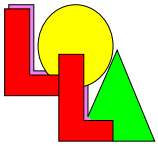
DEPARTAMENTO DE INGENIERIA TELEMATICA
Universidad Politécnica de Madrid
E.T.S.I. de Telecomunicación



LOtos LAboratory

User Manual (version 3R6)

Code: LOLA/N5/V10
Date: February 6, 1995
Authors: S. Pavón
D. Larrabeiti
G. Rabay
State: Final

Address: E.T.S.I. Telecomunicación, Ciudad Universitaria s/n, 28040 Madrid
Telephone: 91 5495700, 91 5495762. Télex: 47430 LCT E. Fax: 91 5432077.
e-mail: spavon@dit.upm.es dlarra@dit.upm.es

Contents

1 LOLA: LOtos LAboratory.

1.1 An Overview

LOLA (LOtos LAboratory) [QFM87, QPF89a, PL91] is a transformational and state exploration tool. It supports the LOTOS based phases of the design cycle, by transformation, execution and testing of LOTOS specifications.

LOTOS (Language of Temporal Ordering Specification [ISO89]) is a formal abstract description language which allows to describe a system in a precise way, abstracting away realization details. This is specially useful in the design and analysis of protocols and distributed systems, where the interactions system-environment have often complex interdependencies. LOTOS is based on several mathematical models, which makes possible to check the validity of the specification, ensuring the correctness of the design. **LOLA** provides the user with a set of tools that help to analyse the behaviour of a system before entering the realization phase.

The functionalities of **LOLA** are classified in four groups:

Simulation/Debugging: These operations allow to simulate LOTOS behaviours interactively step by step, or to evaluate data value expressions.

- **Step:** simulates a behaviour step by step.
- **Rewrite:** evaluates data value expressions.

Testing: These calculate the response of a system specification to a test (*must*, *may* or *reject*) according to the *Testing Equivalence*.

- **TestExpand:** Passes a test to the specification.
- **OneExpand:** Analyses a single random execution of the specification.

Expansion: The expansion transformations compute symbolically all the possible executions of a LOTOS specification, i.e. they obtain its labelled transition system. There exist several types of expansions.

- **Expand:** computes the EFSM (Extended Finite State Machine) of a behaviour.
- **VarExpand:** computes the parameterized EFSM of a behaviour.
- **FreeExpand:** computes the behaviour tree (without detecting duplicate states).
- **InterExpand:** compute the interleaved expansion of a behaviour.

Miscellaneous operations: LOLA also has a set of operations to load and print the specification, to navigate throughout the behaviours, etc.

- **Help:** help about commands.
- **Load:** restore the original specification.
- **Print:** print the current specification.
- **Move:** move **LOLA's** internal cursor.
- **DataTable:** show internal identifiers tables.
- **Statistic:** display memory and CPU time usage.
- **Set:** display/change LOLA configuration.
- **Command:** execute a LOLA command file.
- **Quit:** exit **LOLA**.

1.2 Language

LOLA accepts **full LOTOS** language (IS-8807).

The internal language used in **LOLA** is flattened LOTOS. This means in outline that the original specification is automatically transformed into an equivalent specification where the overloading of identifiers is resolved and the nesting of processes and types definitions is flattened.

LOTOS data types are treated operationally by interpreting equations as rewrite rules from left to right. These rewrite rules should be confluent and terminating to achieve proper operation. Thus some type of Knuth-Bendix completion algorithm is necessary to make the data type definition operational before working with **LOLA**. Data values or expressions containing variables are always treated through their canonical forms during the transformations.

1.3 Using LOLA

To start up the **LOLA** environment with a specification, just enter:

```
topo <spec> -lola [-l <lib>]
```

where `<spec>` is the name of a LOTOS specification and <lib> is the name of a LOTOS types library file.

There exists an X-window interface for **LOLA**, which can be invoked with option `-xlola` instead of `-lola`. This is specially suitable for interactive simulation - eg. selecting transitions and moving around is easily done by clicking the mouse, etc -, but we strongly recommend the textual version for batch test execution and expansion.

After invoking **LOLA**, the syntax and semantics analysis are done (any errors in the specification or in the library are reported), and the prompt **lola**> appears on the screen waiting for the user to input commands.

LOLA commands are referenced by their names (or their abbreviations) and delimited by a carriage return. The following keys provide command line edition capabilities [1] :

Key	Action
Ctrl-A	move to beginning of line
Ctrl-E	move to end of line
Ctrl-F	move forward one character
Ctrl-B	move back one character
Ctrl-D	delete current character or EOF
DEL	delete current character
Ctrl-H	delete left character
Ctrl-K	delete to end of line
Ctrl-P	previous history command
Ctrl-N	next history command
Ctrl-Q	first history command
Ctrl-W	last history command
Ctrl-L	redraw current line
Ctrl-R	redraw current line

LOLA has an internal cursor which always points to a sub-behaviour of the LOTOS specifica-

[1] Based on a line editing input package of Chris Thewalt. Copyright (C) 1991.

tions. Commands are applied to this active sub-behaviour. To see the current sub-behaviour just `print` it.

1.4 Example

The following specification will be used to describe the commands of **LOLA** throughout this document. This specification contains two processes, *Client* and *Bank*. The *Client* can borrow money from the *Bank* with a credit of two units.

```
SPECIFICATION Credit  [bank, work, sleep ] : NOEXIT
   TYPE Boolean IS
      SORTS bool
      OPNS  true,false :      -> bool
            not        : bool -> bool
      EQNS  FORALL x,y : bool
            OFSORT bool
              not(not(x))=x; not(true)=false; not(false)=true;
   ENDTYPE
   TYPE Money IS Boolean
      SORTS money
      OPNS  0        :              -> money
            inc, dec : money        -> money
            _eq_     : money, money -> bool
            noi, nod : money        -> bool
      EQNS  FORALL x,y : money
            OFSORT money
              inc(dec(x))=x; dec(inc(x))=x;
            OFSORT bool
              nod(0) = true; nod(inc(x)) = nod(x);
              noi(0) = true; noi(dec(x)) = noi(x);
              0 eq 0 = true;
              inc(x) eq y = x eq dec(y);     dec(x) eq y = x eq inc(y);
              nod(x) => 0 eq inc(x) = false; noi(x) => 0 eq dec(x) = false;
   ENDTYPE
   TYPE BankOperation IS
      SORTS BANKOP
      OPNS  borrow, pay : -> BankOp
   ENDTYPE
BEHAVIOUR
 Bank[bank,sleep](inc(inc(0)),0) |[bank]| Client[bank,work]
WHERE
      PROCESS Client [ bank, work ] : NOEXIT :=
          bank !borrow; Client[bank,work]
       [] bank !pay   ; Client[bank,work]
       [] work        ; Client[bank,work]
      ENDPROC
      PROCESS Bank [ bank, sleep ] (Max,debt:Money) : NOEXIT :=
          bank !borrow [not(debt eq Max)]; Bank[bank,sleep](Max,inc(debt))
       [] bank !pay    [not(debt eq 0)]  ; Bank[bank,sleep](Max,dec(debt))
       [] sleep                          ; Bank[bank,sleep](Max,debt)
      ENDPROC
ENDSPEC
```

2 Miscellaneous Operations

The main applications of **LOLA** are *Simulation*, *Testing* and *Expansion*. There is also a set of miscellaneous commands such as `help`, `print`, etc. which are described in this section.

2.1 HELP

This command either displays the list of available commands or describes the funcionality of a specified command and its options.

Syntax:

```
Help [<command_name>]
```

<**command_name**> is the name of a **LOLA** command. When this argument is present a detailed description of the command is given; otherwise, the list of available commands is displayed.

2.2 LOAD

This command restores the LOTOS specification.

Syntax:

```
Load
```

Since most **LOLA** commands modify the LOTOS specifications, it is sometimes necessary to start again from the initial situation (or to update the changes made with an auxiliary editor) without quitting, i.e. **Load** reads the specification file.

2.3 PRINT

This command prints the LOTOS behaviour pointed by the internal cursor, up to a specified *depth*, into a file or into the standard output.

Syntax:

```
Print [-p] [-t] [-a] [-c] [<depth>] [<output_file>]
```

-p	print definitions of the processes instantiated until `<depth>`.
-t	print data type definitions.
-a	print the whole specification, with processes and data type definitions.
<**depth**>	printing depth (number of actions visible or invisible printed). A negative *depth* means no boundary.
<**output_file**>	name of the output file. Default is standard output.

For example, the command `print -p -1` prints completely the current behaviour and the processes used within into the standard output. Options <depth>, -p or -t can be set as default and omitted (see command **Set**). Initial default value for <depth> is 4.

2.4 MOVE

This command moves the internal cursor to another place in the specificacion.

Syntax:

```
Move [<position>] [<position>] ...
```

`<Position>` can be:

Position	Movement to
<*number*>	line <*number*> of the current behaviour.
^	the root of the specification.
<*process_name*>	the process definition called <*process_name*>.
<*number*>d	**d**own <*number*> LOTOS operators.
<*number*>u	**u**p <*number*> LOTOS operators.
<*number*>b	<*number*>-th operand (**b**ranch) of the current operator.

In the **XLOLA** moving is as easy as clicking, but in the textual interface is a bit different. There are two ways of moving. Let us illustrate the easiest one with an example. Imagine we want to move the cursor to the last line of process `Client`, which is a self-instantiation. The first thing we have to do is telling **LOLA** that we want to move :

```
lola> move

   1    specification credit [bank,work,sleep] : noexit
   2
   3    behaviour
   4
   5         bank [bank,sleep] (inc(inc(0)),0)
   6       |[bank]|
   7         client [bank,work]
   8
   9    where
   10      process bank [bank,sleep] (max_6:money, debt_7:money) : noexit
   11      process client [bank,work] : noexit
   12
   13   endspec
```

LOLA expects you to move to one of the listed lines. Since we want to move to the process definition `client`, which is at line 11 we type:

```
lola> move 11
```

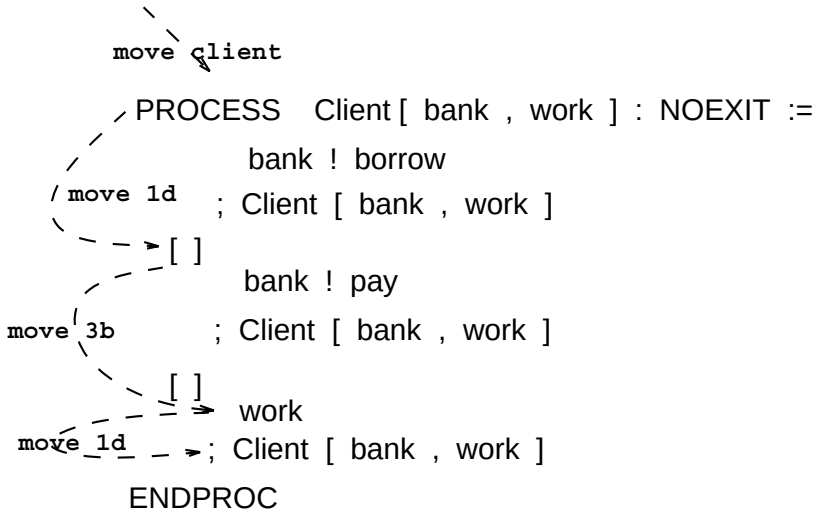
Now we are placed in the process definition. We could see it by printing it or, as we have not reached our destination yet, by *moving* again :

```
lola> move

   1    process client [bank,work] : noexit :=
   2         bank  ! borrow;
   3         client [bank,work]
   4       []
   5         bank  ! pay;
   6         client [bank,work]
   7       []
   8         work;
   9         client [bank,work]
   10   endproc

lola> move 9
```

and we are done, ready to execute any command on this behaviour.

A second way to navigate is using tree-oriented movements. This way is faster but more difficult, since the user needs to know about the syntactical structure of his/her specification. This is a batch-oriented moving mechanism.

For example, to move the internal cursor to the last line of the process definition *Client* in a tree-oriented way, you type in `move client 1d 3b 1d`. This causes the cursor to be positioned in the process definition *Client* and then moved down onto the first operator of this process. The third alternative of the *choice* operator is selected, and finally the cursor is placed below the *action prefix*, on the process instantation *Client*.



2.5 DATATABLE

This command shows the internal tables used in **LOLA**. Useful only for advanced LOLA users willing to have a statistical view of the specification (e.g. to know the number of gates, processes, operations, sorts or variables used) .

Syntax:

```
DataTable -s|v|o|p|g [<low_limit>] [<high_limit>]
```

<**low_limit**> and <**high_limit**> are numbers which indicate the first and last table positions

to be printed. Options **-s**, **-v**, **-o**, **-p** and **-g** select the sort, variable, operation, process or gate table, respectively. If the limits are not specified the entire table is displayed.

For example, to see positions 6 to 10 of the operation table, you should execute the following command:

```
lola> datatable -o 6 10

      TABLE OF OPERATIONS:

      6  = 0 : -> money (2)
      7  = inc : money (2) -> money (2)
           inc(dec(x_2)) = x_2 ;
      8  = dec : money (2) -> money (2)
           dec(inc(x_2)) = x_2 ;
      9  = _eq_ : money (2), money (2) -> bool (3)
           noi(x_2) = true => 0 eq dec(x_2) = false ;
           nod(x_2) = true => 0 eq inc(x_2) = false ;
           0 eq 0 = true ;
           inc(x_2) eq y_3 = x_2 eq dec(y_3) ;
           dec(x_2) eq y_3 = x_2 eq inc(y_3) ;
      10 = noi : money (2) -> bool (3)
           noi(0) = true ;
           noi(dec(x_2)) = noi(x_2) ;
```

2.6 STATISTICS

This command reports memory and CPU time used by **LOLA**.

Syntax:

```
Statistics
```

Example :

```
lola> stat
statistics

   Memory and time usage:  616 Kbytes. 0.14 sec.
```

The availability of this command depends on the system on which **LOLA** runs . The value depends perceptibly on system load for multitask systems.

2.7 SET

Set assigns a default value for command options and pre-expansion options. These default options are stored in internal variables.

Syntax:

```
Set [<variable> [<value>]]
```

With no argument, **Set** displays the values of all these variables. With the <**variable**> argument alone, Set resets <**variable**> to the initial default value.

The following table shows the meaning of each variable and its initial default values:

Variable	Range	Initial Value	Effect
divergence_check	ON | OFF	OFF	divergence analysis before first expansion
unguarded_proc_check	ON | OFF	ON	apply unguarded process detection before first expansion
print_depth	<integer>	4	default print command depth
print_types	ON | OFF	OFF	print types
print_proc	ON | OFF	OFF	print process definitions of printed instantiations
verbose	ON | OFF	OFF	verbose mode
expand_i_removal	ON | OFF	OFF	weak bisimulation congruence in expansion
exploration_depth	<integer>	-1	exploration depth in testing & expansion commands
success_event	<string>		success event in testing commands
test_i_removal	ON | OFF	ON	test equivalent reduction in testexpand
quit_exploration_on_may_response	ON | OFF	ON	abort testing as soon as the test result is MAY without further exploration
expected_response	MUST|MAY|REJECT		abort testing when the test result does not match this value
quit_lola_on_unexpected_response	ON | OFF	OFF	abort testing and exit LOLA with value 1 when the test result does not match the expected response
memory_size	<natural>		size of buffer in Mbytes devoted to heuristic exploration. Turn on partial test expansion.

2.8 COMMAND

Command executes a **LOLA** command file.

Syntax:

```
Command <command_file>
```

A **LOLA** command file is a text file that contains a succession of commands separated by $< cr >$. **Command** impels **LOLA** to take this file as input of commands and execute the commands just as if they were typed in from the command line.

An example of command file :

```
set divergence_check ON
set print_depth      -1
set verbose          ON
set success_event    test_ok
testexpand test1
testexpand test2
testexpand test3
```

2.9 QUIT

This command quits **LOLA**.

Syntax:

```
Quit
```

3 Simulation/Debugging

These operations are used to debug and to simulate LOTOS behaviours and data type definitions. The user can execute interactively any specification step by step assigning values to the variables defined in the specification. Likewise, the canonical form of any data value expression can be evaluated according to the rewrite rules defined in the specification.

3.1 REWRITE

`Rewrite` evaluates data value expressions, i.e. it calculates the normal form of an expression.

Syntax:

```
Rewrite <expression>
```

<`Expression`> is the data value expression to be evaluated according to the rewrite rules specified in the LOTOS specification. The rewrite rules can be printed using both the `print` or the `DataTable` commands.

For example, to evaluate the expression $inc(inc(dec(0)))\ \ eq\ \ dec(0)$:

```
lola> rewrite inc(inc(dec(0))) eq dec(0)

      false
```

3.2 STEP

Step executes the current behaviour (or the composition *current behaviour* - *test*, if a test process is provided) interactively step by step.

All the transitions offered in the current state are presented in a menu from which the user has to choose one to be executed. After the user has selected (i.e. executed) one transition, a new menu is displayed , showing the transitions offered at the new state.

The user can assign expressions to the variable parameters of the process definitions, to the variables in sum-expressions and to the variables defined in gates or *exit* statements offers. Whenever a variable value gets undefined during the simulation, the user is requested to enter a value expression to be assigned to that variable. If no expression is provided then the variable will remain unassigned and treated symbolically in the next states.

Syntax:

```
Step [<success_event> <test_process>]
```

If the parameters <`success_event`> and <`test_proc`> are specified, they are taken to rebuild the parallel composition that is created automatically with the commands `TestExpand` and `OneExpand`. The simulation is performed over this composition. See section 5 for more information.

When the command `Step` is invoked, you enter the *Step* environment and a new prompt appears below the menu of transitions :

```
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?>
```

These are the commands available in *Step mode* :

Print	print the behaviour of the current state. Syntax: `Print [-p] [-t] [-a] [<depth>] [<output_file>]`
Menu	display the menu of transitions offered at the current state.
Refused	display the menu of unsuccessful synchronizations (due to data value offering mismatch) at the current state. The numbers printed in parenthesis are the line numbers of the events involved in the unsuccessful synchronization. Note that the menu labels are negative numbers.
Sync <**n**> [<**proc**>]	- for a transition <n> : show the events that produced it. - for an unsuccessful synchronization <n> : show the events that could not synchronize. Each event is diplayed below the stack of processes instantiated to produce it. If the name <proc> is specified then only the instantiations of that process are displayed.
<**n**>	execute the transition labelled <**n**> from the menu of transitions.
Undo	undo the last simulation step (back to previous state).
Trace	display the sequence of transitions that lead to the current state.
Exit	quit simulation mode.
?	help.

`Step` can be applied to any sub-behaviour of a specification by placing the internal cursor on it previously(see `Move` command). Let us see an example of step-by-step simulation with the specification *Credit* . The user inputs follow the *Step mode* prompt.

```
lola> step
              [  1]  bank  ! borrow;
              [  2]  sleep;
              [  3]  work;

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> 2
                 ==> sleep;
-------------------------------------------------------------------------
              [  1]  bank  ! borrow;
              [  2]  sleep;
              [  3]  work;

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> 1
                 ==> bank  ! borrow;
-------------------------------------------------------------------------
              [  1]  bank  ! borrow;
              [  2]  bank  ! pay;
              [  3]  sleep;
              [  4]  work;

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> 3
                 ==> sleep;
-------------------------------------------------------------------------
              [  1]  bank  ! borrow;
              [  2]  bank  ! pay;
              [  3]  sleep;
              [  4]  work;

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> 1
                 ==> bank  ! borrow;
```

```
------------------------------------------------------------------------
              [  1]  bank  ! pay;
              [  2]  sleep;
              [  3]  work;

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> undo
              Last simulation step undone.
```

We have executed four actions so far, and have undone one, so here is the current event history and the actions offered.

```
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> trace
              [  2] - sleep;
              [  1] - bank  ! borrow;
              [  3] - sleep;

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> menu
              [  1]  bank  ! borrow;
              [  2]  bank  ! pay;
              [  3]  sleep;
              [  4]  work;
```

Now we might be wondering why `bank ! pay` is being offered (obviously we have borrowed money once, so we have to return it). There are two ways to analyse this : 1) `Sync` traces the history of process instantiations that have occurred in each part of the parallel to produce `bank ! pay`; 2) `Print` just displays the current behaviour.

```
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> sync 2
              bank [bank,sleep] (inc(inc(0)),0) (* line 32 *)
              bank [bank,sleep] (inc(inc(0)),0) (* line 44 *)
              bank [bank,sleep] (inc(inc(0)),inc(0)) (* line 42 *)
              bank [bank,sleep] (inc(inc(0)),inc(0)) (* line 44 *)
               bank  ! pay (* line 43 *);
             |[bank]|
              client [bank,work]  (* line 34 *)
              client [bank,work]  (* line 37 *)
               bank  ! pay (* line 38 *);

<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> print
                (Let max_6:money=inc(inc(0)),debt_7:money=inc(0) in
                 bank [bank,sleep] (max_6, debt_7)
                )
              |[bank]|
                  bank  ! borrow; ......
                [] bank  ! pay; ......
                [] work; ......
```

We may also wish to know what is failing to synchronize due to offers mismatch. This is the menu of refused synchronizations :

```
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> refused
              [  -1]  bank (at lines 42,38)
              [  -2]  bank (at lines 43,37
```

Again we can analyse any of these synchronizations with `Sync`:

```
<n>,Undo,Menu,Refused,Sync,Print,Trace,Exit,?> sync -2
              bank [bank,sleep] (inc(inc(0)),0) (* line 32 *)
              bank [bank,sleep] (inc(inc(0)),0) (* line 44 *)
              bank [bank,sleep] (inc(inc(0)),inc(0)) (* line 42 *)
              bank [bank,sleep] (inc(inc(0)),inc(0)) (* line 44 *)
               bank  ! pay (* line 43 *);
            |[bank]|
              client [bank,work]  (* line 34 *)
              client [bank,work]  (* line 37 *)
               bank  ! borrow (* line 37 *);
```

and we check that everything has evolved as expected and `bank !pay` and `bank !borrow` must be offered in each process but cannot synchronize.

Warning : Line numbers displayed with `sync` may be lost if any expansion or testing operation is performed before invoking `step`.

4 Expansion

The expansion transformations produce a compressed version of the transition system generated by a LOTOS specification, that represents its EFSM (Extended Finite State Machine), as an equivalent LOTOS behaviour. These transformations apply a generalized version of the so-called *Expansion Theorem* (see IS8807). The effect of an expansion is the removal of the most complex LOTOS operators (parallel, disabling, enabling,...) from the specification, producing an equivalent specification in terms of action prefix, behaviour choice, guards and data value choice. This transformation can be used for state exploration, deadlock detection, deriving efficient implementations, input graph for model checking, etc.

The expansion commands are `Expand`, `VarExpand`, `FreeExpand` and `InterExpand`. All of them can be applied to any sub-behaviour of the LOTOS specifications.

4.1 EXPAND

`Expand` transforms the current behaviour into an equivalent LOTOS behaviour, which contains only *visible* and *invisible actions, action prefixes, sum-expressions, choices, exits, stops, guards* and *processes*. This transformed behaviour is strong/weak[2] bisimulation equivalent to the original.

Syntax:

```
Expand [<depth>] [-v] [-i]
```

<depth> is an integer number. It limits the maximum depth of the expansion measured in number of actions (visible or invisible) generated from the root of the expansion. A negative `<depth>` means no bound. Default argument is infinite depth.

This expansion performs reduced state explorations of the state space of the specification because it looks for duplicate states, so that the exploration backtracks when a duplication is detected. Two states are detected as duplicate when their behaviours are identical.

The expansion is a tree like exploration of the transition system with backtracks when one of the following conditions is found: an *exit* or a *stop* statements is found, the specified `<depth>` is reached, or a duplicate state is found. For the termination of this expansion only finite sorts of data values and bounded dynamic creation of processes are allowed. Therefore behaviour expressions that can produce infinite transitions can not be expanded completely.

Option **-v** is used to work in verbose mode. In this mode, during the expansion, the depth of the exploration is displayed like a list with the format *a-b/c-d*. The term *a-b/c-d* means that there is a sequence of actions which begins at depth *a* and ends at depth *b*, and which has an alternative sequence of actions beginning at depth *c* and ending at depth *d*. The number of states explored and the number of transitions generated are also displayed during the exploration.

Option **-i** causes the removal of some internal actions from the behaviour (preserving the observational congruence), so that the number of states to explore is reduced. For instance, the behaviour $a;(i;B_1|||B_2)$ is transformed into $a;(B_1|||B_2)$, which has less states and executions to explore [3] .

[2] The type of equivalence is selected in all expansion commands by means of the -i option.

[3] When weak bisimulation equivalence is selected in an expansion, the information relative to which hidden gate the internal action comes from (which normally appears inside comment brackets `(**)`) is deleted from the resultant specification. The reason for that is to avoid apparent transition gaps due to the removal of

After the expansion some statistics about the transitions, states, deadlocks and duplicate states generated are printed.

For example, the EFSM of the *Credit* specification is calculated and printed below:

```
lola> expand -1 -v

                Exploration Tree                              |Transits| States
  0-    3/    2-    3/    2-    3/    1-    2/    1-    2/ |      10|       3
  1-    2/    0-    1/    0-    1/.

  Analysed states       = 3
  Generated transitions = 10
  Duplicated states     = 8
  Deadlocks             = 0

lola> print -p -1

 specification credit [bank,work,sleep] : noexit
 behaviour
    duplicate2 [bank,work,sleep]
 where
 process duplicate2 [bank,work,sleep] : noexit :=
       bank  ! borrow;
       duplicate0 [bank,work,sleep]
    [] sleep;
       duplicate2 [bank,work,sleep]
    [] work;
       duplicate2 [bank,work,sleep]
 endproc
 process duplicate0 [bank,work,sleep] : noexit :=
       bank  ! borrow;
       duplicate1 [bank,work,sleep]
    [] bank  ! pay;
       duplicate2 [bank,work,sleep]
    [] sleep;
       duplicate0 [bank,work,sleep]
    [] work;
       duplicate0 [bank,work,sleep]
 endproc
 process duplicate1 [bank,work,sleep] : noexit :=
       bank  ! pay;
       duplicate0 [bank,work,sleep]
    [] sleep;
       duplicate1 [bank,work,sleep]
    [] work;
       duplicate1 [bank,work,sleep]
 endproc
 endspec
```

Usually, `Expand` and `VarExpand` computation time is shortened by expanding all the process definitions before expanding the main behaviour, in a bottom-up fashion (from the innermost instantiated to the outermost). Bottom-up expansions increase expansion performance mainly in behaviours with very frequently instantiated processes.

---

internal actions that may confuse the user.

4.2 VAREXPAND

`VarExpand` performs the parameterized expansion [QPF89b]. It expands the behaviours like `Expand`, but keeping variables symbolic and detecting parameterized duplicate states (without evaluation of value expressions). Hence, it produces a more compressed representation of the transition system and reduces the state exploration with respect to the `Expand` transformation because the value expressions are always handled symbolically. Here, the duplicate states are detected when behaviours are equal except for some data values, which may be different. The transformed behaviour is also strong/weak bisimulation equivalent to the original.

Syntax:

```
VarExpand [<depth>] [-v] [-i]
```

Options <depth>, `-v` and `-i` have the same meaning that in `Expand`.

With `VarExpand` the exploration backtracks when an *exit* or a *stop* statement is found, the specified <depth> is reached, or a parameterized duplicate state is found. The conditions for the termination of the expansion are now different, because infinite sorts are allowed (data expressions are treated in a parameterized way). However, unbounded dynamic creation of processes can produce divergence. This expansion stops much more quickly than `Expand` and produces less states.

The parameterized EFSM of the *Credit* specification is presented below:

```
lola> varexpand -1 -v

                Exploration Tree                        |Transits.| States
  0-    1/    0-    1/    0-    1/    0-    1/.

  Analysed states       = 1
  Generated transitions = 4
  Duplicated states     = 4
  Deadlocks             = 0

lola> print -p -1

 specification credit [bank,work,sleep]  : noexit
 behaviour
    duplicate0 [bank,work,sleep] (0, inc(inc(0)))
 where
 process duplicate0 [bank,work,sleep](debt_9:money, max_8:money):noexit :=
       bank  ! borrow [not(debt_9 eq max_8) = true];
       duplicate0 [bank,work,sleep] (inc(debt_9), max_8)
    [] bank  ! pay [not(debt_9 eq 0) = true];
       duplicate0 [bank,work,sleep] (dec(debt_9), max_8)
    [] sleep;
       duplicate0 [bank,work,sleep] (debt_9, max_8)
    [] work;
       duplicate0 [bank,work,sleep] (debt_9, max_8)
 endproc
 endspec
```

4.3 FREEEXPAND

`FreeExpand` expands the behaviours like `Expand`, but it does not detect duplicate states. The transformed behaviour is also strong/weak bisimulation equivalent to the original.

Syntax:

```
FreeExpand [<depth>] [-v] [-i]
```

Options <depth>, `-v` and `-i` have the same meaning that in `Expand`.

The exploration of the transition system only backtracks when an *exit* or a *stop* statement is found, or the specified <depth> is reached. Therefore, behaviours that can produce infinite lenght sequences of transitions can not be expanded completely, i.e. unbounded <depths> should not be given with never terminating behaviours. `FreeExpand` is faster than `Expand` and `VarExpand`.

For example, to obtain all the transitions produced by *Credit* until a depth of two actions the following command must be executed:

```
lola> free 2 -v

                Exploration Tree                          |Transits| States
  0-    2/    1-    2/    1-    2/    1-    2/    0-    2/ |      10|       3
  1-    2/    1-    2/    0-    2/    1-    2/    1-    2/.|      13|       4

  Analysed states       = 4
  Generated transitions = 13
  Duplicated states     = 0
  Deadlocks             = 0

lola> print 2

 specification credit [bank,work,sleep] : noexit
 behaviour
       bank  ! borrow;
       (   bank  ! borrow; ......
        [] bank  ! pay; ......
        [] sleep; ......
        [] work; ......
       )
    [] sleep;
       (   bank  ! borrow; ......
        [] sleep; ......
        [] work; ......
       )
    [] work;
       (   bank  ! borrow; ......
        [] sleep; ......
        [] work; ......
       )
 endspec
```

4.4 INTEREXPAND

This command computes the *Interleaved Expansion* [QLP93] of the current behaviour .

Syntax:

```
InterExpand [<depth>] [-d|-p] [-v]
```

Option **-d** is used to perform reduced state explorations, looking for duplicate states.
Option **-p** forces parameterized duplicate state detection.
Option **-v** causes the interleaved expansion to work in verbose mode. Unlike section 4.1 description of option **-v**, the depth of exploration is given in terms of *synchronizations* instead of in number of transitions.

`InterExpand` produces a non-LOTOS specification. This transformation introduces three new statements: the *IT* operator, the *termination* and the *continuation set*. These statements represent the transition system of a LOTOS specification in a different way:
$IT(B_1, \cup_{c \in C} < c > B_c)$. The intuition of this expression is the following. The *IT* operator has a behaviour B_1 which evolves as any LOTOS behaviour containing action prefix, inaction, choice and pure interleaving. B_1 has especial events tagged by integers, called *terminations*, which label some of their states. These terminations are indexes of a set of labelled behaviours $\cup_{c \in C} < c > B_c$ (the continuation set) such that B_c is enabled when B_1 is in a state that offers the set of termination labels composing $< c >$. For instance, the interleaved expansion with duplicate behaviour detection of the specification of a *2-element buffer* :

```
specification buffer2 [input,m,output]  : noexit
  type data is sorts data endtype
 behaviour
  hide m in  b1 [input,m]  |[m]|  b1 [m,output]
 where
  process b1 [input,output] : noexit :=
    input ?x:data;  output !x;  b1 [input,output]
  endproc
endspec
```

is the following:

```
IT( input  ? x_8:data; duplicate0 [input,output] (x_8)
    ,
    <1(x_10:data) ,2> duplicate0 [input,output] (x_10)
  )
where
process duplicate0 [input,output] (x_8:data) : noexit :=
   i; (* m ! x_8 *)
   (  input  ? x_10:data; 1(x_10)
    |||
      output  ! x_8; 2
   )
endproc
```

All the synchronizations of the specification are computed and the cardinality of each synchronization is reflected in the number of terminations that label its continuation. Note that terminations have parameters like processes do.

5 Testing

LOLA follows the definition of *Testing Equivalence* of de Nicola and Hennessy [dNH84]: Tests are passed by specifying a test process and obliging it to synchronize with the behaviour under test. The results of the test are classified into three classes: *Reject*, *Must Pass* and *May pass* .

LOLA implements this testing methodology using only LOTOS:

1. Each test is represented as a LOTOS process which must contain a special termination event which indicates the successful termination of the test. This termination event cannot appear in the behaviour under test.

2. Each test is composed in parallel with the behaviour under test, synchronizing in the union of the gate sets of both (behaviour and test), except for the termination event. This composition is represented below in two cases, which need to be differentiated due to the syntactic constraints imposed by the language. Tests with and without the exit statement will be dealt differently because LOTOS does not allow any event to be added after an *exit* statement. The second composition is necessary only for the case where the termination behaviour exit is tested.

   - Test does not contain *exit*:

     ```
     ( BehaviourUnderTest [<events>]
       |[<events>]|
       Test [<events>, SuccessEvent]
     )
     ```

   - Test contains *exit*:

     ```
     ( BehaviourUnderTest [<events>]
       |[<events>]|
       Test [<events>, SuccessEvent]
     ) >> SuccessEvent; STOP
     ```

   Theses **compositions behaviour-test** are made by **LOLA automatically**.

The successful termination of a test in a given execution consists in reaching a state where the termination event (`SuccessEvent`) is offered. A test does not terminate in a given execution if it reaches a deadlock situation.

The *Testing Equivalence* differentiates two types of tests, *must* and *may*.

> *Definition: May test.* Given a specification L and a test T, T is a *may* test of L if it terminates for at least one execution of the system when applied to L.
>
> *Definition: Must test.* Given a specification L and a test T, T is a *must* test of L if it terminates for every execution of the system when applied to L.

A *reject* test is a test which is neither *may* nor *must*, i.e. no execution terminates successfully.

LOLA determines the response of a behaviour to a test by a state exploration of the composition behaviour–test. It analyzes the test terminations for all the possible evolutions. There are two commands to pass a test: `TestExpand` and `OneExpand`. `TestExpand` makes a complete state exploration and calculates the type of response. It determines *must*, *may* and *reject* responses. `OneExpand` explores only one randomly selected execution of the composition. Thus, it can be used only for determining *may* responses. It is specially useful when testing a specification produces state explosion and an exhaustive exploration with `TestExpand` might take too long.

Different types of tests may be used. Two of them are the *acceptance* and the *rejection* tests. An *acceptance* test determines if the specification accepts a given set of interactions (traces). A *rejection* test determines if a specification rejects a set of events in a given state, i.e. after a given trace.

The following example shows an *acceptance* test. It tests if it is possible to borrow two units of money from the bank, and return them after working. *Success* is the termination event that indicates when the executions of the test are successful. If a *must* response is obtained, then all the executions reach the successful termination event, i. e. all the traces in the test are accepted. A *may* response indicates that there are executions that do not reach the termination event, and a *reject* response indicates that all the executions are unsuccessful.

```
process accept_test_1 [bank,work,success] : noexit :=
   bank ! borrow ;
   bank ! borrow ;
   work ;
   bank ! pay ;
   bank ! pay ;
   success ;
   stop
endproc
```

If a system is nondeterministic, an *acceptance* test may have a *may* response. It is convenient to insert, in such cases, choices in the states where a nondeterministic behaviour exists, to cover all variants so that the response becomes *must*.

The next example is a *rejection* test. It tests that some events are rejected in a given state. After the initial sequence of events, the set of events that must be rejected are offered in a *choice* followed by *stop*. The last alternative in the *choice* is an internal action followed by the successful termination event. If the initial sequence of events can be observed and all the events in the alternative are rejected, the response will be *must*. There is only one path which leads to the successful termination event by executing the internal action. However, if any of the events in the *choice* can be observed, then the response will be *may*; or if the initial sequence cannot be observed, the response will be *reject*. Therefore a *must* termination indicates that the events are rejected. This example tests that the *Client* can borrow money consecutively only twice:

```
process reject_test_1 [bank,work,success] : noexit :=
   bank ! borrow ;
   bank ! borrow ;
   (   bank ! borrow ;
       stop
    [] i ;
       success ;
       stop
   )
endproc
```

Sets of acceptance and rejection tests should be produced to assess what the system must accept and reject.

5.1 TESTEXPAND

`TestExpand` analyzes the response of a specification to a given test. `TestExpand` has been conceived to perform a state exploration of the composition of the *BehaviourUnderTest* with a *Test*, analyzing only the relevant aspects for test termination. The output of `TestExpand` is the type of termination found: *must*, *may* or *reject*.

Syntax:

```
TestExpand [<depth>] <success_event> [<test_proc>] [-v [<states>]]
           [-a][-d][-e][-s][-i][-y]
           [-x <expected_response> [-q]]
           [-p <percent> [<seed>]] [-b <msize>]
```

5.1.1 Basic Procedure

The primary usage of this command is the execution in batch of a series of tests, obtaining the test reponse as soon as possible. This can be achieved using the following parameters :

<**depth**> is an integer number. It limits the maximum depth of the exploration measured in number of actions (visible or invisible) analysed. A negative *depth* means no bound.

<**success_event**> is the name of the successful termination event. This event can only appear in the test process and not in the behaviour under test. Otherwise, an error message is displayed and the test is not passed.

<**test_proc**> is the name of the LOTOS process to be composed in parallel with the behaviour under test. This parallel composition is made automatically by **LOLA** as described above, but the set of test processes to be passed must be in the specification file itself. When <**test_proc**> is not specified the analysis is done over the current sub-behaviour, which is supposed to be a hand-made test composition (in this case **LOLA** does not compose anything with the current sub-behaviour).

Option **-v** is used to work in verbose mode (see `Expand` command). If a number of <**states**> is given, then **LOLA** will display a provisional report about the exploration being carried out every <states> states explored. Regardless of this option, after the analysis, the test result and two different blocks of statistics are printed. The first block is similar to the other expansion statistics. The second provides information about the test result: the number of traces that reach *stop*, *exit*, <`success_event`> and <`depth`> is displayed.

For example, the result of testing the example specification with the test **accept_test_1** is:

```
lola> test -1 success accept_test_1 -v

   Composing behaviour and test :

       accept_test_1 [bank,work,success]
     |[bank,work,sleep]|
        bank [bank,sleep] (inc(inc(0)), 0)
      |[bank]|
        client [bank,work]

                    Exploration Tree                    |Transits| States
    0-    6/.

    Analysed states       = 6
```

```
Generated transitions = 6
Duplicated states     = 0
Deadlocks             = 0

Process Test = accept_test_1
Test result  = MUST PASS.

               successes = 1
                   stops = 0
                   exits = 0
           cuts by depth = 0
```

The result of this test is `MUST pass`, so that every single execution of our system - test behaviour goes through a state where `success` is offered. No deadlock was found (`stops = 0` and only one trace is possible.

5.1.2 Debugging Options

Options **-a**, **-d**, **-e** and **-s** should be used only to analyse errors (unexpected test responses). When any of these options is given then the current behaviour is replaced with a new behaviour where only the selected executions are left.

Option **-a** selects traces leading to the *success_event* (the behaviour after the *success_event* is replaced with *stop*).
Option **-d** selects traces that reach the specified <**depth**>.
Option **-e** selects traces leading to *exit*.
Option **-s** selects traces leading to *stop*.

Note that, since the original behaviour is replaced by the selected traces, after any of this options has been used it is necessary to **load** again the specification before applying a new test or command. On the contrary, if these options are not used, then the behaviour is not modified, so that successive `TestExpand` commands can be applied in sequence and you needn't `load` between tests. Another important practical issue is that `TestExpand` performs test response analysis with a low fixed memory amount. This feature can be spoilt by debugging options (**-a**, **-d**, **-e**, **-s**) that forces **LOLA** to record the selected exploration traces.

Option **-i** is used to preserve all the internal actions on the behaviour under test. Without this option, `TestExpand` removes the internal actions that do not alter the result of the exploration and that reduce the number of states to explore, improving the performance of the analysis. For instance, the behaviour $a;(i;B_1|||B_2)$ is transformed into $a;(B_1|||B_2)$, which has less states and executions to explore. When removing an internal action is not going to improve the performance of the analysis they are kept. Option **-i** disables this feature, so that the internal actions are NOT removed. This option should be used only to analyse errors in the specification, combined with options **-a**, **-d**, **-e** or **-s**. Moreover, this option enables **LOLA** to show the original gate names and offers of the internal actions inside comments e.g. `i;(* g !0 *)`.

`TestExpand` explores only the executions necessary to determine whether the response to a test is *must*, *may* or *reject*. So, if a test response is *may* it is not necessary to explore all the possible executions, because when at least one successful and one unsuccessful executions have been explored, the test result will not change, regardless of the termination of the unexplored executions. Option **-y** modifies that and forces **LOLA** to explore *all* the possible executions, in spite of the fact that the test result might be known without exploring all of them. This option should be used only to analyse errors in the specification.

5.1.3 Suspending Tests

As stated above, **LOLA** testing features are intended to work in batch. The following options facilitates the integration of **LOLA** in testing platforms.

Option **-x** forces **LOLA** to stop the test as soon as the test response is known to be different to the <**expected_response** >. This parameter can be either MUST, MAY or REJECT. Option **-q** makes **LOLA** exit with a 1 value if this condition is met.

5.1.4 Partial Exploration

For really huge state spaces, it may not be practical to perform exhaustive state explorations, in spite of **LOLA**'s facilities to simplify the exploration (testing equivalent minimization, early test response, ...). **LOLA** provides some facilities to test partially a specification, i.e. not trying every possible execution of the system. This means that the test coverage is not 100 % like in previous sections.

The following options enable non-exhaustive test passing:

Option **-p** forces **LOLA** to explore only a percentage of the transitions offered in each state. The selection is made randomly using <**seed**>. Note that n percent explored does not imply n percent out of the total state space analysed . <**percent**> ranges from 0 (one trace exploration) to 100 (exhaustive exploration).

Option **-b** performs a variant of the bit state hashing algorithm [Hol91], using <**msize**> MBytes. The total memory used is given by the sum of <msize> and the currently used memory (see 2.6 **Stat** command).

Both options can be used in conjunction.

5.1.5 Other considerations

One condition needed to assure that the result of the test expansion is reliable, is that no variables remain without any value assigned within guards or selection predicates during the exploration. Note that a potentially (hidden) deadlock may exist if there are unbounded variables in guards or in selection predicates. Two types of variable definition cases may lead to such a situation. The first one is in value acceptances of event denotations. This can be avoided by using tests which have only value offerings in their action denotations. The second case is with choice statements (`choice .. x:T .. [] B`). This could be resolved automatically by exploring the state space of B for all the possible values of type `T` (this is not supported). After the exploration a warning message will be displayed if any guards or selection predicates have not been resolved during the test exploration.

5.2 ONEEXPAND

This command is used to execute random traces of the current behaviour, or to compose it in parallel (like with `TestExpand`) with a test process to analize only single executions.

Syntax:

```
OneExpand <depth> [<success_event> <test_process>] [<seed> [execs]] [-v][-i]
```

When neither the <`success_event`> nor the <`test_process`> are given, **LOLA** produces any sequence of events that can be generated by the current behaviour. It they are specified, then `OneExpand` works like `TestExpand` but it only explores one random trace.

<**depth**> is an integer number. It limits the maximum depth of the exploration measured in number of actions (visible or invisible) generated. A negative *depth* means no bound.

<**success_event**> is the name of the termination event. See command `TestExpand`.

<**test_process**> is the name of the test process to be composed in parallel with the behaviour under test. See command `TestExpand`.

<**seed**> is an integer number used as the initial value for a random number generator. The analized trace is selected randomly depending on the given <`seed`>, but it is always fixed for the same behaviour and <`seed`>.

<**execs**> is the number of random traces to be executed taking as seed the previous random number generator status.

Option **-v** enables verbose mode (see `Expand` command).

Option **-i** is used to preserve the internal actions on the behaviour under test and show their original gate names and offers. See command `TestExpand`.

`OneExpand` finishes when either *stop, exit* or the <`success_event`> are found, or the specified <`depth`> is reached.

This expansion is recommended for testing specifications with a huge number of states in which `TestExpand` may spend a long time even with partial exploration options. After the exploration some statistics are printed, and the executed trace is classified as *Rejected* or *Successful execution.*

A Appendix: Non-LOTOS Operators

Most **LOLA** commands transform LOTOS specifications into other LOTOS specifications (IS-8807 compatible LOTOS). In order to achieve this, it has been necessary to give a special treatment to the *Relabelling* operation and to the composition of premises in selection predicates and guards. The *Relabelling* operator, which is created when a proccess is instantiated, is transformed into a *choice* operator. Nevertheless, a comment with the associated *Relabel* operator is kept.

$$Relabelling\ [a/b, c/d]\ in\ B \Leftrightarrow choice\ b\ in\ [a],\ d\ in\ [c]\ [\ ]\ B$$

To solve the problem of the composition of premises in selection predicates and guards, those premises are printed as a succession of guards with single premises. However, all the commands treat this sequence of operators as a unique operator; for instace, `move` cannot place the internal cursor in the middle of the sequence.

$$a\ \ ?x:t\ \ [f(x)=c1]\ \ ;\ \ stop\ \ ||\ \ a\ \ ?y:t\ \ [g(y)=c2]\ \ ;\ \ stop$$
$$\Downarrow$$
$$CHOICE\ \ ?x:t\ \ []\ \ [f(x)=c1]->\ \ [g(x)=c2]->\ \ a\ \ !x\ \ ;\ \ stop$$

B Appendix: Preprocessing

After the semantical analysis phase and before doing any transformation, the LOTOS specification is preprocessed. All data value expressions are internally rewritten, and the processes are analized to detect unguarded process instantiations.

The name of the processes are displayed as their value expressions are being rewritten.

```
Rewriting expressions in the specification.
 1 = credit
 2 = bank
 3 = client
Rewriting done.
```

Then, the LOTOS specification gets its expandability checked. All the processes within the specification are submitted to a static analysis to determine if they are not guarded. The user is warned when any *potential* recursion problem is detected, and a list with the process instantiations involved is printed.

Un example of unguarded behaviour is given by the following process:

```
process ung[a]:noexit:=
 ung[a] ||| a; stop
endproc
```

In this case, the divergence analysis step produces the following output :

```
Analysing unguarded conditions.
   2 = ung
WARNING :  Unguarded behaviour.
      See path: ung, ung
Analysis done.
```

These checkings can be enabled/disabled by means of the **Set** command.

C A timed prototype of LOLA

A preliminary experimental timed version of **LOLA** based on a Time Extended LOTOS is available for trial in this package. This model is in line with the LOTOS timed model considered in ISO within the E-LOTOS work item (Annex A of [LLF+94]). Some papers that have contributed to the model are [QF87], [QAF90],[Led92], [LL93],[MFV93],[QAF93]. The main features of the model are:

- the use of a time dense domain
- urgency for internal actions

Currently no front end tool supports Time Extended LOTOS. Hence, LOLA accepts only standard LOTOS syntax as input and therefore a trick is used for the representation of time attributes which by internal pre-processing translates them into the Timed Extended LOTOS syntax.

This feature is enabled by compiling LOLA with options -DTIME -DASAP. LOLA is not distributed with these options by default. The user should be aware that this timed version of LOLA is still a beta version and we strongly appreciate any feedback or error reports. From now on when mentioning LOLA we refer to the timed prototype of LOLA.

C.1 The Language

Time Extended LOTOS introduces timed constructions to action prefix and exit and defines a prefix delay operator as shown in Table 1. As mentioned before LOLA accepts as input standard LOTOS so we have adopted an ad hoc solution to introduce time constraints. It consists of a three-step procedure:

1. Introduce a special gate named *time* in all gate lists of the specification (specification and processes declarations/instantiations).
2. Prefix every time constrained action with the special gate *time* and an offer list that will be translated into the time constraints of that action. The offer list may contain 3 offers: a variable of sort time , the interval lower and upper bound. Part of the offer list and even the gate *time* can be omitted according to the necessity as we will see latter on in an example.
3. Use the data type *time_nat* as defined in the distributed file `time.lot`.

After pre-processing the standard LOTOS input specification (with the elements we have described above) LOLA generates the equivalent ET-LOTOS one. The user may elaborate scripts in order to facilitate the generation of the input specification.

C.2 Example

In this section we will show how to describe timed LOTOS specifications in such a way that LOLA accepts them.

Let us use a timed version of the *Client* example process as described before in Section 1.4. This client can pay his debts after 2 time units and no latter than 10 time units after the last event. No assumption is made about the time units. Let it be, for instance, months.

Name	ET-LOTOS syntax	LOLA syntax
Internal Action Prefix	$i\{t\ in\ t_1..t_2\};B$	$time?t:time!t_1!t_2;i;B$
	$i\{t_1..t_2\};B$	$time!t_1!t_2;i;B$
	$i\{t_1\};B$	$time!t_1;i;B$
Observable Action Prefix	$gd_1...d_n\{t\ in\ t_1..t_2\}[SP];B$	$time?t:time!t_1!t_2;gd_1...d_n[SP];B$
	$gd_1...d_n\{t_1..t_2\}[SP];B$	$time!t_1!t_2;gd_1...d_n[SP];B$
	$gd_1...d_n\{t_1\}[SP];B$	$time!t_1;gd_1...d_n[SP];B$
Wait	$Wait(t);B$	not supported
Process Def.	$P[g_1,..,g_n]$	$P[g_1,..,g_n,time]$
	$(x_1:s_1,..,x_n:s_n):=B$	$(x_1:s_1,..,x_n:s_n):=B$
Process Inst.	$P[g_1,..,g_n](y_1,..,y_n)$	$P[g_1,..,g_n,time](y_1,..,y_n)$
Termination	$exit(E_1,...,E_n)\{t_1..t_2\}$	$time!t_1!t_2;exit(E_1,...,E_n)$
	$exit(E_1,...,E_n)\{t_1\}$	$time!t_1;exit(E_1,...,E_n)$

Table 1: Timed LOTOS and LOLA equivalent syntax

```
PROCESS Client [ bank, work ] : NOEXIT :=
         bank !borrow  {0..inf}; Client[bank,work]
       [] bank !pay     {2..10}; Client[bank,work]
       [] work        {0..inf}; Client[bank,work]
     ENDPROC
```

That is what we get inside LOLA. The corresponding LOTOS specification which should be input to LOLA is:

```
PROCESS Client [ bank, work,time ] : NOEXIT :=
         bank !borrow  ; Client[bank,work,time]
       [] time !2!10;
         bank !pay    ; Client[bank,work,time]
       [] work        ; Client[bank,work,time]
     ENDPROC
```

As events *bank! borrow* and *work* have no time constraints then the preceding gate *time* can be omitted. By default absence of time action prefix means no time constraint, i.e. $0..inf$. The time constraints of *bank! pay* are described in *time !2!10.*

Now let us complicate our description a bit. In this version we introduce time variables. Now the instant when the client can borrow money depends on the instant he has payed the bank last time. The latter that instant (between 2 and 10) then more time it will take to receive more credit next time. Time variable *current* records the instant of time when the client pays the bank.

```
PROCESS Client [ bank, work ](last:time) : NOEXIT :=
         bank !borrow  {last-2..inf}; Client[bank,work](last)
       [] bank !pay  {current in 2..10}; Client[bank,work](current)
       [] work        {0..inf}; Client[bank,work](last)
     ENDPROC
```

The corresponding LOTOS specification which should be input to LOLA is:

```
PROCESS Client [ bank, work,time ](last:time) : NOEXIT :=
         time !last-2!inf;
            bank !borrow  ; Client[bank,work,time](last)
       [] time ?actual:time!2!10;
            bank !pay    ; Client[bank,work,time](actual)
       [] work         ; Client[bank,work,time] (last)
      ENDPROC
```

C.3 Restrictions

The main restriction of LOLA is not accepting ET-LOTOS syntax. Auxiliary *time* gates have to be introduced instead. Other restrictions are:

1. The prefix delay operator $Wait$ is not supported. This is not an important restriction as far as we have observed that this facility is seldom used in most timed specifications and usually can be translated into the explicit event-interval construction.

2. Dense time description is not supported. The current version works with a discrete time domain based on the natural numbers as defined in the type library file `time.lot` with all the limitations of LOTOS data types. No narrowing is performed with the data types.

3. Interleaved Expansion (see section 4.4) is not supported.

C.4 Compilation Flags

In order to compile the timed LOLA prototype the flags $TIME$ and $ASAP$ must be included in the list of $CFLAGS$ in the makefile

```
CFLAGS = -DTIME -DASAP -O
```

References

[dNH84] R. de Nicola and M. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34(1,2):83–133, Nov 1984.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[ISO89] ISO. *Information Processing Systems – Open Systems Interconnection – LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. IS-8807. International Standards Organization, 1989. [published 15/feb/1989].

[PL91] Santiago Pavón and Martín Llamas. The Testing Functionalities of LOLA. In Juan Quemada, José A. Mañas, and Enrique Vázquez, editors, *Formal Description Techniques, III*, pages 559–562, Madrid (ES), 1991. IFIP, Elsevier Science B.V. (North-Holland). Proceedings FORTE'90, 5–8 November, 1990.

[QFM87] Juan Quemada, Angel Fernández, and José A. Mañas. LOLA: Design and Verification of Protocols Using LOTOS. In *IBERIAN Conference on Data Communications*, Lisbon, May 1987. Also in Computer Communication Systems A. Cerveira (ed) North-Holland (1988).

[QLP93] J. Quemada, D. Larrabeiti, and S. Pavón. Compressed State Space Representation of LOTOS Specifications. In Ken J. Turner, editor, *Formal Description Techniques, VI*, pages 19 – 34, Boston, Massachussetts, EEUU, 1993. IFIP, North-Holland. Proceedings FORTE'93, 26–29 October, 1993.

[QPF89a] Juan Quemada, Santiago Pavón, and Angel Fernández. State Exploration by Transformation with LOLA. In *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, June 1989.

[QPF89b] Juan Quemada, Santiago Pavón, and Angel Fernández. Transforming LOTOS Specifications with LOLA: The Parameterized Expansion. In Ken J. Turner, editor, *Formal Description Techniques, I*, pages 45–54, Stirling, Scotland, UK, 1989. IFIP, North-Holland. Proceedings FORTE'88, 6–9 September, 1988.

[DG93] M. Diaz and R. Groz, editors. *Formal Description Techniques V*. North-Holland, 1993.

[Led92] G. Leduc. An Upward Compatible Timed Extension to LOTOS. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques IV*, pages 217–232. North-Holland, 1992.

[LL93] G. Leduc and L. Léonard. A Timed LOTOS Supporting a Dense Time Domain and Including new Timed Operators . In Diaz and Groz [DG93], pages 87–102.

[LLF+94] G. Leduc, L. Léonard, D.de Frutos, L. Llana, C. Miguel, J. Quemada, and G. Rabay. Belgian-Spanish Proposal for a Time Extended LOTOS. In J.Quemada, editor, *Working Draft on Enhancements to LOTOS, ISO/IEC JTC1/SC21/WG1*, October 1994.

[MFV93] C. Miguel, A. Fernandez, and L. Vidaller. Extending LOTOS Towards Performance Evaluation. In Diaz and Groz [DG93].

[QAF90] J. Quemada, A. Azcorra, and D. Frutos. A Timed Calculus for LOTOS. In S. T. Vuong, editor, *Formal Description Techniques II*. North-Holland, 1990.

[QAF93] J. Quemada, A. Azcorra, and D. Frutos. TIC: A TImed Calculus. *Formal Aspects of Computing*, (5:224-252), June 1993.

[QF87] J. Quemada and A. Fernández. Introduction of Quantitative Relative Time into Lotos. In *Workshop on Protocol Specification, Testing and Verification: VII*, Zurich, May 1987. IFIP.