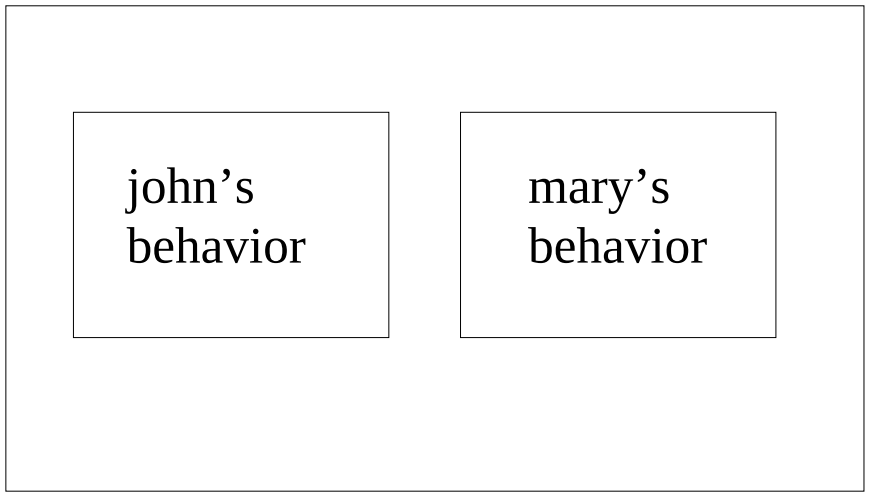
There are three types of parallelism:

- independent parallelism
- dependent parallelism
- mixed parallelism

2.3.1 independence: interleave operator "|||"

This is the case where two processes evolve independently from each other. This means that each process can perform actions in any order relative to each other.

This also means that after executing an action from one process the next possible action could be any action from both processes in parallel.



The following example shows John and Mary doing a cash withdrawal at a banking station where there are two cash dispensers. It is clear that they can perform the required sequence of actions in any order relative to each other.

The following specification is contained in file "LEX1_3.l" on the PCLOTOS system diskette and should be on your working disk.

```
specification interleave_ex [john_arrives,
                             john_inserts_card,
                             john_gets_cash,
                             mary_arrives,
                             mary_inserts_card,
                             mary_gets_cash]:noexit
   behaviour
      john_arrives ;  john_inserts_card ;
      john_gets_cash ; stop
    |||
      mary_arrives ;  mary_inserts_card ;
      mary_gets_cash ; stop
endspec
```

The execution tree resulting from this specification is quite large because it is composed of all the combinations of interleaving John and Mary's actions.

```
john_arrives
|   john_inserts_card
|   |   john_gets_cash
|   |   |   mary_arrives
|   |   |   |   mary_inserts_card
|   |   |   |   |   mary_gets_cash
|   |   mary_arrives
|   |   |   john_gets_cash
|   |   |   |   mary_inserts_card
|   |   |   |   |   mary_gets_cash
|   |   |   mary_inserts_card
|   |   |   |   john_gets_cash
|   |   |   |   |   mary_gets_cash
|   |   |   mary_gets_cash
|   |   |   |   |   john_gets_cash
|   mary_arrives
|   |   john_inserts_card
|   |   |   john_gets_cash
|   |   |   |   mary_inserts_card
|   |   |   |   |   mary_gets_cash
```

```
|   |   |   mary_inserts_card
|   |   |   |   john_gets_cash
|   |   |   |   |   mary_gets_cash
|   |   |   |   mary_gets_cash
|   |   |   |   |   john_gets_cash
|   |   mary_inserts_card
|   |   |   john_inserts_card
|   |   |   |   john_gets_cash
|   |   |   |   |   mary_gets_cash
|   |   |   |   mary_gets_cash
|   |   |   |   |   john_gets_cash
|   |   |   mary_gets_cash
|   |   |   |   john_inserts_card
|   |   |   |   |   john_gets_cash
mary_arrives
|   john_arrives
|   |   john_inserts_card
|   |   |   john_gets_cash
|   |   |   |   mary_inserts_card
|   |   |   |   |   mary_gets_cash
|   |   |   mary_inserts_card
|   |   |   |   john_gets_cash
|   |   |   |   |   mary_gets_cash
|   |   |   |   mary_gets_cash
|   |   |   |   |   john_gets_cash
|   |   mary_inserts_card
|   |   |   john_inserts_card
|   |   |   |   john_gets_cash
|   |   |   |   |   mary_gets_cash
|   |   |   |   mary_gets_cash
|   |   |   |   |   john_gets_cash
|   |   |   mary_gets_cash
|   |   |   |   john_inserts_card
|   |   |   |   |   john_gets_cash
|   mary_inserts_card
|   |   john_arrives
```

```
|   |   |   john_inserts_card
|   |   |   |   john_gets_cash
|   |   |   |   |   mary_gets_cash
|   |   |   |   mary_gets_cash
|   |   |   |   |   john_gets_cash
|   |   |   mary_gets_cash
|   |   |   |   john_inserts_card
|   |   |   |   |   john_gets_cash
|   |   mary_gets_cash
|   |   |   john_arrives
|   |   |   |   john_inserts_card
|   |   |   |   |   john_gets_cash
```

The following excerpt of the resulting behaviour window shows that while John performs his actions, Mary's actions are still available. Note that we have used the option to show only the first action of a sequence of action options. This option is used to show which actions are available as the next possible actions only. It hides all actions that will be available only after one of the next possible actions have been executed. It is a very handy feature when executing large specifications.

```
inferring behaviour:
 john_arrives ;  .....
|||
 mary_arrives ;  .....
==================================================
resulting behaviour:
performed action: john_arrives ---->
 john_inserts_card ;  .....
|||
 mary_arrives ;  .....
--------------------------------------------------
infering behaviour:
 john_inserts_card ;  .....
|||
 mary_arrives ;  .....
```

```
=====================================================
resulting behaviour:
performed action: john_inserts_card ---->
 john_gets_cash ;  .....
|||
 mary_arrives ;  .....
-----------------------------------------------------
infering behaviour:
 john_gets_cash ;  .....
|||
 mary_arrives ;  .....
   etc ...
```

2.3.2 parallel composition synchronization operator "||"

This operator means that the two processes in parallel have to agree on each single action they can perform. The sequence of actions must be identical in both processes.

In example 1.4 — file "LEX1_4.L" we can observe the behaviour of a user in parallel to a cash dispenser. In the first alternative, they follow the same sequence of actions. In the second alternative, the customer attempts to enter the amount before entering its P.I.N., and deadlock ensues.

```
specification synchronization_ex [ inserts_card ,
                         enter_PIN , enter_amount ,
                         press_green_button ]:noexit
   behaviour
  (
      inserts_card ; enter_PIN ; enter_amount ;
      press_green_button ; stop
    ||
      inserts_card ; enter_PIN ; enter_amount ;
      press_green_button ; stop
  )
 []
  (
      inserts_card ; enter_amount ; enter_PIN ;
      press_green_button ; stop
    ||
      inserts_card ; enter_PIN ; enter_amount ;
      press_green_button ; stop
  )
endspec
```

The resulting execution tree will clearly show the deadlock of the second alternative:

```
inserts_card
|   enter_PIN
|   |   enter_amount
|   |   |   press_green_button
inserts_card    <---- Deadlock, no further actions
```

On the behaviour window we can observe how an action is executed simultaneously in both parallel processes:

```
=====================================================
resulting behaviour:
performed action: inserts_card ---->
 enter_PIN ;  enter_amount ;  press_green_button ;
   stop
|[inserts_card,enter_PIN,enter_amount,
                                  press_green_button]|
 enter_PIN ;  enter_amount ;  press_green_button ;
  stop
=====================================================
resulting behaviour:
performed action: enter_PIN ---->
 enter_amount ;  press_green_button ;  stop
|[inserts_card,enter_PIN,enter_amount,
                                  press_green_button]|
 enter_amount ;  press_green_button ;  stop
=====================================================
resulting behaviour:
performed action: enter_amount ---->
 press_green_button ;  stop
|[inserts_card,enter_PIN,enter_amount,
                                 press_green_button]|
 press_green_button ;  stop
=====================================================
resulting behaviour:
performed action: press_green_button ---->
 stop
|[inserts_card,enter_PIN,enter_amount,
                                   press_green_button]|
 stop
```

2.3.3 mixed synchronization and interleave operator: "|[...]|"

This operator is used more frequently, because processes do not always agree with each other. They perform various actions independently and at times they rendezvous to exchange information, etc...

The common actions on which both processes must be synchronous are indicated in a list between the parallel bars.

Example 1.5 — file "LEX1_5.L", shows the behaviour of John and Mary, who wake up and work in the morning independently and decide to go to lunch together and work together in the afternoon. They then go home independently.

```
specification mixed_parallelism [ john_wake_up,
                                  john_works,john_goes_home,
                                  mary_wake_up,mary_works,
                                  mary_goes_home,
                                  lunch_together,
                                  work_together ]:noexit
  behaviour
    john_wake_up ; john_works ;  lunch_together ;
    work_together ;  john_goes_home ; stop
   |[lunch_together,work_together]|
    mary_wake_up ; mary_works ;  lunch_together ;
    work_together ; mary_goes_home ; stop
endspec
```

This is the resulting execution tree for the above specification. It shows the various alternatives resulting from the interleaveaving of John and Mary's independent actions before they get together for lunch and the subsequent alternatives once they have lunched and worked together.

```
john_wake_up
|   john_works
|   |   mary_wake_up
|   |   |   mary_works
|   |   |   |   lunch_together
|   |   |   |   |   work_together
|   |   |   |   |   |   john_goes_home
|   |   |   |   |   |   |   mary_goes_home
```

```
|   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   |   john_goes_home
|   mary_wake_up
|   |   john_works
|   |   |   mary_works
|   |   |   |   lunch_together
|   |   |   |   |   work_together
|   |   |   |   |   |   john_goes_home
|   |   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   |   john_goes_home
|   |   mary_works
|   |   |   john_works
|   |   |   |   lunch_together
|   |   |   |   |   work_together
|   |   |   |   |   |   john_goes_home
|   |   |   |   |   |   |   mrygoes_home
|   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   |   john_goes_home
mary_wake_up
|   john_wake_up
|   |   john_works
|   |   |   mary_works
|   |   |   |   lunch_together
|   |   |   |   |   work_together
|   |   |   |   |   |   john_goes_home
|   |   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   |   john_goes_home
|   |   mary_works
|   |   |   john_works
|   |   |   |   lunch_together
|   |   |   |   |   work_together
|   |   |   |   |   |   john_goes_home
|   |   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   mary_goes_home
```

```
|   |   |   |   |   |   |   john_goes_home
|   mary_works
|   |   john_wake_up
|   |   |   john_works
|   |   |   |   lunch_together
|   |   |   |   |   work_together
|   |   |   |   |   |   john_goes_home
|   |   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   mary_goes_home
|   |   |   |   |   |   |   john_goes_home
```

2.3.4 the internal event “i”

In LOTOS, an action is always the result of an interaction between one or more processes. Sometimes, one needs to specify that an unspecified internal action should occur before further interactions can occur. This internal action has no effect on the rest of the behaviour. In LOTOS terminology we say that it does not have to synchronize. It is represented merely by the letter **“i”**.

The following example is correct and will execute entirely:

a ; i ; b ; c ; stop

||

a ; b ; i ; c ; stop

will produce the valid sequence of events: **a –> i —> b –> i —> c**

The observable sequence, however, is a —> b —> c, in the sense that action i should be considered invisible outside of the system being specified.

Section 2.4 disable operator: “[>”

The disable operator indicates interruption of a process.It is useful to specify events such as interruption of service, common in telecommunication applications. This interruption can occur at any point of the disabled process. The hang-up action is an obvious example of a telephone specification.

The following specification is contained in file "LEX1_6.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification disable_ex [ off_hook,tone,dial,
                        connect,talk, hang_up ]:noexit
  behaviour
    off_hook ;
              (
               tone ; dial ; connect ; talk ; stop
                     [>  hang_up ; stop
              )
endspec
```

Note the scope of the hang-up action. The brackets prevent a hang-up from occuring before an off-hook has been executed. The following execution tree shows that there is the alternative of hanging up the phone for each action after an off_hook action.

```
off_hook
|   tone
|   |   dial
|   |   |   connect
|   |   |   |   talk
|   |   |   |   |   hang_up
|   |   |   |   hang_up
|   |   |   hang_up
|   |   hang_up
|   hang_up
```

Section 2.5 enable operator: “>>”

LOTOS behaviours expression resemble trees. When there are alternatives, one is committed to the alternative. The enable operator allows to merge back to a common path after alternatives.

example:

wake_up ;

(work ; lunch ; ...

[]

go_fishing ; lunch ; ...

)

requires to define twice the sequence “**lunch ; ...** “.

Using the enable operator, this can be reduced to:

wake_up ; (work ; exit [] go_fishing ; exit) >> **lunch ; ...**

Note the word “**EXIT**” which means **successful termination** in LOTOS.

example 1.7 — file “LEX1_7.L” shows an alternative morning program between work and golf that both can lead to lunch with some further alternative to work or play tennis in the afternoon that can also lead to relaxing for the rest of the day.

```
specification enable_ex[wake_up,breakfast,
                        work,lunch,work,relax,dinner,
                        sleep,golf,tennis]:noexit
   behaviour
     wake_up ;
     breakfast ;
     ( work ;  exit [] golf ; exit )
             >>
               lunch ;
               ( work ; exit [] tennis ; exit )
                     >>
                        relax ; dinner ; sleep ; stop
endspec
```

The execution tree of the above specification shows the various alternative paths of actions:

```
wake_up
|   breakfast
|   |   work
|   |   |   exit
|   |   |   |   lunch
|   |   |   |   |   work
|   |   |   |   |   |   exit
|   |   |   |   |   |   |   relax
|   |   |   |   |   |   |   |   dinner
|   |   |   |   |   |   |   |   |   sleep
|   |   |   |   |   tennis
|   |   |   |   |   |   exit
|   |   |   |   |   |   |   relax
|   |   |   |   |   |   |   |   dinner
|   |   |   |   |   |   |   |   |   sleep
|   |   golf
|   |   |   exit
|   |   |   |   lunch
|   |   |   |   |   work
|   |   |   |   |   |   exit
|   |   |   |   |   |   |   relax
|   |   |   |   |   |   |   |   dinner
|   |   |   |   |   |   |   |   |   sleep
|   |   |   |   |   tennis
|   |   |   |   |   |   exit
|   |   |   |   |   |   |   relax
|   |   |   |   |   |   |   |   dinner
|   |   |   |   |   |   |   |   |   sleep
```

Unsuccessful termination

In the next example, there is a choice between working and skiing in the morning, with the unfortunate event of breaking a leg which ruins the rest of the day. This is a case of unsuccessful termination preventing further actions despite the enable operator.

The following specification is contained in file "LEX1_8.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification enable_ex[wake_up,breakfast,
                        work,lunch,work,relax,dinner,
                        break_a_leg,sleep,ski,golf,
                                    tennis]:noexit
   behaviour
     wake_up ;
     breakfast ;
     ( work ; exit
       []
       ski ; break_a_leg ; stop
     )
       >>
         lunch ;
                ( work ; exit [] tennis ; exit )
                      >>
                         relax ; dinner ; sleep ;
stop
endspec
```

The execution tree of this specification is reduced compared to the previous tree that did not include an unsuccessful termination:

```
wake_up
|   breakfast
|   |   work
|   |   |   exit
|   |   |   |   lunch
|   |   |   |   |   work
|   |   |   |   |   |   exit
|   |   |   |   |   |   |   relax
|   |   |   |   |   |   |   |   dinner
|   |   |   |   |   |   |   |   |   sleep
|   |   |   |   |   tennis
|   |   |   |   |   |   exit
|   |   |   |   |   |   |   relax
|   |   |   |   |   |   |   |   dinner
|   |   |   |   |   |   |   |   |   sleep
|   |   ski
|   |   |   break_a_leg <---- unsuccessful termination
```

Section 2.6 resolving non determinism

The non deterministic choice operator provides alternatives that the system cannot resolve by itself. The environment will determine which path the behaviour will follow. If such a behaviour is in parallel with another process, that process may dictate the path to follow.

The following specification is contained in file "LEX1_9.L" on the PCLOTOS system diskette and should be on your working disk.

In the following example of an interaction between a phone process and a controller, we can see that the controller will force the dial_number action to be executed and no ring will then occur for this call initiator phone, because the controller side does not offer a ring action.

```
specification telephone [ off_hook,tone,dial_number,
                          ring,answer,connect,talk
                                                ]:noexit
     behaviour
     (
      off_hook ; tone ; dial_number ; connect ;
      talk ; stop
      []
      ring  ; answer ; connect ; talk ; stop
     )
   |[dial_number,ring,connect]|
     dial_number ; connect ; stop
endspec
```

The only possible path for this specification is:

```
off_hook
|   tone
|   |   dial_number
|   |   |   connect
|   |   |   |   talk
```

Chapter 3 LOTOS AND DATA

Data are handled in ISO LOTOS by a complex Abstract Data Type (ADT) definition language called ACT ONE.

ACT One is a very powerful language, but it may present difficulties for novices.

Therefore, In this PC version of LOTOS we have chosen to relieve the user of the Abstract Data Type burden that is unnecessary to understand the “process” part of LOTOS.

Data are consequently considered as **plain character string** information.

Section 3.1 action denotation

an action or event in LOTOS is composed of two main components:

- an interaction point between two processes

- value offers

example:

controller ! 236-4563 ! ring

is an event that occurs at the interaction point named “controller”, and provides the phone number and the ring as values.

Data in LOTOS are used for three main purposes:

- as **identifiers** of entities that communicate via the same interaction point.

- as **indication** of the type of event that is occurring (primitives)

- finally as **data in the traditional sense** of input / output.

there are two symbols to indicate the type of offer:

! : means that the value is offered (similar to the output concept)

? : means that the value is requested (similar to the input concept) this symbol is used only in connection with a variable name.

The following example shows how the simple phone system can be rewritten using data. Most actions are merely providing phone number identification and type of action through data, and the dial action is requesting a Called Number to be supplied by the environment, in this case the user.

The following specification is contained in file "LEX2_1.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification data_ex [ phone,controller ]:noexit
  behaviour
    phone ! 5641234 ! off_hook ;
    (
       phone ! 5641234 ! tone ;
       controller ! 5641234 ! dial ? Called_Number ;
       controller ! 5641234 ! connect ! Called_Number ;
       phone ! 5641234 ! talk ;
       phone ! Called_Number ! talk ;
       stop
             [>  phone ! 5641234 ! hang_up ; stop
     )
endspec
```

Section 3.2 parallelism and data

Here LOTOS is different from the traditional concept of input / output of traditional computer languages.

When two processes communicate, they are interested in two aspects:

- to agree on a common action before proceeding further.

- to exchange information (value passing)

agreement synchronization:

restaurant ! lunch ; work ; ...

|[restaurant]|

restaurant ! lunch ; gohome ; ...

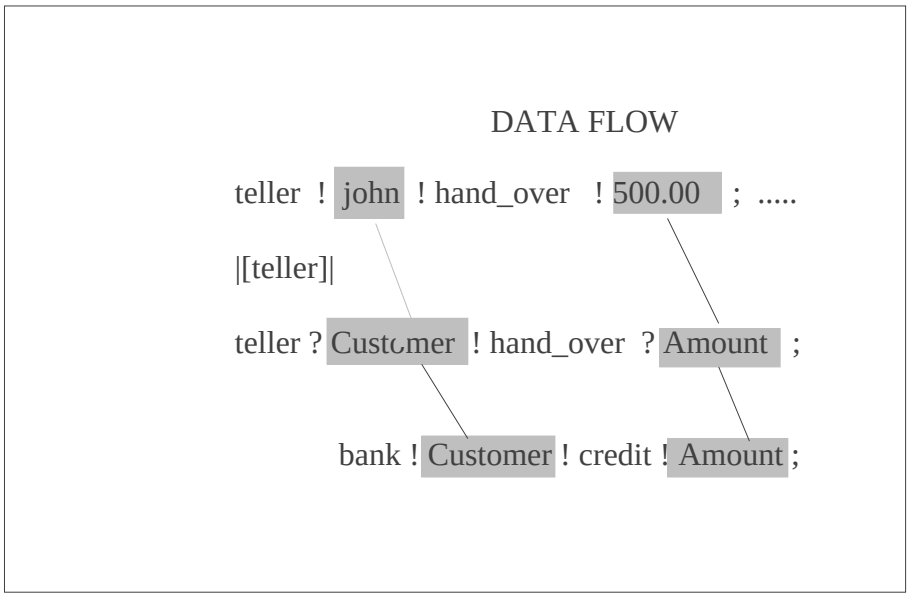
value passing synchronization:

teller ! john ! hand_over ! 500.00 ; ...

|[teller]|

teller ? Customer ! hand_over ? Amount ;

bank ! Customer ! credit ! Amount ; ...



In this example, an action occurs between a customer and a bank at an interaction point called "teller". The customer provides his or her identity and the amount of money, while the bank is requesting the identity and the amount of money before it can proceed and credit the customer's account.

Customer and Money are variables that will contain the value "john" and 500.00 after synchronization.

In the LOTOS world, a variable is referred to as a value declaration. The scope of such a variable is different from what it would be in traditional programming languages, because of the temporal ordering aspect. A variable holds a value for anything that follows that action. Consequently, variables having the same name and occurring in alternate or parallel behaviours do not interfere with each other.

Example:

g1 ? X ; g2 ! X ; stop

[]

g3 ? X ; g4 ! X ; stop

the value contained in X will be different at interaction point g2 than at g4 because the two behaviours cannot mix.

3.2.1 synchronization and deadlock

example 2.2

deadlock is achieved in two ways:

- non matching points of interaction.

- non matching values.

In the following example, john went to the airport (matching interaction point between john and the transportation industry), but the only way to go to Paris was via train while john wanted to go by plane (mismatch on data).

The following specification is contained in file "LEX2_2.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification data_ex [ john,airport,paris  ]:noexit
  behaviour
  john ! taxi ;
  (
      airport ! plane ; paris ! dinner ; stop
```

```
    [[airport]]
      airport ! train ; paris ! dinner ; stop
  )
endspec
```

3.2.2 synchronization and value passing

Using the classic telephone example we can observe some interesting examples of value passing.

First the phone interacts with the user to determine what is the calling number via action off_hook, then the user interacts with the phone again to determine the called party number. The phone line then interacts with the controller to provide a connection request between a calling and a called number.

The controller will then interact with another phone, the responding phone, by ringing that phone number, etc...

The following specification is contained in file "LEX2_3.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification telephone [ phone, controller ]:noexit
     behaviour
     (
      phone ! off_hook ? Calling_Num ;
      phone ! Calling_Num ! dial_number ? Called_Num;
      controller ! Calling_Num ! con_req ! Called_Num ;
      controller ! Calling_Num ! connect ;
      phone ! Calling_Num ! talk ;    stop
      |||
      controller ? Respond_Num ! ring  ;
      phone ! Respond_Num ! answer ;
      controller ? Respond_Num ! connect ;
      phone ! Respond_Num ! talk ; stop
     )
   |[controller]|
      controller ? Origin_Num ! con_req
                                ? Destination_Num ;
      controller ! Destination_Num ! ring  ;
```

```
      controller ! Destination_Num ! connect ;
      controller ! Origin_Num ! connect ;  stop
endspec
```

This is the execution tree of such a specification. Note the question marks “?” followed by a value “222” meaning that a value has been provided by the environment (here the user).

```
phone ! off_hook ? 111
|   phone ! 111 ! dial_number ? 222
|   |   controller ! 111 ! con_req ! 222
|   |   |   controller ! 222 ! ring
|   |   |   |   phone ! 222 ! answer
|   |   |   |   |   controller ! 222 ! connect
|   |   |   |   |   |   controller ! 111 ! connect
|   |   |   |   |   |   |   phone ! 111 ! talk
|   |   |   |   |   |   |   |   phone ! 222 ! talk
|   |   |   |   |   |   |   phone ! 222 ! talk
|   |   |   |   |   |   |   |   phone ! 111 ! talk
|   |   |   |   |   |   phone ! 222 ! talk
|   |   |   |   |   |   |  controller ! 111 ! connect
|   |   |   |   |   |   |   |   phone ! 111 ! talk
```

Section 3.3 conditions on data

Like any other language, LOTOS provides the capability to decide on the next course of action with decisions. However, LOTOS departs from traditional programming languages in the way such decisions can be made.

3.3.1 guards "[...] ->

A guard is very similar to an IF statement. It is however different in its scope. A guard precisely guards the entry of a subsequent behaviour. There is however no ELSE concept.

The above telephone example could be rewritten using a guard. The controller process could query a signal, and if the detected signal is a connection request, it then will proceed in ringing the called party number.

example 2.4 — file “LEX2_4.L

```
specification telephone [ phone, controller ]:noexit
  behaviour
    (
     phone ! off_hook ? Calling_Num ;
     phone ! Calling_Num ! dial_number ? Called_Num;
     controller ! Calling_Num ! con_req ! Called_Num;
     controller ! Calling_Num ! connect ;
     phone ! Calling_Num ! talk ;    stop
      |||
     controller ? Respond_Num ! ring  ;
     phone ! Respond_Num ! answer ;
     controller ? Respond_Num ! connect ;
     phone ! Respond_Num ! talk ; stop
    )
  |[controller]|
    controller ? Origin_Num ? Signal ? Destin_Num ;
    [ Signal eq con_req ] ->
             controller ! Destin_Num ! ring  ;
             controller ! Destin_Num ! connect ;
             controller ! Origin_Num ! connect;
             stop
endspec
```

3.3.2 Predicates in actions

The action denotation that we presented above had a missing element: the predicate. The predicate is a condition on data that will enter the free variables of this action.

For example, if John is a vegetarian, an action specifying his behaviour in a restaurant may look like this:

restaurant ! John ? FOOD [FOOD eq vegetables] ; ...

The main function of predicates is in conjunction with synchronization in parallel behaviour. It is another device to resolve non deterministic choices. This is an example of a specification depicting John choosing restaurants:

(

restaurant ! chinese ? Customer ! vegetables ; stop

[]

restaurant ! fast_food ? Customer ! hamburger ; stop

[]

restaurant ! italian ? Customer ! spaghetti ; stop

)

||

restaurant ? TYPE ! john ? FOOD [FOOD eq vegetables] ; go_home ; stop

The evaluation of the predicate on the free variable FOOD in John's action will determine that the only TYPE of restaurant john will choose is Chinese. Note that the three restaurant actions will synchronize with john's action. In this case we have a bilateral value passing. Variable TYPE will receive the values describing the type of restaurant, Customer will always be passed the value john, and FOOD will be passed the various types of food. The only tuple for which the synchronization will succeed is however the one for which the predicate on FOOD is satisfied.

The following specification is contained in file "LEX2_5.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification going_out [restaurant,theatre]:noexit
  behaviour
  (
   (
    restaurant ! chinese ? Customer ! vegetables ;
    exit
     []
    restaurant ! fast_food ? Customer ! hamburger ;
    exit
     []
    restaurant ! italian ? Customer ! spaghetti ;
    exit
   )
     >>
       (
        theatre ! rambo ? customer ! violence ; stop
          []
        theatre ! hair ? customer ! sixties ; stop
          []
        theatre ! splash ? customer ! romantic ; stop
        )
   )
  ||
  restaurant ? TYPE  ! john ? FOOD
                              [FOOD eq vegetables] ;
  theatre    ? MOVIE ! john ? STYLE
                              [STYLE ne violence] ;
  stop
endspec
```

The above specification will produce the following tree where we observe only one choice of restaurant but two alternatives for movies.

```
restaurant ! chinese ! john ! vegetables
|   exit
|   |   theatre ! hair ! john ! sixties
|   |   theatre ! splash ! john ! romantic
```

Chapter 4 STRUCTURING IN LOTOS

Section 4.1 process definition

behaviour expressions can be broken down in processes following top down design principles.

Processes are generic and reusable in two ways:

- reusable interaction points: relabelling

- formal parameters to pass values.

This is a good example of reusability:

```
deliver[paris,rome](bread)
where
    process deliver[point_a,point_b](Item):noexit:=
       point_a ! pick_up ! Item ;
       point_b ! drop_off ! Item ;
       stop
    endproc
```

the above instantiation will generate the sequence:

paris ! pick_up ! bread —> rome ! drop_off ! bread

For example a telephone system can be broken down into two top level components: Phones and a controller. Phones will then be broken down in individual instances of phones that each have a number. Finally a phone process can be broken down into two roles: the call initiator role and the call responder role.

Phone will have a formal parameter containing it’s identification number that will be associated to all its actions.

The following specification is contained in file "LEX3_1.L" on the PCLOTOS system diskette and should be on your working disk.

```
specification telephone[network,user]:noexit
  behaviour
     (
      phone[network,user](111)
         |||
      phone[network,user](222)
     )
     |[network]|
       controller[network]
     where
        process phone[n,u](NUM):noexit:=
           call_initiator_phone[n,u](NUM)
           []
           call_responder_phone[n,u](NUM)
        endproc
        process call_initiator_phone[n,u](NUM):
                                              noexit:=
           u ! NUM ! offhook;
           u ! NUM ! dial ? CDNUM ;
           n ! NUM ! conreq ! CDNUM ;
           n ! NUM ! connect;
           u ! NUM ! talk;
           stop
        endproc
        process call_responder_phone[n,u](NUM):
                                              noexit:=
           n ! NUM ! ring;
           u ! NUM ! answer;
           n ! NUM ! connect;
           u ! NUM ! talk;
           stop
        endproc
        process controller[n] :noexit:=
            n ? NUM   ! conreq ? CDNUM ;
            n ! CDNUM ! ring ;
            n ! CDNUM ! answer ;
```

```
            n ! NUM    ! connect ;
            stop
        endproc
endspec
```

Note the difference in interaction point names. At the highest level we use the names “network” and “user”, while in the process definitions we use the names “n” and “u”. During execution, LOTOS will **re-label** the process definition interaction point names according to the names present in the instantiation of these processes. Consequently gate **“n”** will be re-labelled **“network”** and gate **“u”** will be re-labelled **“user”**.

The execution tree for this structured specification is larger because we have two phones (number 111 and 222) that are in parallel and independent from each other. Consequently for each action of phone 111, there could be an action of phone 222. This sometimes results in incomplete branches. For instance, if phone 222 gets off_hook before the controller rings it, the ring action is no longer available, and a deadlock for that branch occurs. This deadlock in essence represents that the contacted number is busy.

```
user ! 111 ! offhook
|   user ! 111 ! dial ? 222
|   |   network ! 111 ! conreq ! 222
|   |   |   network ! 222 ! ring
|   |   |   |   user ! 222 ! answer
|   |   |   |   |   network ! 222 ! connect
|   |   |   |   |   |   network ! 111 ! connect
|   |   |   |   |   |   |   user ! 111 ! talk
|   |   |   |   |   |   |   |   user ! 222 ! talk
|   |   |   |   |   |   |   user ! 222 ! talk
|   |   |   |   |   |   |   |   user ! 111 ! talk
|   |   |   |   |   |   user ! 222 ! talk
|   |   |   |   |   |   |   network ! 111 ! connect
|   |   |   |   |   |   |   |   user ! 111 ! talk
|   |   |   user ! 222 ! offhook
|   |   |   |   user ! 222 ! dial ? 111
|   |   user ! 222 ! offhook
|   |   |   network ! 111 ! conreq ! 222
|   |   |   |   user ! 222 ! dial ? 111
|   |   |   user ! 222 ! dial ? 111
|   |   |   |   network ! 111 ! conreq ! 222
|   |   |   |   network ! 222 ! conreq ! 111
|   user ! 222 ! offhook
|   |   user ! 111 ! dial ? 222
|   |   |   network ! 111 ! conreq ! 222
|   |   |   |   user ! 222 ! dial ? 111
|   |   |   user ! 222 ! dial ? 111
|   |   |   |   network ! 111 ! conreq ! 222
|   |   |   |   network ! 222 ! conreq ! 111
```

```
|   |   user ! 222 ! dial ? 111
|   |   |   network ! 222 ! conreq ! 111
|   |   |   |   user ! 111 ! dial ? 222
|   |   |   user ! 111 ! dial ? 222
|   |   |   |   network ! 111 ! conreq ! 222
|   |   |   |   network ! 222 ! conreq ! 111
user ! 222 ! offhook
etc ...
```

The following shows the various resulting behaviours after execution of some actions:

```
-----------------------------------------------------
infering behaviour:
 n ! 111 ! connect ;  .....
|||
 phone[n,u](222)
|[n]|
 n ! 222 ! ring ;  .....
=====================================================
resulting behaviour:
performed action: n ! 222 ! ring ---->
 n ! 111 ! connect ;  .....
|||
 n ! 222 ! answer ;  .....
|[n]|
 n ! 222 ! answer ;  .....
-----------------------------------------------------
infering behaviour:
 n ! 111 ! connect ;  .....
|||
 n ! 222 ! answer ;  .....
|[n]|
 n ! 222 ! answer ;  .....
=====================================================
resulting behaviour:
performed action: n ! 222 ! answer ---->
```

```
 n ! 111 ! connect ;  .....
|||
 n ! 222 ! connect ;  .....
|[n]|
 n ! 222 ! connect ;  .....
-----------------------------------------------------
infering behaviour:
 n ! 111 ! connect ;  .....
|||
 n ! 222 ! connect ;  .....
|[n]|
 n ! 222 ! connect ;  .....
=====================================================
resulting behaviour:
performed action: n ! 222 ! connect ---->
 n ! 111 ! connect ;  .....
|||
 u ! 222 ! talk ;  .....
|[n]|
 n ! 111 ! connect ;  .....
-----------------------------------------------------
infering behaviour:
 n ! 111 ! connect ;  .....
|||
 u ! 222 ! talk ;  .....
|[n]|
 n ! 111 ! connect ;  .....
=====================================================
resulting behaviour:
performed action: n ! 111 ! connect ---->
 u ! 111 ! talk ;  .....
|||
 u ! 222 ! talk ;  .....
|[n]|
 stop
```

Section 4.2 recursion

LOTOS provides for tail recursion only.

The following simple example of the transport protocol provides a never ending transport service:

It is found on your work disk under the name **“LEX3_3.L”**, and it shows two typical recursions of the transport service. The **data_transfer** process is recursive and can only be left through the **termination_phase** process. When a termination occurs, its successful termination is represented by an exit, enabling it to further recurse to a new instance of the process **connection_phase**.

```
specification transport [t]:noexit
   behaviour
      connection_phase[t]
        >>
          ( data_transfer[t]  [> termination_phase[t] )
            >> connection_phase[t]
    where
      process connection_phase[t] :exit :=
         initiator_role[t]
          []
         responder_role[t]
      endproc
      process initiator_role[t]:exit:=
          t ! conreq ;
                     (
                       t ! tconconf ; exit
                       []
                       termination_phase[t]
                     )
      endproc
      process responder_role[t]:exit:=
         t ! tcon_ind ;
                     (
                       t ! tcon_resp ; exit
                       []
```

```
                        termination_phase[t]
                    )
      endproc
      process data_transfer[t]:noexit:=
         t ! data_req ; data_transfer[t]
         []
         t ! data_ind ; data_transfer[t]
      endproc
      process termination_phase[t]:exit:=
         t ! discon_req ; exit
         []
         t ! discon_ind ; exit
       endproc
endspec
```

There are two kinds of recursion in the above specification:

— direct recursion as in the process data_transfer.

— indirect recursion as in the behaviour of the specification itself where the activation of the termination phase enables a new connection to be started

The following is an example of an execution tree that can be obtained from the above specification.

WARNING: since the tree is generated depth first, you have to intervene to stop endless recursion on one specific branch of the tree.

press the **'ESC'** key whenever you are in the tree window and you want to prune the execution of the specific branch you are exploring.

The message "!!! pruned" will follow the last action in the branch whenever you press the **"ESC"** key.

```
t ! conreq
|   t ! tconconf
|   |   exit
|   |   |   t ! data_req
|   |   |   |   t ! data_req
|   |   |   |   |   t ! data_req !!! pruned
|   |   |   |   |   t ! data_ind
```

```
|   |   |   |   |   |   t ! data_req
|   |   |   |   |   |   |   t ! data_req !!! pruned
|   |   |   |   |   |   |   t ! data_ind !!! pruned
|   |   |   |   |   |   |   t ! discon_req !!! pruned
|   |   |   |   |   |   |   t ! discon_ind !!! pruned
|   |   |   |   |   |   t ! data_ind
|   |   |   |   |   |   |   t ! data_req !!! pruned
|   |   |   |   |   |   |   t ! data_ind !!! pruned
|   |   |   |   |   |   |   t ! discon_req !!! pruned
|   |   |   |   |   |   |   t ! discon_ind !!! pruned
|   |   |   |   |   |   t ! discon_req    etc ...
|   |   |   |   |   |   t ! discon_ind
|   |   |   |   |   t ! discon_req
|   |   |   |   |   t ! discon_ind
|   |   |   |   t ! data_ind
|   |   |   |   |   t ! data_req
|   |   |   |   |   |   t ! data_req
|   |   |   |   |   |   |   t ! data_req
|   |   |   |   |   |   |   t ! data_ind
|   |   |   |   |   |   |   t ! discon_req
|   |   |   |   |   |   |   t ! discon_ind
|   |   |   |   |   |   t ! data_ind
|   |   |   |   |   |   t ! discon_req
|   |   |   |   |   |   t ! discon_ind
|   |   |   |   |   t ! data_ind
|   |   |   |   |   t ! discon_req
|   |   |   |   |   t ! discon_ind
|   |   |   |   t ! discon_req
|   |   |   |   t ! discon_ind
|   |   |   t ! data_ind
|   |   |   t ! discon_req
|   |   |   t ! discon_ind
|   t ! discon_req
|   t ! discon_ind
t ! tcon_ind
|   t ! tcon_resp
```

```
|   |   exit
|   |   |   t ! data_req
|   |   |   |   t ! data_req
|   |   |   |   t ! data_ind
|   |   |   |   t ! discon_req
|   |   |   |   t ! discon_ind
|   |   |   t ! data_ind
|   |   |   t ! discon_req
|   |   |   t ! discon_ind
|   t ! discon_req
|   t ! discon_ind
```

As an exercise, you can attempt to transform example LEX3_1.L into a recursive telephone. The complete example can be found in file LEX3_2.L.

Section 4.3 Special note on relabelling

Do not confuse relabelling with substitution. While the example provided above would tend to make you think that relabelling is a mere substitution of gate names (interaction points), this is strictly due to the nature of the example.

The following example is a notorious case illustrating this difference:

consider the following process definition and then let's try to instantiate it in various ways:

```
process P[left,middle,right]:noexit :=
          left ; middle ; stop
            |[middle]|
          middle ; right ; stop
endproc
```

the following instance of this process will produce a sequence mapping the left, middle, right expected sequence.

instance of P:

P[paris,berlin,moscow]

sequence:

paris —> berlin —> moscow

One should note that the results are identical to the ones that would have been achieved with substitution instead of relabelling.

The next instance of P depicts the fact that someone will start in paris, come back to paris, and then go to berlin. This is a very frequent example of finite state machine.

P[paris,paris,berlin]

using **relabelling** will produce the sequence:

paris —> paris —> berlin

however, using **substitution** this same instance of P will produce:

paris —> berlin

The difference is due to the fact that LOTOS will first generate the sequence left —> middle —> right and then re-label each action by the instantiated values paris, paris again and berlin.

substitution would behave as if we had rewritten the process P as follows:

substituted process P[paris,paris,berlin]:noexit:= (* wrong *)

paris; paris ; stop

|[paris]|

paris ; berlin

endproc

One can easily see that the action “Paris” can synchronize only **once**, producing the resulting behaviour:

paris ; stop

|[paris]|

berlin ; stop

that can only produce a “berlin” action leaving the remaining “paris” action pending.

The following example is in file “LEX3_4.L” and illustrates the above concepts.

```
specification relabel_ex
                          [paris,berlin,moskva]:noexit
     behaviour
      P_relabel[paris,berlin,moskva]
      []
      P_relabel[paris,paris,berlin]
      []
      P_substitute[paris,paris,berlin]
     where
       process P_relabel[left,middle,right]:noexit:=
          left ; middle ; stop
           |[middle]|
          middle ; right ; stop
       endproc
       process P_substitute[paris,paris,berlin]:noexit:=
          paris ; paris ; stop
          |[paris]|
          paris ; berlin ; stop
       endproc
endspec
```

The three different choices in the behaviour expression will produce the following tree, that shows three sub trees (most left starts).

```
paris       <-----   FIRST SUBTREE
|   berlin
|   |   moskva
paris       <-----   SECOND SUBTREE
|   paris
|   |   berlin
paris       <-----   THIRD SUBTREE
|   berlin
```

Section 4.4 ADVANCED EXAMPLES

In this section, we will present some examples that will mainly illustrate some stylistic issues among the LOTOS users community and some introduction on the usefullness of abstract data types which have been left out of this PC version of LOTOS.

4.4.1 Database representation in LOTOS

This example shows how to perform the traditional database operations of validating the sign-on of a user and performing insertions of new users and deletions of existing ones.

First of all, the actual database is represented as a set of user-ids. The representation of this set has to be formalized in order to enable the compiler to recognize that it is a set, but also to enable operations to be performed on this set. This will give us an oportunity to present some abstract data type concepts.

Abstract Data Types are composed of two parts:

— operator definitions.

— equations that give the computation rules for each operator.

There are two kinds of operators:

— constructors and selectors.

Constructors are used to build data. In our case, we want to build a collection of user-ids. First, one must start with an empty set represented by the terminal

symbol "**{}**". The empty set is the basic constructor also called the seed. A new set can be constructed by inserting a new user-id into this empty set such as:

Insert(bernard,{}) is a new set.

We can further insert new users by using the same constructor "Insert" that is defined generically as Insert(element,set) as follows:

Insert(mary,Insert(bernard,{})) and so on...

Destructors can be defined in a very similar way as Remove(element,set):

Remove(bernard,Insert(mary,Insert(bernard,{}))).

We now have to define selectors that will manipulate a set. We have built-in two such selectors, one to verify the existence of an element in a set and its opposite that verifies its non-existence. These are the constructors and selectors that we will use in our database example.

The syntax of these two selectors is:

IsIn(element,set) and **NotIn(element,set)**.

For example, one may query the database by stating:

IsIn(bernard,Remove(bernard,Insert(mary,Insert(bernard,{})))) ? and get as a reply False.

The reverse example should generate true:

NotIn(bernard,Remove(bernard,Insert(mary,Insert(bernard,{})))) ?

The equations that would produce such results are quite simple:

IsIn(X,{}) = False; states that X is never in an empty set.

IsIn(X,Insert(X,Y)) = True; states that an element X is in a set if that element has been inserted in that set.

Finally, the above rule may not succeed immediatly if the Insert constructor for that element is not the first element of the set. A set in this abstract data type system is comparable to a list where there is no direct access to elements. The following equation enables to examine each elements of the set by stating that if the element is not in the current Insert constructor, keep looking in the remaining set:

IsIn(X,Insert(Y,Z)) = IsIn(X,Z);

From a practical point of view, these equations have been built-in in this version of PCLOTOS. The syntax of the constructors is the only aspect to remember for their use in the database example.

Using sets to picture database operations

The database interface is specified as an instance of an interface process that has two interaction points, one with the users and one with the database itself. The database itself is represented as a formal parameter of the interface process. Operations to the database can be performed with one instance of the interface process. New operations require a new instance of the interface process. This can be implemented using the tail recursion to the interface process. The resulting modified database is then passed on to the formal parameter of the new instance of the interface process.

The interface process is composed of three nondeterministic choices corresponding to the three types of operations we are specifying: query an existing user, insert and delete users.

The first action "**dbase ! USERS**" of the query section consists in displaying the set of current users to help understand the mechanism involved. The query prompt is represented by the "**user ? QUERY_USER**" action. The next set of actions are guarded actions. Each guard actually performs the database query . The guard "**[QUERY_USER IsIn USERS] —>** " will allow the message "**user is authorized**" to be displayed by the database system. The other choice uses the **NotIn** selector to display the "**user not found**" message. In this case we continue the processing by asking the user if he wants to continue. In the affirmative, we recurse to another instance of the interface process.

Adding a new user consists merely in prompting the user for a new user-id and then recursing to another instance of the interface process by modifying its formal parameter by inserting the new user-id in the current set of user-ids.

Deleting an existing user-id is very similar to the above insertion behavior. The only difference is in the modification of the formal parameter of the interface process. The remove destructor is used to indicate that the user-id has been deleted.

```
specification database_interface
                              [ user,dbase ]:noexit
     /* example the database is represented
        by a set of users in the interface
        process formal parameters */
       /* example LEX4_1.L */
     behaviour
```

```
      interface[user,dbase]
                    (Insert(john,Insert(mary,{})))
   where
      process interface[user,dbase](USERS):noexit:=
        /* database query functions */
         dbase ! USERS ; user ? QUERY_USER ;
         (
          [QUERY_USER IsIn USERS]->
                     dbase ! "user is authorized" ;
          interface[user,dbase](USERS)
          []
          (
           [QUERY_USER NotIn USERS]->
                        dbase ! "user not found" ;
           user ! "continue ?" ? ANSWER ;
           (
            [ANSWER eq yes]->
                      interface[user,dbase](USERS)
             []
            [ANSWER eq no]-> stop
           )
          )
         )
         []
         /* add a new user to the data base  */
         user ! new_user ? NEW_USER ;
         interface[user,dbase]
                           (Insert(NEW_USER,USERS)
         []
         /* delete an existing user from the
                                   data base  */
         user ! delete_user ? DEL_USER;
         interface[user,dbase]
                           (Remove(DEL_USER,USERS)
     endproc
endspec
```

The database example execution tree.

This execution tree has been achieved by pruning the original infinite tree that would exist due to the recursion to the interface process. Pruning is achieved using the ESC key at the appropriate step.

```
                    insert(bernard,
          insert(john, insert(mary, {}))))
```

4.4.2 Finite state machine representation in LOTOS

Finite state machines are extensively used in protocol design and testing. They are very easily represented in LOTOS using a single recursive process that is composed of nondeterministic choices of state transitions that are guarded by state identification guards. States are passed on via a formal parameter of the recursive state machine process. Each transition leads into a recursion where the formal parameter is set to the new state.

For example, the transport service represented by a state machine will start with an instance of the transport process with the value **idle** in its formal parameter. This will allow the first choice of action **[STATE eq idle]-> t ! conreq** to be fired. It then recurses with another instance of process transport with the value **requested** in its formal parameter. This will in turn enable to fire the second choice action **[STATE eq requested]-> t ! tcon_ind** and so on.

Finite state machines are easy to understand but they have some annoying side effects. First it is difficult to depict concurrency, but also it can obscure the original intentions of the specifier. The following specification of a transport service using a state machine shows an awkward way to specify disconnection. The guard is long and does not enable the reader to relate this condition with their associated disabled behaviors.

```
specification state_transport [ t ]:noexit
     /* example LEYX4_2.L */
     behaviour
        transport[t](idle)
     where
       process transport[t](STATE):noexit:=
           [STATE eq idle]-> t ! conreq ;
                            transport[t](requested)
            []
           [STATE eq requested]-> t ! tcon_ind ;
                            transport[t](indicated)
            []
```

```
          [STATE eq indicated]-> t ! tcon_resp ;
                                 transport[t](responded)
           []
          [STATE eq responded]-> t ! tconconf ;
                                 data_transfer[t]
           []
          [STATE eq requested  or
           STATE eq indicated or
           STATE eq responded]->
                   (
                     t ! discon_req ; stop
                      []
                     t ! discon_ind ; stop
                   )
       endproc
       process data_transfer[t]:noexit:=
         t ! data_req ; data_transfer[t]
         []
         t ! data_ind ; data_transfer[t]
      endproc
endspec
```

Execution tree of the state oriented transport service

```
execution tree:  state_transport
t   !   conreq
|    t   !   tcon_ind
|    |    t   !   tcon_resp
|    |    |    t   !   tconconf
|    |    |    |    t   !   data_req
|    |    |    |    |    t   !   data_req
|    |    |    |    |    t   !   data_ind
|    |    |    |    |    |    t   !   data_req
|    |    |    |    |    |    t   !   data_ind
|    |    |    |    t   !   data_ind
```

4.4.3 The constraint oriented specification style

One may specify protocols by just using the sequence and the choice operators. This would in essence represent a symbolic tree of possible sequences of actions. This however would first generate a useless redondancy of action sequences but further more totally obscure the underlying structures of the specification. The constraint oriented style is the most natural way to specify concurrency without redondancy and as a benefit allow clarity in the specification.

The transport service is used as an example to illustrate the constraint oriented style. It is very similar in structure to the telephone example presented earlier. It is composed of two basic processes: the station process that represents the behavior of a communicating entity and the network process that represents the end-to-end behavior of a connection. The station process has been further decomposed to represent the two possible behaviors or roles of connection initiator or responder.

There are two interaction points:

t: is the interaction point with the upper service layer.

n: is the interaction point with the network.

The data transfer and termination process are similar to the previous monolithic transport service example.

```
specification constraint_transport [t]:noexit
   /* example LEX4_3.L */
   behaviour
      (
       station[t](address1)
         |||
       station[t](address2)
      )
       ||
       network[t]
    where
      process station[t](ADDRESS):noexit:=
         initiator_role[t](ADDRESS)
         |||
         responder_role[t](ADDRESS)
      endproc
```

```
      process network[t]:noexit:=
         t ? INIT_ADDRESS ! conreq ? RESP_ADDRESS ;
         t ! RESP_ADDRESS ! tcon_ind ;
         t ! RESP_ADDRESS ! tcon_resp ;
         t ! INIT_ADDRESS ! tconconf ;
         stop
      endproc
      process initiator_role[t](ADDRESS):exit:=
         t ! ADDRESS ! conreq ? CALLED_ADDRESS ;
                     (
                       t ! ADDRESS ! tconconf ; exit
                       []
                       termination_phase[t](ADDRESS)
                     )      endproc
      process responder_role[t](ADDRESS):exit:=
         t ! ADDRESS ! tcon_ind ;
                     (
                       t ! ADDRESS ! tcon_resp ; exit
                       []
                       termination_phase[t](ADDRESS)
                     )
      endproc
      process data_transfer[t](ADDRESS):noexit:=
         t ! ADDRESS ! data_req ;
         data_transfer[t](ADDRESS)
         []
         t ! ADDRESS ! data_ind ;
         data_transfer[t](ADDRESS)
      endproc
      process termination_phase[t](ADDRESS):exit:=
         t ! ADDRESS ! discon_req ; stop
         []
         t ! ADDRESS ! discon_ind ; stop
      endproc
endspec
```

4.4.4 Busy location representation in LOTOS

There are different ways to verify if a communicating entity is busy. The first one consists in using some database where the state of the communicating entity is stored. One needs to query that database to obtain its state and using a guard proceed to the appropriate behavior branch.

Another way uses the synchronization properties of LOTOS. The communication with another entity is accomplished via synchronization on common actions. When a communicating entity is busy, this means that it usually cannot synchronize on an initial action of a communication sequence. This busy entity can however indicate to any new potential communication requesting party that it is busy. This can easely be represented with the interleave operator "|||" in the following structure:

```
trigger_action ;
            ( normal_sequence ||| busy_sequence )
```

The following specification is similar to the previous specification of the constraint oriented style of the transport service but it is augmented with the busy sequences.

```
specification busy_transport [t]:noexit
  /* example LEX4_4.L */
   behaviour
      (
       station[t](address1)
         |||
       station[t](address2)
      )
       ||
       network[t]
    where
      process station[t](ADDRESS):noexit:=
         initiator_role[t](ADDRESS)
         |||
         responder_role[t](ADDRESS)
      endproc
      process network[t]:noexit:=
```

```
   t ? INIT_ADDRESS ! conreq ? RESP_ADDRESS ;
   t ! RESP_ADDRESS ! tcon_ind ;
   (
    t ! RESP_ADDRESS ! tcon_resp ;
    t ! INIT_ADDRESS ! tconconf ; stop
    []
    t ! RESP_ADDRESS ! busy ; stop
   )
endproc
process initiator_role[t](ADDRESS):exit:=
  t ! ADDRESS ! conreq ? CALLED_ADDRESS;
          (
              (
                t ! ADDRESS ! tconconf ; exit
                []
                termination_phase[t](ADDRESS)
              )
            |||
              t ! ADDRESS ! busy ; stop
          )
endproc
process responder_role[t](ADDRESS):exit:=
   t ! ADDRESS ! tcon_ind ;
          (
              (
                t ! ADDRESS ! tcon_resp ; exit
                []
                termination_phase[t](ADDRESS)
              )
             |||
              t ! ADDRESS ! busy ; stop
          )
endproc
process data_transfer[t](ADDRESS):noexit:=
   t ! ADDRESS ! data_req ;
   data_transfer[t](ADDRESS)
```

```
          []
          t ! ADDRESS ! data_ind ;
          data_transfer[t](ADDRESS)
       endproc
       process termination_phase[t](ADDRESS):exit:=
          t ! ADDRESS ! discon_req ; stop
          []
          t ! ADDRESS ! discon_ind ; stop
       endproc
endspec
```

Chapter 5 SPREADING YOUR WINGS

Writing your own LOTOS specification

Section 5.1 PC LOTOS syntax

PC LOTOS is a **subset** of the ISO international standard. It It is different in two ways:

- there are no Abstract Data Types (**ADTs**). Data is considered as a character string.

- there are a few operators missing. (***hide, distributed choice operators, accept in, and the functionality of the exit operator is zero only***).

A LOTOS specification has the following main skeleton:

specification <specification name> [<interaction points list>] (<formal parameters list>) : < functionality>

behaviour

< behaviour expression >

where ***(* if there are process declarations *)***

< process declaration >

endspec

A process declaration has the form:

process <process name> [interaction point list] (formal parameters list) : <functionality> :=

<behaviour expression>

endproc

a behaviour expression is:

operators acting on action denotations and process instantiations.

This is the syntax for each operator:

using the following symbols:

a : action denotation

B : behaviour expression

E : Boolean expression

operator syntax:

a ; B : action denotation

B1 [] B2 : non-deterministic choice operator

B1 ||| B2 : parallel interleave operator

B1 || B2 : strict synchronization parallel operator

B1 |[g1,...,g2]| B2 : mixed synchronization and interleave parallel operator

B1 [> B2 : disable operator

B1 >> B2 : enable operator

[E] -> B : guard operator

i : the internal event (does not have to synchronize)

stop : inaction (can be only found after an action prefix operator and as the last action of a sequence.

exit : is a behaviour expression and means successful termination

process instantiation:

<process name> [actual interaction points list](actual formal parameters values)

example: if process P1 was declared as

process**P1[g1,g2](X,Y):exit:=**

...

endproc

a process instantiation would look like:

P1[network,user](2345,"active")

action denotation:

gate <offer-list> <predicates>

an offer-list is none or many offers.

an offer is:

! <value or variable>

? <variable identifier>

example: network ! 1234 ? Called_number

Section 5.2 Debugging your specification

5.2.1 most common bugs

- undeclared gates (interaction points) and relabelling surprises

gates used in a behaviour expression must belong to the interaction point list declared in its corresponding process or specification declaration.

example:

specification test1 [transport,network] :noexit

behaviour

P1 [transport,network]

where

process P1 [g1,g2]:noexit:=

P2 [g1] [] P3[g2]

endproc

process P2[a]:noexit:=

a ! tconreq ; stop

endproc

process P3[a]:noexit:=

a ! nconind ; stop

endproc

endspec

This specification will not compile and the error message : undeclared gate "usr" will be displayed because the instantiation of P1 in the main behaviour is wrong. Only the gate "user" should have been used.

This is the only bug this specification will show. However further execution of this specification will acquaint you with the art of relabelling. P2 and P3 are both using an interaction point "a". However, after execution these two interaction points will be re-labelled differently.

the interaction point **“a”** of process P2 will be re-labelled "transport" while the interaction point **“a”** of process P3 will be re-labelled "network".

This is of course an extreme case that we do not advise you to use.

- wrong number of gates or formal parameters in process instantiation.

if you declare a process P[a,b,c](X,Y,Z) and you instantiate it somewhere in your specification as P[f,g]("aaa","bbb") you will get a "wrong number of gates" and "wrong number of formal parameters" message.

- unbalanced parentheses.

The same rules as in traditional programming languages apply.

- undeclared process message

As the result of a spelling mistake in a process name, or you forgot to define a process. If you wish to declare your process at a later time while testing your specification, you can do so as long as you specify at least a "stop" or an "exit" behaviour.

- data is string in PCLOTOS

if you are using non atomic data such as octet representation similar to the standard library, enclose your data in double quotes.

g ! "Octet(1,1,1,1,0,0,0,0)" ; stop

or in a more compact version: g ! 11110000 ; stop which is allowed in PC LOTOS only and not in IS LOTOS.

- free variable representation difference with International Standard LOTOS (IS):

Since there are no abstract data types in PC LOTOS, there is no type declaration in action denotations when a "?" is present, or in formal parameters in process declarations.

For those unaware of the official IS LOTOS syntax, there is nothing to worry about.

consequently PC LOTOS correct action denotations look like:

g ! 1234 ? X ! ack ? Y ; ...

and process declarations:

process P1[g,h,e](A,B,C):noexit:=

.....

endproc

WARNING: This of course brings up an interesting point: how do we differentiate values from free variables in PC LOTOS?

In full IS LOTOS, this is resolved using abstract data type declarations. A value can be used only if declared. The compiler will produce an error message

in case a value is not declared. As a matter of fact the compiler will produce a somewhat puzzling error message, because it will think that the value you have entered is a free variable. Free variables can be used as long they are declared upstream in a behaviour expression.

example:

g ? X ; g ! X ; stop

is syntactically **correct** because "X" has been declared before it has been used. This is why in LOTOS one refers to **value declaration** for the query symbol **"?"**.

g ! Y ; g ! "aaa" ; stop

is syntactically **incorrect** because the free variable "Y" has never been declared. However PC LOTOS will think that the “Y” variable is the value “Y” (string value), because there is no ADT definition to find out its real nature.

The above two examples are PC LOTOS syntax examples. The correct IS LOTOS syntax would have been:

g ? X:type_t ; g ! X ; stop

and no difference for the second example except for an error message:

g ! Y ; g ! "aaa" ; stop

In PC LOTOS there is no logical mechanism to differentiate a value from a free variable. We however advise the reader to comply with a self-disciplined rule of using upper case letters for variables and lower case letters for values. It will make the specification easier to read as well. It will not however resolve the syntax error of an undeclared free variable, and your upper case identifier will still be considered as a plain data value. It will however help you debug your specification. This of course shows the usefulness of Abstract Data types, which will be the subject of an upcoming tutorial.

- local definitions are not permitted in PCLOTOS.

there can be only one **"where"** keyword immediately following the main behaviour expression. This is a significant difference from IS LOTOS. This means that **no "where" keywords should exist within a process declaration.** In PCLOTOS we have deliberately chosen the rule to declare each process independently from each other. This is a choice derived from experience with working with large specifications.

Chapter 6 Moving to International Standard LOTOS

The main goal of this tutorial was to allow the reader to become familiar with LOTOS basic operators and concepts with hands-on experience. A second goal was to provide the full LOTOS expert with an "out of the lab" tool to be used while travelling or for demonstration purpose at sites not equipped with the necessary environment for IS LOTOS.

Section 6.1 Available LOTOS tools

Serious LOTOS specifications can only be achieved with full IS LOTOS. There are a number of tools available from various research centres. Among them are:

ISLA : University of Ottawa, Canada in both TTY and Xwindows versions including Graphic Lotos.

HIPPO: Twente University, based on the CORNELL synthesizer

LOLA: University of Madrid.

most of these tools work on VAX and/or SUN workstations.

6.1.1 The University of Ottawa LOTOS Toolkit

The University of Ottawa has developed a toolkit which takes a LOTOS specification as input, checks its static semantics, and executes the specification to produce the next possible actions as output. Full standard LOTOS is supported, and specifications of thousands of lines have been executed. The toolkit can be used in either of two modes: (1) step-by-step mode (interpretation) and (2) symbolic tree generation mode. In mode (1) the user is prompted, at each step, with the next possible actions. The user chooses the next action to be executed and, if the action involves a choice of data, the user is required to provide it. Used in this way, the specification may be considered as a running prototype of the entity specified. In mode (2) the user may generate a tree of all possible actions from any point. However, variable values are represented by expressions that may involve other variables.

A brief description of some of the features available to the user follows:

1. Alternative execution paths (Back): This command allows the user to go back (up the tree) and select another branch for execution. The user may select to go up one level only or many levels, up to the root of the tree.

2. Setting checkpoints: The set of execution paths of a specification is represented as a tree. For medium and large specifications, it is a very time-consuming task to travel, in the step-by-step mode, the various branches of the tree. Several sessions are usually necessary. It is also impractical to start the execution from the beginning for every testing session. The interpreter provides a way around this problem. The user travels the tree, say from the root R, up to some node, say N, and then invokes the checkpoint setting command to save the current behaviour expression. In the next testing session, testing continues from the behaviour expression saved for node N. Several checkpoints can be saved.

3. Composing behaviour expressions: This command allows the user to compose behaviour expressions. For example, a tester behaviour expression can be composed with another behaviour expression under test.

4. Value expression evaluation: This command allows the user to evaluate any given value expression, without interfering with the execution of the current behaviour expression.

5. Complex predicates: In the case of complex predicates, sets of tuples of value expressions can be submitted for evaluation. The interpreter will determine which tuples are true for the given predicates.

6. Shorthands for constants: shorthands for long and commonly used value expressions can be defined.

Hardware and Software Requirements

The current toolkit, which requires only a tty terminal, runs on UNIX 4.0 and later versions under SUN 3 and SUN 4. Normally, the object code only is released.

You can acquire the toolkit by the following means:

o SCSI Cassette tapes;

o 1/2 in. Magnetic tapes (density 6250)

o Exabyte tapes (8 mm.)

o Electronic mail.

Upgrades and Improvements

The development team is currently working on a graphical version of the toolkit which runs in the XWindow system environment. A DEC station version is also being planned. No date for release has been set yet.

Technical Assistance

If you require technical information on the system, please contact:

Dr. Luigi Logrippo
University of Ottawa,
Protocols Research Group,
Department of Computer Science
Ottawa, Ontario, Canada K1N 6N5
E-Mail: lmlsl@uottawa.bitnet
Tel.: (613) 564 - 5450 Fax: (613) 564 - 9486

Section 6.2 upload / download between PC LOTOS and IS LOTOS.

You certainly can do that, but you have to remember that PCLOTOS is a subset of IS LOTOS. Consequently you can expect numerous syntax errors both ways. As a reminder, remove any type reference when going from IS LOTOS to PCLOTOS

Bibliography

Index