CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
LABORATOIRE d'ANALYSE et d'ARCHITECTURE des SYSTÈMES

7, Avenue du Colonel Roche, 31077 Toulouse Cedex, France
Tel: (+33) 61 33 62 00. Telex: LAASTSE.520930F. Telefax: (+33) 61 55 35 77

# LCS USERS MANUAL

Version 5.0 of June1998

**Release Notes**
**Language Reference**
**User Interfaces**
**Libraries**

Bernard Berthomieu

Bernard.Berthomieu@laas.fr
tel: +33/0 5 61 33 63 63
fax: +33/0 5 61 33 64 11

LAAS/CNRS Report 98269, June 1998

User Interfaces

# Contents

*page*

# 1 The execution interfaces

All user interfaces provided with the LCS system are interactive. The user enters declarations, directives or commands, which are successively evaluated, each from the environment resulting from evaluation of the previous. One exits from user interfaces by typing Control D (i.e. by closing the interactive input stream `std_in`) or by evaluating expression “`quit();`”.

## 1.1 Process options

The LCS system actually provides three execution user interface (there are also two simulator interfaces, described in Section 2). The choice of interface is made when calling `lcs`, by selecting process option (-none, -light, or -strong, see the man pages of command lcs for a quick description of options). Options can be abbreviated by their first character.

- If option “-s” is specified, then the interface called, together with the runtime selected, implements the full LCS language, including strong processes.

- If no process option is specified, or option “-l” is selected (this is the default process option), then a version of LCS is run that supports only light processes. That interface actually accepts the same language and commands than the previous, but strong compositions of processes are interpreted exactly like light compositions (all processes share the same store). That interface may be used when processes are needed, but the application does not require processes to have private references, what makes it suitable for the vast majority of applications.

- Finally, if the “-n” option is passed, the interface and runtime called do not support behaviors at all. This interface accepts only the functional subset of LCS (everything except behaviors and events); it has the same reserved words and behavior than a Standard ML environment (besides the slight differences in the language discussed in the *Language Reference* manual). Its performances are not significantly better than those of the previous interface/runtime, except for possibly smaller garbage collection times since the user interface is smaller in size.

By default, lcs selects the “-light” option, which is convenient for nearly all purposes. In what follows, we will make precise when the features described require the strong processes option (“-s”).

## 1.2 The top-level type

The LCS top-level is itself implemented as a system of communicating processes which initially are the only running processes. The top-level provides specific commands for starting processes from behavior expressions. The effects of these commands are that of creating processes to run in parallel with all other currently running processes, including the top-level processes. Any running process may potentially communicate with any other running process.

Since processes successively started may have to communicate, it is essential that these processes have compatible types in the sense that they all transmit or receive messages of same types on ports with same labels. This typing constraint is enforced when processes are started, rather than when declared. In other words, port labels are not statically bound to communication lines at compile-time, but are rather dynamically bound. This is not an exception to the “static binding” rules of ML, as ports do not denote values, it is further consistent with the restriction and relabelling constructions offered by the language. So, users have not to worry about the types of the running processes when declaring agents, but they must worry about the types of behaviors when started (i.e. when turned into processes).

For enforcing type compatibility between running processes, the LCS top-level maintains a behavior type which, at any time, is the type of the parallel composition of all processes started so far from the beginning of the session. This type may also be seen as the type of the top-level process, incrementally built from the type of all processes started, since this process must have the same type than that of all agents running in parallel with it.

The top-level type, initially unconstrained (`_a`) is updated by process creation commands, and is reset to the unconstrained behavior type by execution of the **`kill`** command, soon to be introduced.

## 1.3 start, sstart, starting processes on background

Commands **`start`** and **`sstart`** start processes to run in background. These commands return immediately, binding to the identifier `it` the value of the behavior just started with, as type assigned, the current top-level type unified with the type of the behavior started. Command **`start`** differs from command **`sstart`** by the store it assigns to the process being started. The former creates a process that shares its store with the top-level process, the latter (the prefixing letter "s" stands for "strong") creates a process with its own private store, consisting of a copy of the current top-level store. Obviously, the difference is only visible when lcs is called with the "-s" process option, otherwise both commands work as **`start`**.

```
example% lcs
lcs -l version 5.0, created 11-5-1998
val it = () : unit

- start stop;
val it = - : _a                     (* the initial toplevel type  *)

- agent a = p! 1 => stop
= and   b = p! "hello\n" => stop
= and   c = p?_ => c;
val a = - : {p: int}_a
val b = - : {p: string}_a
val c = - : {p: 'a}_b

- start c;
val it = - : {p: 'a}_b              (* toplevel type is extended *)

- start a;
val it = - : {p: int}_a             (* toplevel type is extended *)

- start b;
types mismatch                (* p has type int in toplevel type  *)
instance of type: string
should match:     int
typecheck failed

- start b {q/p};
val it = - : {p: int, q: string}_a
```

## 1.4 call, scall, return, starting processes on foreground

Though the start commands covers almost all needs for process creation, it is occasionally useful to run lcs processes in foreground, the top-level being suspended until computation terminates, rather than in background. Typical cases are running applications that require interactions on the same window that the top-level window (if the top-level was running simultaneously, it could not be decided which process should be the recipient of keyboard inputs).

If the implementation of the start commands seems "natural" in a CCS framework, implementing a call command, which would return when the computation is complete, is more difficult, for the simple reason that the property "computation is terminated" cannot be decided in general, as long as some process runs in parallel with the process called.

Foreground running capabilities, with return to top-level on "completion", is provided in LCS by a combined use of the **call**, or **scall**, command, and of the **return** predefined behavior. Commands **call** and **scall** start processes to run in foreground, the call returns on expansion of a specific **return** behavior, if executed as part of the system of processes called. For typechecking purposes, agent **return** has type _a, for communication purposes, it is equivalent to behavior **stop**.

```
- call return;
val it = - : {p: int, q: string}_a

- agent d = q? s => do (print (s:string)) => return;
val d = - : {q: string}_a

- call d;
hello
val it = - : {p: int, q: string}_a
```

It may happen inadvertently that several returns occur in the text of the behavior called. In this case, the call returns on the first "return" behavior expanded, and other returns have the same effect as behavior "**stop**". It may also happen that no return is executed in the behavior called; one may in that case force computation of the process called to proceed in background, by hitting ^C in the control window of the toplevel; this restores the toplevel loop too. Finally, it may also happen that one starts (rather than calls) an agent which includes a **return**, or that return is executed in a group of process that has already returned; **return**, in these cases, are handled as "**stop**", with a "job returned" warning printed.

```
- call d /\ d /\ b {q/p} /\ b {q/p};
hello
val it = - : {p: int, q: string}_a

- hello
warning: job returned
```

The **return** agent is a specific construction for implementing **call** commands, and is not intended to be used as an alternative for the **stop** behavior, though their effects are the same if not under a **call** command (**return**, as **call**, **scall**, **start** and **sstart** pertain to the user interface, rather than the language).

## 1.5 kill, process clean-up

There is a single process clean-up command: **kill**. Evaluation of **kill**() has the effect of aborting execution of all processes, and of restarting the top-level with the current top-level environment and store.

```
- kill ();
val it = () : unit
```

Command **kill** has the additional effects of renewing the top-level type, as previously mentioned.

At any time, the number of running processes may be printed as a side effect of the pp function (pp stands for "process-pruning"). The other effects of function pp are explained in the section on runtime systems, suffices here to notice that it also prints the number of processes currently active (including event handler processes, processes waiting for I/O, and processes implementing the user interface).

```
- pp ();
1 pp(s), 5 process(es)
val it = () : unit
```

The processes remaining after a **kill** command are those implementing the execution interface (their number may vary with releases).

Command **`kill`** is not provided in the restricted interface not supporting processes (process option “-n”).

## 1.6 Interrupt handling

At any time, the user may type a keyboard interrupt (bound to the control-C character under Unix). Its effect, if typed in the window controlling the top-level, is that of aborting the ongoing evaluation, if any, and of restarting the top-level with the same user environment as before evaluation of the command aborted (except for any side-effects, which are not undone by interruption).

Interrupting an evaluation at top-level has no effects on the system of processes running in background, unless some of these processes were explicitly controlled by the same window than the top-level. Interrupting a **`call`** has the effects of backgrounding execution of the process called and of resuming the top-level.

The interruption mechanism is designed so that it is flexible enough for attaching a control terminal to any process. A control terminal is some physical window from which controls may be sent to a process (see Section 2.3 of the *Language Reference* manual, and structure `Beh` in the *Libraries* manual).

Finally, interrupt handling in the restricted “-n” interface is that of Standard ML (Control-C typed in the console window raises exception `Interrupt`).

## 1.7 The architecture of the top-level

(Process abstraction of the start/call/kill commands, TO BE PROVIDED)

## 1.8 reboot, generating standalone applications

A last command related with process creation is the **`reboot`** command. The **`reboot`** command, which takes a behavior as argument, has the effect of starting its argument and making it the sole process running on the lcs machine. This process inherits the current store of the top-level; the garbage collector will remove from the heap all cells not needed for operation of the application process.

Command **`reboot`**, in conjunction with function `save` or `dump` for creating core images of the state of the runtime, allows the user to build stand-alone applications. The only constraint for command **`reboot`** is that the type of its argument must have the behavior type constant “_”. This is explained by the fact that, as this application will replace all running processes, it does not need to communicate with any other process (obviously, running the application may create a system of arbitrarily many communicating processes).

The **`reboot`** command never returns. If no progress of the process installed is possible, then the LCS virtual machine will exit; it is up to the user to provide means for terminating execution of its standalone applications.

In the “-n” restricted interface, `reboot` is provided as a function, rather than a command, with type `(unit-> unit)-> unit`. It allows to restart the lcs -n runtime on any user provided function of type `unit->unit`.

## 1.9 Passing parameters to lcs applications

(This is explained in the *Libraries* manual, structure `System`, function `arg_list`, and the *Release Notes* and man pages, see flag “-a” of the `lcs` command).

## 1.10 save, load, dump, restore, saving/loading images

Function `dump` takes a pair of strings as argument. The first is the name to be given to the application created, the second is a sequence of parameters of the lcs command to be passed implicitly to the lcs command when calling the application. It has the side effect of creating a file, whose name is the string passed as argument of `dump`, holding a copy of the state of the runtime. The file may be restored later from the LCS top-level, using function `restore`, or it may be invoked explicitly as the core to be run when calling LCS, or it may just be run. When the file has been loaded, computation proceeds exactly as if returning from the `dump` function call that produced it. Function dump is also described in the *Libraries* manual, structure `System`.

Core images files are ASCII files and are portable across machines. The format in which core images are written is independent of the machine on which the file was created (e.g. one may use core files produced on Sun workstations on a Silicon Graphics station, and vice-versa).

Functions `save` and `load` are specialized versions of functions `dump` and `restore`. `save` is intended to be used from an LCS top-level for saving and restoring images of top-levels (including the user environment). Function `save` creates a core image and restarts the user interface. Function `load` behaves like restore, except it uses the lcs search path to locate the core to be loaded.

As an example, the following sequence of commands produces a core image of an application that opens four windows, echoes lines typed in the first three windows in the fourth window, and exits when a Control-C is typed in the fourth window. The command line parameter "-q" disables printing of the LCS banner and of all runtime messages when running the application:

```
agent echo_demo () =
   let val nm = (10,20)
       val w1 = open_set_win ("xterm","w1",(100,100),nm,"","","",1)
       and w2 = open_set_win ("xterm","w2",(300,100),nm,"","","",1)
       and w3 = open_set_win ("xterm","w3",(500,100),nm,"","","",1)
       and w4 = open_set_win ("xterm","w4",(800,100),nm,"","","",1)
       agent readw w = let val s = input_line_win w in p! s => readw w end
       agent echo w = p? s => do (output_win (w,s)) => echo w
       event BYE
       agent exit w = let val c = wait_ctrl_win w in signal BYE end
   in ((readw w1 /\ readw w2 /\ readw w3 /\ echo w4 /\ exit w4) {/p}
      catch BYE => do (map close_win [w1,w2,w3,w4]; print "bye\n") =>
                   stop)
   end
   handle Io _ => do (print "got an IO exception\n") => stop;

reboot do (dump ("wecho","-q")) => echo_demo ();
```

The `dump` function produces a core image for this application of less than 6 kilobytes (compared to the about 330 kilobytes for an LCS execution interface). This core image may be loaded from the LCS top-level (by function `restore` or `load` from structure `System`), or run with the `lcs` command (`lcs -f wecho`) or simply run as a stand-alone application (just type "`wecho`").

## 1.11 Observing progress of process systems

Systems of processes typically operate on streams, bound either to windows or files, but do not generally return results printed on the user's terminal as the evaluation of a functional expression at top-level would do. Progress of a system of processes, or termination, does not necessarily show up on the terminal.

Structure `Pio` provides some utilities to help observe progress of process systems. It provides behavioral interfaces to the standard input and output, as well as suggested behavioral interfaces for file and window input/output. Behavior `Stdio`, for instance, provides a port names `stdout`; any string sent to that port is printed on the user's terminal as a side-effect:

```
- use "pio";
val it = () : unit

- open Pio;
...

- start Stdio;
val it = - : {stdin: string, stdout: string}_a
```

```
- call stdin? s => stdout! s => return;
? hello                                    (* typed-in by the user *)
! hello                                    (* echoed by Stdout     *)
val it = - : {stdin: string, stdout: string}_a
```

Obviously, the user may always use the stream output primitives to make its processes print themselves information about their progress or results.

For debugging behaviors, one might use the LCS simulator too, which allow the user to interactively expand behaviors and visualize their communications; this user interface is described in Section 2 of this document.

## 1.12 Interfacing with the host operating system

Structure `Prelude.Host` provides a number of primitives for communication with the host operating system or accessing some of the facilities it provides. These primitives include an `execute` function which starts a Unix process and links its standard input and standard output with LCS streams returned as results of `execute`. No interface with Unix signals is currently provided.

## 1.13 Some advice to new users

Concurrent programming is difficult, and requires a discipline and mechanisms of thought of its own. We would consider the goals of the LCS project reached if the tools it provides help make concurrent programming a little less difficult an activity. Our experience taught us some recipes:

- When a strange behavior is observed while running an application, we recommend that you analyze and test your system of processes, after cutting it into smaller pieces. Then, test interactions between your program components.
- Choose a discipline for communication ports naming. That retained in the LCS libraries and examples is, except for particular cases, to name `inp1`, `inp2`, etc. and `out1`, `out2`, etc., the input and output ports of all processes, respectively. This helps designing a set of generic combinators (such as piping combinators) which will help intuition for building complex systems of processes.
- For every application, choose a discipline for concurrency. Concurrent programming, when parallelism is made explicit, offers more alternatives for programming an application than sequential programming does, from the least parallel version (fully functional) to the most parallel version (fully behavioral).
- Avoid as much as possible concurrent assignment of references shared by several processes. Though the implementation supports shared references, it does not enforce any consistency protocol for their updates. Concurrent assignments are most often the source of inextricable misbehaviors.
- Similarly, avoid, as much as possible, performing stream IOs within actions. The coexistence of two different input/output mechanisms in the language (stream-based functional and behavioral), may occasionally cause some strange (though explainable) scheduling effects.

# 2 The simulation interfaces

## 2.1 Introduction

The execution interfaces (`lcs`), with its runtime systems, are provided for optimal speed execution of LCS programs. The simulation user interfaces (`slcs` or `glcs`) provide means for testing the behavior of systems of processes. The simulators have also proven to be a valuable tool for teaching, or learning, CCS, in its LCS implementation. The `glcs` interface extends the textual-only `slcs` interface with a graphic window picturing the behaviors simulated.

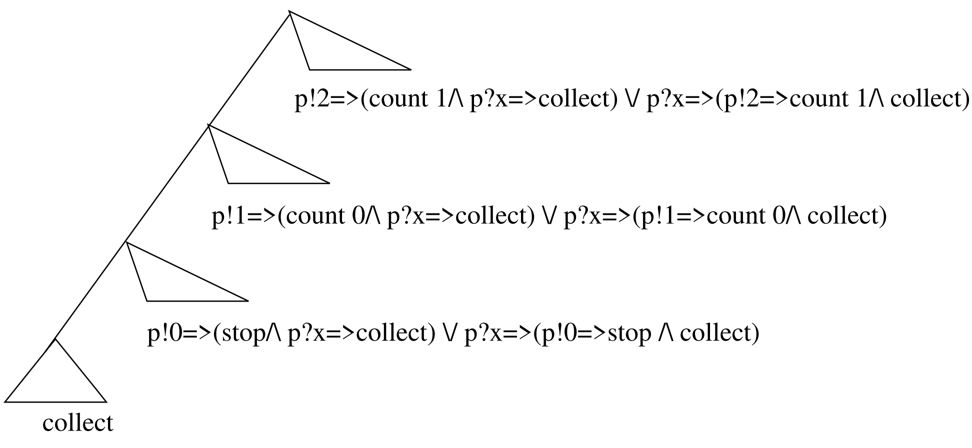
While in the execution interfaces, behavior values are used to create processes; in the simulators, behavior values are handled as values of a built-in, abstract, behavior types, with functions provided for expanding a behavior (transforming it into another), and facilities for visualizing and organizing these transformations. Behaviors are just another abstract data type, except that they enjoy a built-in concrete syntax.

The simulators interpret behaviors as communication trees [Mil 80]. The expansion commands of the simulators develop a random path in those trees, performing possible communications and/or commitments. The result is a behavior that has the same communication capabilities as the initial one, but with some paths developed. The expansion rule used is exactly an implementation of the CCS expansion theorem.

The interpreter informs the user of its progress by printing the communication traffic on visible ports on the user's terminal. The interpreter prints "deadlock" when the path taken cannot be developed farther. Here is an example of expansion:

```
- fun count n = if n<0 then stop else p! n => count (n-1)
= agent collect = p? _ => collect;
val count = - : int -> {p: int}_a
val collect = - : {p: 'a}_a

- try collect /\ count 2;
p !? 2 : int
p !? 1 : int
p !? 0 : int
deadlock
val it = - : {p: int}_a
val next = - : {p: int}_a
```

With every behavior expanded, one may associate a communication tree [Mil 80]. For the example above, this tree could be pictured as follows:



## 2.2 Language and features accepted

The simulation interfaces accept the same language that the execution interfaces but, due to the particular interpretation of behaviors in this environment, not all features of the language are supported. The main difference is the absence of support for private process stores, all behaviors operate on the same, shared, store. Worse, the store is shared among all the alternative behaviors constituting the communication trees, i.e. going backward in the communication tree, as possible using the "`right`" simulation command (to be described), does not undo side-effects. So, the simulators should not be used on behaviors using side-effects (store assignment or input/output)[1].

In the following sections, the features of the simulators are presented in textual form, as if using the textual-only slcs interface; the mixed graphic+textual interface will be introduced later, in Section 2.7.

## 2.3 Expansion requests

`slcs` allows the expansion of one behavior at a time. There are three possible expansion commands: **`xpand`**, **`run`** and **`scan`**. All three are based upon the CCS expansion theorem.

### xpand expansion requests

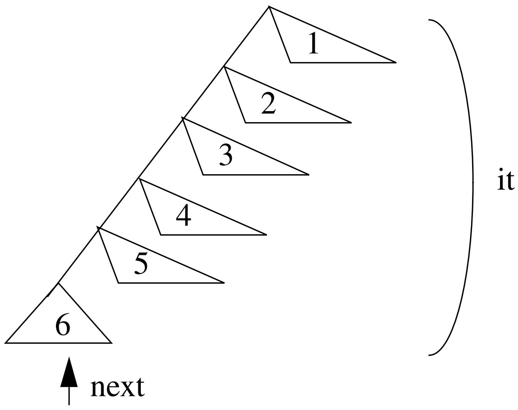
The **`xpand`** command expands a behavior, evaluating rendez-vous and/or commitments. The request develops a randomly chosen path among all those possible and builds as a result a behavior which has the same communication capabilities as the original behavior. The result behavior may be seen as a tree in which a path is developed, and in which untaken paths appear as non-developed subtrees with all necessary informations for resumption embodied.

```
example% slcs
LCS Project - CNRS/LAAS - 1991
slcs version 3.0, created 16-9-1991
val it = () : unit

- fun btree l k = if k=0 then stop else
=       p! (1::l) => btree (1::l) (k-1)
=    \/ p! (2::l) => btree (2::l) (k-1)
= agent collect = p?_ => collect;
val btree = - : int list -> {p: int list}_a
val collect = - {p: 'a}_a

- xpand collect /\ btree [] 5;
p !? [2] : int list
p !? [2, 2] : int list
p !? [1, 2, 2] : int list
p !? [2, 1, 2, 2] : int list
p !? [1, 2, 1, 2, 2] : int list
deadlock
val it = - : {p: int list}_a
val next = - : {p: int list}_a
```

The communication traffic on visible ports is printed on the user's terminal as a side effect. The message `deadlock` is printed when the path taken cannot be developed farther. Internally, the result of this expansion

1. The V5.0 simulators do not support aliasing construct yet (renaming of a simple port into an extended port), nor events and the **`signal`**/**`catch`** constructions. We hope to provide them in a forthcoming release.

may be represented as the following tree of behaviors; which can be read as a behavior, interpreting nodes as the CCS-sums of the behaviors attached to that node and edges as commitments (CCS-$\tau$ moves):




The behavior sent for expansion is:

```
collect  /\  btree [] 5
```

Or, after unfolding recursion and applying distributivity of compositions:

```
    (p?_ => collect  /\  p! [1] => btree [1] 4)
\/  (p?_ => collect  /\  p! [2] => btree [2] 4)
```

Now, using the CCS expansion theorem, this behavior is equivalent to:

```
=>B1  \/  B2
```

where B1 is

```
collect  /\  btree [2] 4
```

and B2 is

```
     (p?_ => collect  /\  p! [1] => btree [1] 4)
 \/  p?_ => (collect /\ p! [2] => btree [2] 4)
 \/  p! [2] => (p?_ => collect  /\  btree [2] 4)
```

The subtree (1) in the tree pictured above represents behavior B2; the edges represent commitments; the tree in which the branch (1) and the first edge are removed represents behavior B1 after four more successive applications of the expansion theorem.

The whole result tree, taken as a behavior, is bound after expansion to identifier `it`; the subtree that was about to be expanded is bound to the identifier `next` (this subtree is that pointed to by an arrow in our representations). The agent `next` receives the same type as `it`.

It must be noticed that the user does not realize here an observation in the sense of [Mil 80]; the user acts an a privileged witness who does not influence the behavior of the agent.

## run expansion requests

Keeping all alternative behaviors is undoubtedly useful for testing agents but, obviously, this feature is time and space consuming for long computations, even if the size of the resumption information is kept small.

Under a **run** expansion request, the alternative behavior is thrown after every move. The result is identical to the value of the agent next, taken after a computation under the substitutive **xpand** rule. Under the non-substitutive **run** rule, identifiers `it` is nor bound; only identifier `next` is.

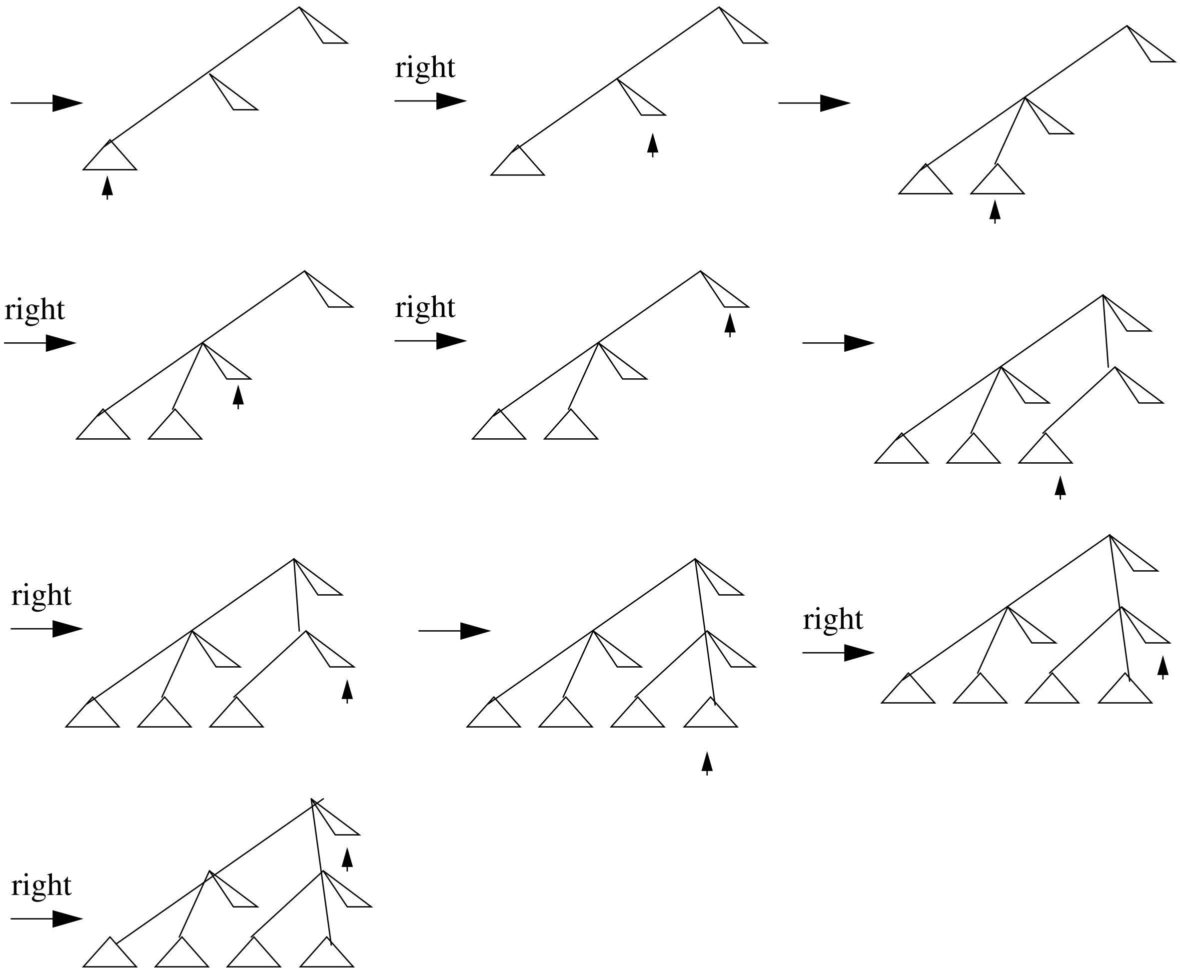
```
- run collect /\ btree [] 5;
p !? [2] : int list
p !? [2, 2] : int list
p !? [1, 2, 2] : int list
p !? [2, 1, 2, 2] : int list
p !? [1, 2, 1, 2, 2] : int list
deadlock
val next = - : {p: int list}_a
```

## scan expansion requests

For tricky cases, the user may ask, using a **scan** expansion request, for a full expansion of the behavior tree, down to some programmed depth.

```
- scan collect /\ btree [] 2;
p !? [2] : int list
p !? [2, 2] : int list
deadlock
> right
p !? [1, 2] : int list
deadlock
> right
deadlock
> right
p !? [1] : int list
p !? [2, 1] : int list
deadlock
> right
p !? [1, 1] : int list
deadlock
> right
deadlock
> right
deadlock
> scan complete
val it = - : {p: int list}_a
val next = - : {p: int list}_a
```

The different steps of this expansion are explained by the picture below, in which unlabeled arrows represent sequences of actions resulting from the expansion of the behavior pointed to in the tree on the left side of the arrow.




After a **scan**, the value of `next` is that of the last subtree of which the expansion was attempted (the right-most subtree in our representation).

It should be noticed that, in practice, the use of **scan** should be rare since communication trees generally grow very quickly.

## try, mode

The full syntax for expansion requests requires that the behavior expression be prefixed by an expansion command (one of **xpand**, **run** or **scan**). Alternatively, users may use a generic **try** command, defaulting to an expansion request established by the **mode** command. A **try** command defaults to **xpand** if the default mode is **xpand**, or to a **run** or **scan** command otherwise.

The **mode** command may be used either for changing the default mode or for enquiring about the current default mode. One can switch from one default evaluation mode to the other by sending a **mode** command with a string argument:

```
- mode ();
is run

- mode "xpand";
was run
```

```
- try collect /\ btree [] 2;
p !? [2] : int list
p !? [2, 2] : int list
deadlock
val it = - : {p: int list}_a
val next = - : {p: int list}_a
```

It should be noted in the example above that command **`mode`**, as well as other commands soon to be discussed, does not obey the typing rules enforced for functions or agents (**`mode`** takes either a string or the empty product as argument). Commands in SLCS have their own syntax checking, implemented separately from the typechecker for expressions.

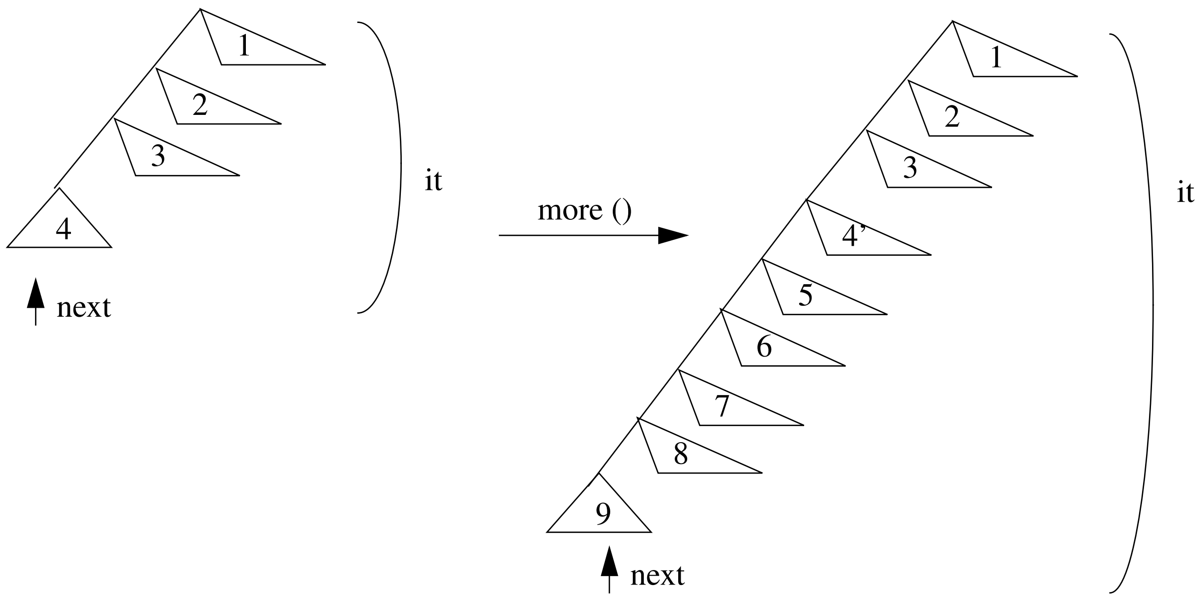
## 2.4 Interrupting and resuming expansion

### steps, more

Expansions do not necessarily terminate; interruption of expansions can be programmed with the **`steps`** command. This command sets a bound on the length of the path to be developed when expanding a behavior. When this bound is reached, expansion is interrupted and a specific message is printed. The expansion can be resumed, for as many steps as currently set, by the command **`more`**.

```
- steps ();
25 steps

- steps 3;
3 steps

- agent from k = p! k => from (k+1);
val from = - : int -> {p: int}_a

- try collect /\ from 0;
p !? 0 : int
p !? 1 : int
p !? 2 : int
eval interrupted -- steps
val it = - : {p: int}_a
val next = - : {p: int}_a

- steps 5;
5 steps

- more ();
p !? 3 : int
p !? 4 : int
p !? 5 : int
p !? 6 : int
p !? 7 : int
eval interrupted -- steps
val it = - : {p: int}_a
val next = - : {p: int}_a
```

The command **`more`** resumes expansion from the point at which it was interrupted in the previous expansion (the subtree bound to identifier `next`), and resumes it according to the current default expansion mode. In other words, under xpand mode (resp. run mode, resp. scan mode), command "**`more`**()" is equivalent to "**`xpand`** `next`" (resp. "**`run`** `next`", resp "**`scan`** `next`"). In addition, on completion, and in under modes xpand or scan, the subtree bound to `next` in the tree bound to `it`, both taken before applying **`more`**, is physically replaced by the result of its expansion. This is illustrated by the picture below:




## Local communications and commitments

Occurrences of local communications (those occurring through hidden ports) and commitments are signalled by printing periods (values could not be printed, in general, as they may have types which have local scope, and, at expansion time, typing information is no longer available). This feature may be cancelled or requested using a **`commits`** command.

```
- commits ();
commitments displayed

- agent through = p? x => q! x => through;
val through = - : {p: 'a, q: 'a}_b

- try (from 5  /\ through /\ collect {q/p}) {/p};
.
q !? 5 : int
.
q !? 6 : int
.
run interrupted -- steps
val it = - : {p: int}_a
val next = - : {p: int}_a

- commits false;
commitments not displayed

- more ();
q !? 7 : int
q !? 8 : int
q !? 9 : int
```

```
run interrupted -- steps
val next = - : {p: int}_a

- commits true;
commitments displayed
```

## 2.5 Exploring alternative scenarios

### Resumption moves

After expansion has been interrupted, it may be resumed from the point at which interruption occurred, referred to as the resumption pointer in the sequel. Expansion of alternative paths (than the path chosen by the simulator) is performed after moving this resumption pointer in the communication tree prior to emitting command "**more** ()".

A number of facilities are provided for this. The commands **right**, **left**, **rmost** and **lmost** allow relative moves of the resumption pointer; commands **tag** and **jump** allow the user to name a sub-tree and make absolute moves of the pointer, respectively; finally, command **search** allows the user to move the resumption pointer to the right to the next sub-tree which may perform a move.

Resumption moves commands are defined in **xpand** and **scan** mode only, all of them would fail when used in the **run** mode. All resumption move commands update agent `next`, so that agent next is always the resumption sub-behavior, but leave agent `it` unchanged.

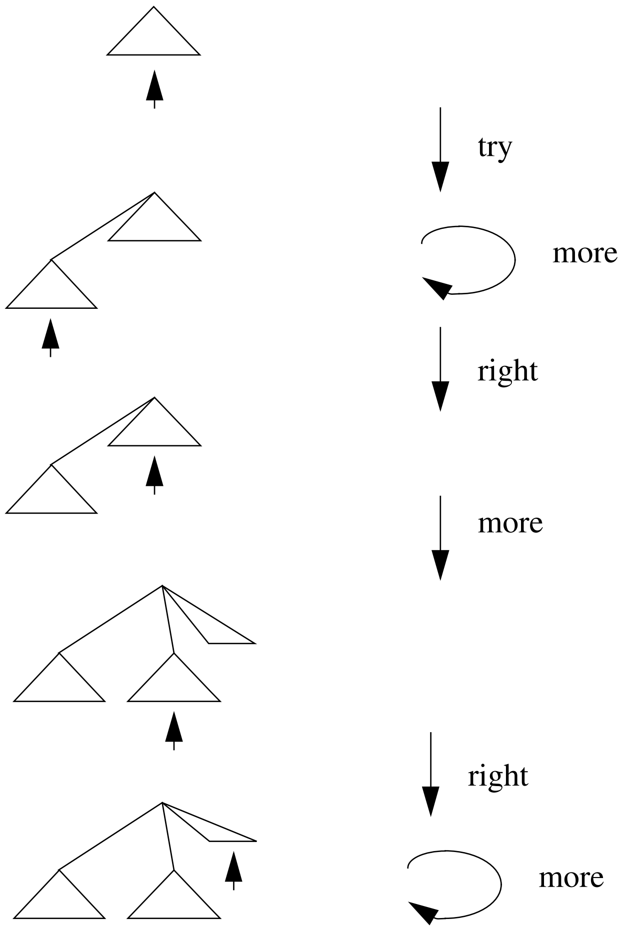
### Relative Moves: left, right, lmost, rmost

Command "**right** n" (resp. "**left** n"), where n is an integer, moves the resumption pointer n positions to the right (resp. left) in the communication tree. Command "**rmost** ()" (resp. "**lmost** ()") moves the resumption pointer to point to the rightmost (resp. leftmost) sub-tree. For assuring consistent moves, the implementation guarantees that an alternative behavior is created at each expansion step, even if this pending behavior is the **stop** behavior.

```
- agent choose = p! true => stop \/ p! false => stop;
val choose = - : {p: bool}_a

- try choose /\ p? x => stop;
p !? false : bool
deadlock
val it = - : {p: bool}_a
val next = - : {p: bool}_a

- more ();
deadlock
val it = - : {p: bool}_a
val next = - : {p: bool}_a

- right 1;
val next = - : {p: bool}_a

- more ();
p !? true
deadlock
val it = - : {p: bool}_a
val next = - : {p: bool}_a

- right 1;
```

```
val next = - : {p: bool}_a

- more ();
deadlock
val it = - : {p: bool}_a
val next = - : {p: bool}_a
```

Move requests for the resumption pointer are acknowledged by a message recalling that agent `next` was rebound, when move requests are meaningful; otherwise, an error message is printed and the command fails. The effects of the sequence of commands above may be visualized as follows.



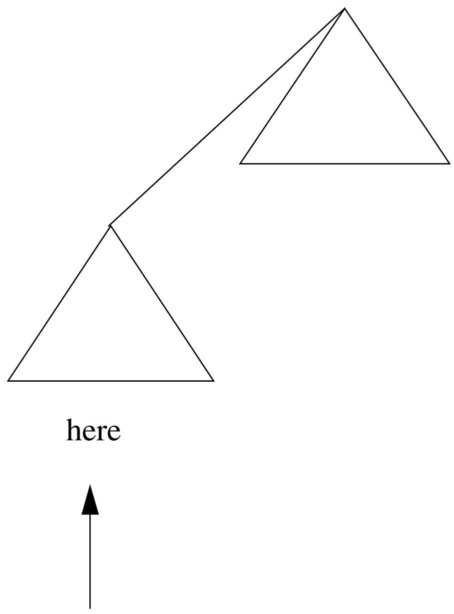

## Absolute Moves: tag, jump

Subtrees in the communication tree may be given names. Command "**`tag`** s", where s is any string, names or renames s the subtree currently pointed to by the resumption pointer. Command "**`tag`** ()" prints the tag associated with the current resumption, if non empty.

Command **`jump`**, with a string argument, moves the resumption pointer so that it points to the first sub-behavior (from leftmost towards right) which is tagged with that string; the command fails if no such sub-behavior is found.

```
- mode "xpand";
was xpand
```

```
- try stop \/ => stop;
.
deadlock
val it = - : _a
val next = - : _a

- tag "here";
here, new tag

- right 1;
val next = - : _a

- more ();
deadlock
val it = - : _a
val next = - : _a

- jump "here";
val next = - : _a

- more ();
deadlock
val it = - : _a
val next = - : _a
```




### Contextual Moves: search and dead

Command "**search** ()" starts a rightward traversal of the communication tree, starting from the subtree currently pointed to by the resumption pointer; the traversal stops when the subtree currently pointed to may perform a move (either a commitment, or a rendez-vous). Command "**dead** ()" prints a message indicating if behavior `next` is a deadlock (may not perform any move) or not.

## 2.6 Extracting and reusing behaviors

Exploring a communication tree is often made easier by cutting it into smaller pieces and analyzing them separately. Subtrees of a communication tree are extracted using identifier `next` and the resumption move commands, taking advantage of the fact that these commands update the value bound to identifier `next`.

It should be noticed, however, that tags are not inherited by the sub-behavior extracted. There can be only one current set of active tags, that of the last behavior expanded by a **try**, **xpand** or **scan** command, and eventually resumed by **more** commands.

```
- try p!0=> p!1=> (p!2=> stop \/ q! => stop) /\ p?x=> stop;
p !? 0 : int
deadlock
val it = - : {q: unit, p: int}_a
val next = - : {q: unit, p: int}_a

- try next /\ p?x=> (p?x=> stop \/ q?x=> stop);
p !? 1 : int
q !? () : unit
deadlock
val it = - : {q: unit, p: int}_a
```

```
val next = - : {q: unit, p: int}_a

- try it;
.
.
deadlock
val it = - : {q: unit, p: int}_a
val next = - : {q: unit, p: int}_a

- more ();
p !? 2 : int
deadlock
val next = - : {p: int, q}_a
```

## 2.7 The graphic viewer

### glcs

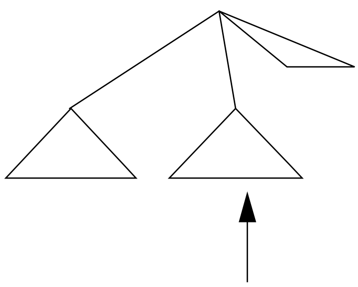
The lcs simulator is available with two interfaces: a purely textual one (`slcs`), and a both textual and graphic one (`glcs`). The latter requires the Tcl/Tk system and its `wish` interface installed on the user's site (type `wish` on your workstation to check this).

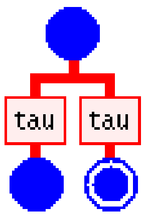
The `glcs` interface manages two synchronized windows. The lower window is textual, one can type in it declarations and simulation commands, exactly as they would be typed in the `slcs` textual interface. The upper window synchronously maintains a graphical representation of the behavior being simulated, if any. One can proceed the other way round too: once some behavior is shown in the upper window (following one of the commands **run**, **xpand**, **scan**, or **try**), the user may issue simulation commands by clicking on the graphical representation and/or the buttons in the central panel; the equivalent textual commands are simultaneously echoed in the lower window.

The `glcs` display is shown in Figure 1. The picture shows the state of both windows after some simulation command has been typed in the textual window.

### Graphical representation of behaviors

The figure below compares the representation of trees used in the glcs interface with that used so far in the examples of this manual. Behaviors are represented by filled circles. The current resumption has a white disk superposed on the node. The color of a node indicates whether it is unexplored yet (when blue), or if it is a deadlock node (grey) or a failure node (black).






The above picture does not show the node tags, but shows the actions. Node tags and actions may be shown individually (see mouse controls below), or on a whole basis through the buttons of the control panel.

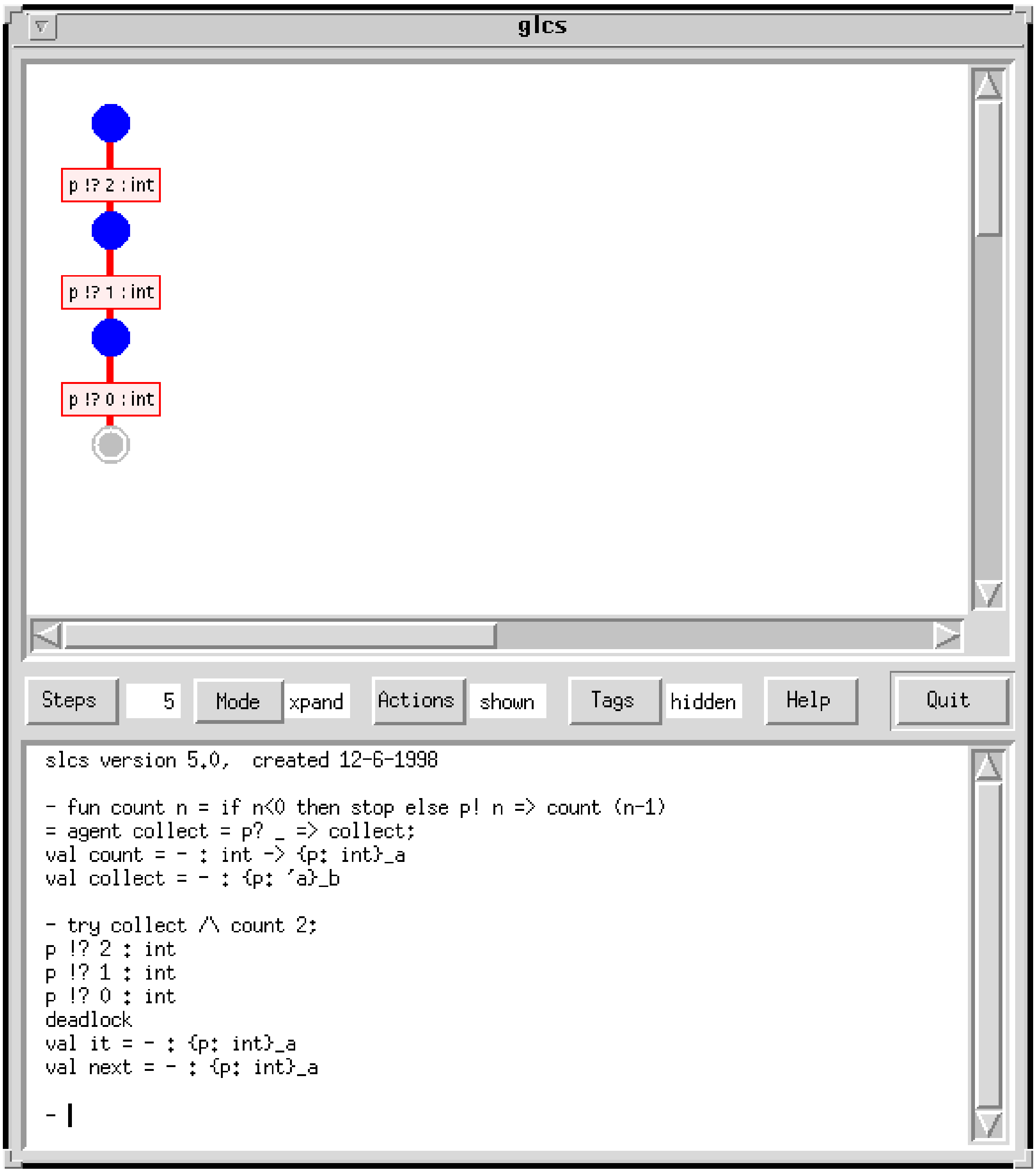



Figure 1. The glcs display

In addition, for convenience, a subtree may be folded (by clicking on with the middle mouse button). Clicking on a folded tree unfolds it. Folded trees are represented by blue triangles, possibly with a white triangle-shaped wire superposed (if the current resumption resides in the folded behavior).

## Controls

The relative sizes of the textual and graphic window can be changed by dragging the control pane up or down with the left mouse button.

The central control panel allows the user to set a number of expansion steps and an expansion mode. In addition, it controls the view of all actions and node tags (clicking on the Actions or Tags button switch on/off the view of these).

Mouse issued controls are summarized in the help window, shown in Figure 2.

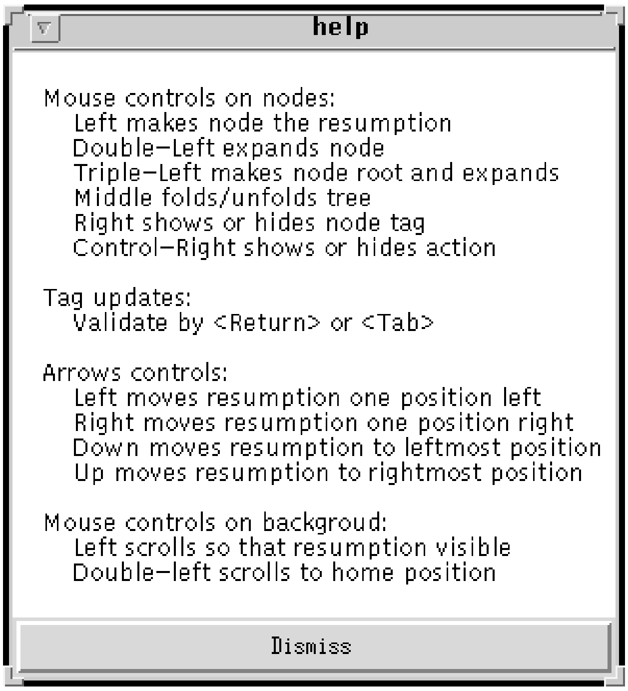



Figure 2. Mouse controls

In addition to **`steps`**, **`mode`** and **`tag`**, the commands that can be issued "graphically" by mouse clicks are **`more`**, **`right`**, **`left`**, **`lmost`**, **`rmost`**, **`move`**, and extraction of a behavior combined with an expansion request (equivalent to "**`try`** `next;`"). The correspondence between graphic and textual commands is as follows:

| *Mouse controls on nodes*: | | textual equivalence |
|---|---|---|
| Left button | makes that node the resumption | **`right`**`/`**`left`** `n;` |
| Double-Left | makes node the resumption and expands it | **`right`**`/`**`left`** `n;` **`more`** `();` |
| Triple-Left | Makes node the root of behavior and expands it | **`try`** `next;` |
| Middle | Folds/Unfolds subtree | (not applicable) |
| Right | Shows/Hides tag of node or to-arc | (not applicable) |
| Control-Right | Shows/Hides tag of to-arc | (not applicable) |

*Tag updates*:

| | |
|---|---|
| Make the tag of the node visible by clicking (Right mouse button) on the node; then update the tag. Validate the update by <Return> or <Tag>. | **`right`** `n;` **`tag`** `s;` **`left`** `n;` `or` **`left`** `n;` **`tag`** `s;` **`right`** `n;` |

*Arrows controls*:

| | | |
|---|---|---|
| Left | moves resumption one position left | **`left`** 1; |
| Right | moves resumption one position right | **`right`** 1; |
| Down | moves resumption to leftmost position | **`lmost`** (); |
| Up | moves resumption to rightmost position | **`rmost`** (); |

*Mouse controls on background of upper window*:

| | | |
|---|---|---|
| Left | scrolls so that resumption visible | (not applicable) |
| Double-left | scrolls to home position | (not applicable) |

## A simple simulation session

Figure 3 shows the series of displays gone through in the simple simulation session presented in Section 2.5, about relative moves.

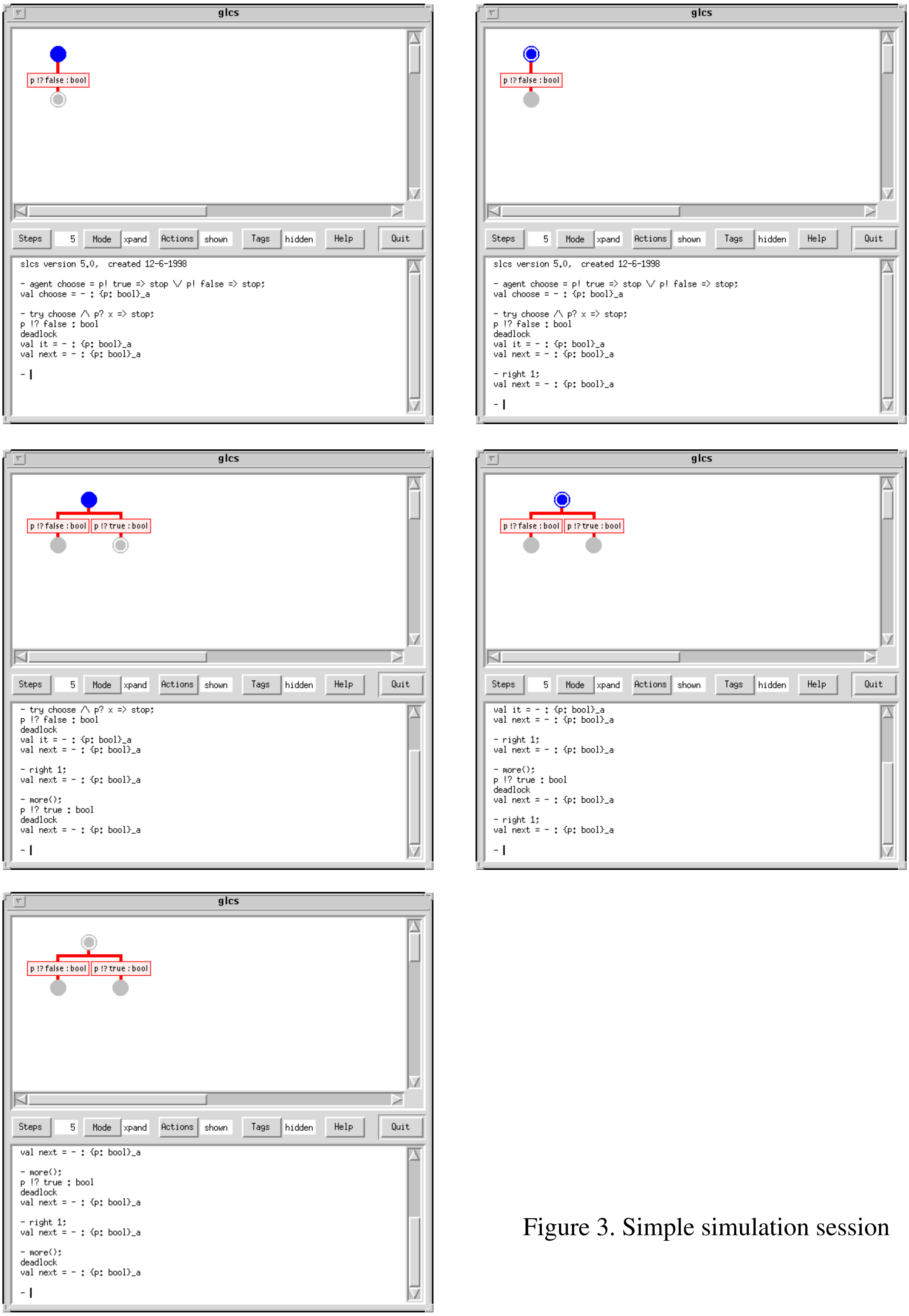



Figure 3. Simple simulation session

# 3 The runtime systems

## Runtime systems

As already mentioned, there are three runtime systems provided with the LCS system. Selection of one of them follows from the process option used when invoking the `lcs` command.

## Memory management

All runtimes provide automatic memory management, with generational garbage collection. All allocate data in a heap and use a separated stack for function calls. Heap size and stack size grow and shrink automatically from initial sizes and tune-up parameters. Check the `lcs` man page, or the release notes accompanying the distribution, for the parameters of the lcs command and lcs applications.

A heap or stack overflow would occur when the application requires more virtual space than the host operating system may grant to the application. Such overflows are unrecoverable and results in termination of the runtime.

Cells kept in the heap have size limits, but no object size overflow should appear at run-time. For some sorts of cells, this size limit is well beyond the largest virtual storage size any known operating system would grant (a heap overflow error would thus occur first); for the other kinds of cells, object size overflow are detected at compile-time. The lcs compiler may refuse to compile an expression or declaration for the reason that the code generated for it or some value computed by this code would overflow object sizes limits. This is not likely to happen, but if this ever occurs it should suffice to break the expression or declaration into smaller pieces.

Garbage collection is two spaces, stop-and-copy and generational; it also compacts the heap space. Garbage collection may be requested by the user, by the `gc` primitive (see *Libraries* manual, structure `System`).

## Scheduling

Abstractly, all processes started may be assumed to run concurrently, each with its own processor. In practice, processes share a single processor and a scheduling strategy determines processor allocation. This section makes precise the scheduling strategy of the `lcs` execution runtimes (the `slcs` and `glcs` simulators use a random scheduling strategy of their own and are not discussed here).

The LCS virtual machine maintains a queue Rp of processes waiting for the processor. The scheduling strategy is basically round-robin and communication bound. This basic strategy is further combined with time-slicing.

- Upon termination of a process, the scheduler will be called to allocate the next ready process in Rp to the processor.

- If no partner is found for an input or output action, the process offering communication is suspended and the next ready process in Rp is resumed. Otherwise the message is put in the environment of the receiver, the partner is enqueued in Rp, and the current process proceeds.

- One of the processes created upon execution of one of the LCS composition combinators will be enqueued in Rp, while the other will continue execution.

- Evaluation of commitments causes no process suspensions.

- Stream input primitives and the `sleep` primitive are non-blocking (at the level of the virtual machine), so that a process getting stuck on a stream on which no data is available, or waiting for some time to elapse, does not prevent other processes from running. However, stream input and sleep functions appear blocking to the user; they returns only when the required number of characters have been read, or the delay computed has elapsed.

- Finally, a time-slicing mechanism assures that all ready processes can make progress, even if some of them perform no communications (e.g. the top-level process, or a process that loops or performs commitments forever).

## Process pruning

Process pruning consists of the elimination of retracted communication offers or of useless event handlers from the system of processes. It is combined with garbage collection as its cost would be very high if realized at each process move. Pruning can also be requested by the user, by the `pp` primitive (see *Libraries*, structure `System`).

## Distributed programming services

Usage of most library functions in the `Network` structure require prior installation of the `lcsd` distributed services daemon; this daemon is implemented as an lcs application (see *Libraries*, structures `Network` and `Register`).

The lcsd daemon is typically started on the default host, and default port number, declared when installing the `lcs` command (see the *Release Notes* for installation of the `lcs` command). More generally, an lcs daemon can be started on any host, and with any port number, by passing these parameters to the `lcsd` application when launching it (flag -r); the particular host and port number must then be passed through the "-r" flag to the lcs applications wishing to use that particular daemon.

# 4 Status of implementation

Some aspects of the user interfaces and runtime systems still need improvements, among which:

- Process management in the execution interfaces deserves a richer set of process management commands extending the basic start/call/kill set, allowing for instance selective kills or retrieval of informations or statistics about the currently running system of processes. Considering the very large number of processes lcs applications may generate, this is not an as simple task as one could expect (how to report status information for 1000 processes in some usable form?).

- The simulator does not currently allow the user to define new, derived, simulation commands. This should be provided in a later release, as a programming interface for a set of basic simulation commands, from which more complex commands could be built. The `glcs` viewer for the simulation interface should also allow to observe the internals of behaviors (value environment).

# A1 Summary of commands

## A1.1 User interfaces

The LCS system provides five distinct user interfaces: three execution interfaces, and two simulation interfaces. The former three are invoked by the `lcs` command, with a process option (`-n`, `-l` (default), or `-s`), these interfaces are running on three distinct run-time systems. The latter two are standalone lcs applications (`slcs` and `glcs`).

The `lcs -l`, `lcs -s`, `slcs`, and `glcs` interfaces allow the full LCS language to be used; the `lcs -n` interface is restricted to all non-behavioral constructions. All interfaces except the last make use of specific commands, distinct from declarations. These commands may be used at top-level only; the full list of commands is described in the following sections.

Command arguments classes are denoted by letters, with meanings:

- s string constant
- i non negative integer constant
- b boolean constant
- agt expression typing as a behavior

## A1.2 Execution interface commands

### Starting processes

**`start`** agt (eqv. **`lstart`** agt)

**`sstart`** agt

Starts a process, with behavior agt, running in the background, in parallel with all other currently running processes. These commands return immediately. Command **`start`** (or **`lstart`**), evaluates its argument and starts a process from that behavior, which shares its store with the top-level process. Command **`sstart`** creates a process which has, as store, a private copy of the top-level store at start time. Both commands bind identifier `it` to the value of the behavior started. On the runtime supporting only light processes, command **`sstart`** behaves like **`start`**.

**`call`** agt (eqv. **`lcall`** agt)

**`scall`** agt

**`return`**

Starts a process, with behavior agt, running in the foreground, in parallel with all other currently running processes. The call commands return when the system of processes called executes a **`return`** behavior. Command **`call`** (or **`lcall`**) starts a process that shares its store with the top-level process, command **`scall`** starts a process which has, as store, a private copy of the top-level store at start time. Both commands bind identifier `it` to the value of the behavior started. On the runtime supporting only light processes, command **`scall`** behaves like **`call`**.

### Process management

**`^C`**

Restarts the top level. In addition, backgrounds the current foreground job, if any is running.

**`kill`** () (eqv. **`^\`**)

Kills all running processes and restarts the top-level. Also bound to keyboard “quit” code (`^\`) .

**reboot** agt

Restarts the LCS virtual machine with a single process running behavior agt.

# A1.3 Simulator interface commands

Expansion requests expand a behavior into a communication tree (reusable as a behavior), and produce two results: the tree itself (bound to identifier `it`), and the sub-behavior that was about to be expanded (bound to identifier `next`, and referred to in the sequel as the "resumption"). Commands are provided for interrupting and resuming expansion, for parameterizing expansion, and for changing the current resumption in the current expansion tree.

## Expansion requests

**run** agt

**xpand** agt

**scan** agt

Behavior expansion requests. Command **run** follows a random path but forget the alternative behaviors, it binds both identifiers `it` and `next` to the resumption. Command **xpand** follows a random path keeps unexpanded alternatives in the tree, command **scan** explores all alternatives, both bind identifier `it` to the result of the expansion (made into a behavior), and identifier `next` to the resumption. The paths developed by the expansion are bounded to the value specified by command **steps** below.

**try** agt

Generic behavior expansion request. Defaults to "**run** agt" in "run" expansion mode, to "**xpand** agt" in "xpand" expansion mode, or to "**scan** agt" in "scan" expansion mode. The expansion mode is set by command **mode** below.

**more** ()

Resumes the last interrupted expansion at the current resumption, according to the current expansion mode. Values it and next are assumed unchanged since the last interrupted expansion. Behavior `it` is physically updated (renewed in run mode) and agent `next` is bound to the new resumption.

**mode** (s|())

Changes the expansion mode for commands **try** and **more** to that indicated by its string argument (s must be one of "xpand", "run" or "scan"); prints the current mode if argument is ().

## Change and Query of expansion parameters

**steps** (i|())

Sets as i the bound on the number of commitments and/or rendez-vous to be performed when expanding behaviors. When the bound is reached, expansion is interrupted; it may be resumed later with command **more**. Prints the currently set number of steps is argument is ().

**commits** (b|())

Cancels (if b is `false`), or requests (if b is `true`) the printing of commitments and hidden communications when expanding behaviors. When their printing is enabled, commitments and local communications are printed as periods. Prints current setting if argument is ().

## Resumption moving commands

**right** i

**left** i

Command “**right** `i`” (rest. “**left** `i`”) sets as current resumption the $i^{th}$ sub-tree on the right (resp. left) from the current resumption. Both commands fail if no behavior is pending at that place; both rebind identifier `next` in case of success.

**rmost** ()

**lmost** ()

Command “**most** ()” (resp. “**lmost** ()”) sets as current resumption the rightmost (resp. leftmost) sub-tree in the expansion tree. Both rebind identifier `next` in case of success.

**tag** (s|())

Tags the current resumption with string s. If argument is (), prints the tag assigned with the current resumption, if any.

**jump** s

Makes the first found (from leftmost) sub-behavior tagged with string `s` the current resumption. Rebinds `next` if successful.

**search** ()

Traverses the tree of pending behaviors rightward from the current resumption, and makes the first sub-behavior found which is not a deadlock the current resumption. Rebinds `next` if successful.

**dead** ()

Indicates if the current resumption is a deadlock or not.

# A2 Error and warning messages

Here is the complete list of error messages. Most of them are self-explanatory and are not further commented.

## A2.1 User interfaces messages

User interface messages are printed on the current standard output stream of the top-level (that returned by std_output()).

### all passes

lcs error <*some diagnostic*>, please report it

You just hit a bug guard in the user interface. Please report such occurrences.

### scanner and parser

in <*some context*>; expected <*some token*>; found <*another token*>

A typical syntax error.

in <*some context*>; <*some constraint violation*>

Another one.

unbound structure <*path*>

<*some specific error message*>

E.g. illegal integer constant.

warning: end-of-file found in comment

warning: imperative type variable promoted by parser

The parser read some imperative type variable of form 'na (in which n is some integer), or'_a (for compatibility with SML), which it transformed into'a.

warning: imperative beh variable promoted by parser

The parser read some imperative behavior type variable of form _na (in which n is some integer), which it transformed into _a.

warning: source and destination label sets not disjoint in renaming

warning: signatures not supported yet, sorry

A signature declaration was attempted.

warning: skipped signature constraint for <token>

A signature constraint was found following a structure identifier. The constraint has been skipped.

### typechecker

unbound <*structure|variable|type identifier*> <*ident*>

in <some context> <*some constraint violation*>

equality type required .....

types mismatch ...

types mismatch for overloaded <*identifier*> ...

overloaded versions of <*identifier*> must have monotypes ...

overloaded version of <*identifier*> has polytype after open <*path*> ...

unresolved type for overloaded symbol <*identifier*> ...

warning : type constraint annotated

Weakness annotations were added to a type constraint.

`warning : type constraint strengthened`

A type constraint was strengthened.

`warning : redeclaring <identifier>`

A new version of an overloaded symbol is declared.

`warning : binding contains no variables`

### compiler

`compiler overflow; <which one>`

No code can be generated for the current declaration. This is not likely to happen. It if ever happened, the declaration should just be broken into smaller pieces.

`up applied to non exhaustive pattern`

Issued when nonexhaustive lazy patterns are disabled and one is found (see the *Language Reference* manual).

`warning : redundant pattern(s) in match`

A redundant pattern has been detected in a match. Note than any universal handler in an exception handler match causes this message to be printed; the reason is that universal exception handlers are redundant with the pattern for the default handler introduced by the compiler.

`warning : <match|binding> not exhaustive`

`warning : up applied to non exhaustive pattern`

Issued as a warning when nonexhaustive lazy patterns are enabled.

`warning : <some construction> not supported, sorry`

Some versions of the simulator may not support some language constructions yet.

### evaluation

`uncaught exception -- EXCEPTION_VALUE`

Printed when an uncaught exception reaches top-level.

## A2.2 Runtime system messages

Runtime error messages are printed on the standard output stream (`std_out`).

`fatal error: heap space exhausted`

No more heap space is available. Will occur when the host operating system cannot provide more virtual memory for extending the heap.

`fatal error: stack space exhausted`

No more stack space is available. Will occur when the host operating system cannot provide more virtual memory for extending the stack.

`fatal error: malloc area overflow`

No more malloc space is available (e.g. for TCP/IP buffers). Will occur when the host operating system cannot provide more virtual memory for extending the malloc area.

`fatal error: object too big`

Object to be allocated is too large, it would overflow some fields in cell headers. Not likely to happen.

`fatal error: unknown opcode <code>, please report it`

A non implemented instruction was fetched from the code (should not normally happen).

`fatal error: cannot find file: <file>`

The bytecode file <*file*> cannot be found.

`fatal error: cannot find runtime for core: <file>`

The runtime able to run bytecode file <*file*> cannot be found.

`fatal error: incompatible core file: <file>`

The bytecode file <*file*> cannot be run on that version of runtimes.

`fatal error: cannot allocate enough memory for core: <file>`

The bytecode file <*file*> cannot be loaded (the operating system cannot provide enough space to load it).

`fatal error: cannot open any descriptor`

No more file descriptors can be open when entering lcs. This should not normally happen.

`fatal error: uncaught exception reached machine level`

An uncaught exception reached machine level in the top-level process (should not normally happen).

`fatal error: machine error, please report it`

Result of some unexpected Unix signals received by the runtime (should not normally happen).

`warning: can't find DISPLAY variable, using default: :0`

Environment variable DISPLAY is checked when opening windows.

`warning: cannot find TERM variable in environment`

Environment variable TERM is checked when opening windows and entering lcs.

`warning: cannot find terminfo for <term>`

The unix terminfo database is searched when opening windows and entering lcs.

`warning: core file <file> has older version than expected`

Bytecode file has an older version than the current runtime, but it can be run.

`warning: job returned`

A return was executed, with no call waiting for it; it was handled as stop.

`warning: uncaught event signalled (X)`

An event (X) was signalled by some process (possibly resulting from some untrapped exception raised from within the process), and there are no catchers for it. The process was aborted. As a debugging aid, the event raised is printed, though it may be out of scope at top-level.