The Logix System User Manual
Version 2.0

William Silverman, Michael Hirsch,
Avshalom Houri and *Ehud Shapiro*

Department of Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

Preliminary Version

Last revision March, 1993

Technical Report CS-21

Contents

1 Introduction

Concurrent Prolog [5] is a logic programming language designed for concurrent programming and parallel execution. It is a process oriented language, which embodies dataflow synchronization and guarded-command indeterminacy as the basic control mechanisms. Flat Concurrent Prolog (FCP) [3], was identified as a practical and efficient subset of Concurrent Prolog, which supports most of its programming techniques. The reader is referred to [6] for a survey of the language and its programming techniques, and for an extensive bibliography. Further evolution of the language resulted in the development of FCP(:,?), which is the language currently supported by the Logix System. The reader is referred to [9] for language definition, and to [10] for program examples and techniques.

The purpose of this manual is to serve as a concise specification for Logix users, as well as to document the various aspects of the system which are of interest to the research community. It is not necessarily the best starting point for someone not familiar with concurrent logic programming.

Logix is a programming environment written in Flat Concurrent Prolog (FCP) which allows the development and testing of concurrent logic programs. Logix includes a compiler, which compiles FCP programs into an FCP abstract machine instruction set, and an emulator, written in C, which emulates these instructions.

Logix is organized around three types of objects: computations, modules, and services. A computation is the unit of execution, control and debugging in Logix; a module is the unit of compilation; a service provides access to facilities and capabilities of the underlying system, for example the screen and the file system. Several computations may proceed in parallel under Logix. A computation may span many services; several concurrent computations may execute code in the same module.

A computation occurs in time. It starts with an initial process, typically specified interactively by the user. At each point in time a computation consists of a multiset of processes; it terminates when this becomes empty. A computation proceeds by processes performing actions. A process may terminate, fork, or change state. While doing so it may assign values to variables by unification, thus achieving the effect of communication in general, and output construction in particular. On a multiprocessor Logix, a computation could span several processors.

A module is a unit of compilation. It consists of a set of procedures, preceded by an optional declaration. A procedure may call and be called by procedures in other modules using remote procedure calls. Modules are loaded automatically when called and compiled automatically as needed. The active module is a service, which may be either a server or a monitor.

Remote procedure calls between services are implemented via the stream communication capabilities of FCP. In a parallel implementation of Logix different services may reside in different processors, and remote procedure calls may cross processor boundaries [7].

A Logix user is provided with an initial set of services, which are accessed via remote procedure calls. These include a screen service, a file service, and a compile service, among others. In addition, a server or monitor is created for each active module, to serve remote procedure calls directed to it.

The rest of the manual is organized as follows. Section 3 contains an annotated session with Logix, demonstrating some of its capabilities. Section 4 explains the user interface. Section 5 the notions of computation and event, and the commands for manipulating and inspecting computations. Section 6 introduces the notion of modules and explains their compilation and manipulation. Section 7 reviews some of the system services provided by Logix. Section 8 explains the debugger.Section 10 lists some utilities, and

2 Getting Started with Logix

The basic cycle of program development under Logix is similar to other systems:

Compose a program
Repeat
  Test the program
  Modify the program
Until the program works to your satisfaction.

For example, assume that we want to compose a program for reversing a list. Initially we compose the following program, using a text editor, and save it in the file *rev.cp*.

```
reverse([X—Xs],Ys) ←
    reverse(Xs,Zs), append(Zs,X,Ys).
reverse([ ],Ys) :- true : Ys = [ ].

append([X—Xs],Ys,Zs) :- true : Zs = [X—Zs']) ←
    append(Xs,Ys,Zs').
append([ ],Ys,Zs) :- true : Zs = Ys.
```

We start Logix, which prompts us with '@'. To test *append/3*, we issue to Logix the command:

```
@rev#append([1,2,3],[4,5],L1)
```

The module *rev* is compiled automatically, and following a successful compilation the computation of the goal starts:

⟨*1*⟩ *started*
⟨*1*⟩ *terminated*
@

When computation ⟨*1*⟩ ends, and Logix is idle, it prompts again. Variable bindings are kept through the interaction with Logix. To see the result of the computation, i.e. the value of *L1*, we use the command:

@L1↑
L1 = [*1, 2, 3, 4, 5*]

A variable in the goal can be marked with ↑, to indicate that its value should be printed when available, as in:

@rev#append([a,b,c—Xs1],[d,e],L2↑)
⟨*2*⟩ *started*
L2/1 = *a*
L2/2 = *b*
L2/3 = *c*

which prints the stream of elements of *L2* as they are produced. The values of goal variables can be provided incrementally, for example:

@Xs1=[f,g,h—Xs2]
L2/4 = f
L2/5 = g
L2/6 = h

@Xs2=[ ]
L2/7 = d
L2/8 = e
L2/ = [ ]
⟨*2*⟩ *terminated*
@

We turn to test *reverse*. For the goal:

@rev#reverse([a,b,c],L3↑)

we get the response:

⟨*3*⟩ *started*
⟨*3*⟩ *failed(rev#append(c, b, _))*
⟨*3*⟩ *suspended*

which indicates that the specified *append* goal has failed. The goal has failed since no clause unifies with it. *append* expects a list as its first argument. A look at the code shows that *reverse* called *append* with the wrong arguments. To fix the *rev* module, we give the command:

@edit(rev)

which indicates to Logix that we plan to edit this file, suspends Logix, and brings us back to Unix, where we enter our favorite text editor and edit *rev.cp*. We fix the recursive clause of *reverse* to be:

reverse([X—Xs],Ys) ←
    reverse(Xs,Zs), append(Zs,[X],Ys).

return to Logix, and retry the program:

@rev#reverse([a,b,c—Xs3],L3↑)
⟨*4*⟩ *started*

we do not get any output, since *reverse* starts producing its output only after the input list is completed. However, we can inspect the current state of the computation, i.e. the resolvent:

@resolvent
⟨*4*⟩*suspended*
⟨*4*⟩*goal-1 = reverse(_, _)*
⟨*4*⟩*goal-2 = append(_, [c], _)*
⟨*4*⟩*goal-3 = append(_, [b], _)*
⟨*4*⟩*goal-4 = append(_, [a], _)*
@

which shows that there are three *append* processes and one *reverse* process suspended. If we wish to observe parts of the computation in greater detail, we can do that. The command

@debug(1)

calls the Debug Service to debug process number 1. We can supply it with further input:

@Xs3=[d,e—Xs4]

Inspecting the resolvent causes a computation to suspend. The command

@resume

resumes it, and initiates the following debugging session:

⟨*4*⟩ *resumed*
⟨*4*⟩ *Debug Reduction Started*
⟨*4*⟩ $reverse([d,\ e — \_], \_) \leftarrow ?$
quary → *trace*

The "trace" directive to the Debug Service requests it to print the reducing clause for each goal reduced.

⟨*4*⟩ $reverse([d,\ e — \_], \_) \leftarrow$
$\quad reverse([e — \_], \_),\ append(\_, [d], \_).$
⟨*4*⟩ $reverse([e — \_], \_) \leftarrow$
$\quad reverse(\_, \_),\ append(\_, [e], \_).$

The computation is blocked. We can close the input stream now:

@debug → $Xs4 = [\ ]$
$L3/1 = e$
⟨*4*⟩ $reserve([\ ], [\ ]).$
⟨*4*⟩ $append([\ ], [e], [e]).$
⟨*4*⟩ $append([e], [d], [e — \_]) \leftarrow$
$\quad append([\ ], [d], \_).$
$L3/2 = d$
⟨*4*⟩ $append([\ ], [d], [d]).$
$L3/3 = c$
⟨*4*⟩ *Debug Reduction Terminated*
$L3/4 = b$
$L3/5 = a$
⟨*4*⟩ *terminated*
$L3/ = [\ ]$
@

3 The User Interface

3.1 Interacting with Logix

Commands are typed to a command interpreter. A prompt '@' appears whenever the system becomes idle. A command may be typed when the system is not idle — a prompt is prefixed to the echoed command to indicate that the system is responding.

Several commands may be entered on a single line, separated by semi-colons. To continue a command over several lines, end each line except the last with "%".

There are several ways to exit from Logix:

⟨control⟩ Z
: Suspend Logix and return to the Unix shell.

⟨control⟩ C
: Abort Logix.

⟨control⟩ G
: Graceful shut-down. Deadlocked processes appear in the post-mortem dump, and some useful statistics are printed.

⟨control⟩
: Unconditional termination. All active and suspended processes appear in the post-mortem dump.

3.2 The binding environment

The scope of symbolic variable names entered interactively (or input from a file — see the input command in Section 7.5) is an entire session. In other words, a variable name used more than once, even in different commands, represents the same term. An exception is the anonymous variable name "_", which is unbound (and unique) for each use. Another exception is when a variable is annotated with ↑ within a command; its old value is forgotten, and its new value is displayed when it is instantiated.

The following commands relate to the binding environment:

X↑	Display the current value of variable X.
↑	Display the current values of all variables used so far.
unbind(X)	Unbind variable X.
unbind	Discard all variable bindings.

3.3 Line editing

Logix supports many of the line-editing features of *tcsh* shell under Unix. They are documented in Appendix 3.

3.4 File input

The command:

input(Name)

reads and echos command lines from the file *Name*. Following each line the Logix System becomes idle before the next line is read. The *input* command may appear within a command file.

The command file *.logix* of the working directory is automatically input when the Logix System starts.

4 Computations

A computation is the basic unit of execution, control, and debugging in Logix. A computation can be started, suspended, resumed, aborted, and inspected. Furthermore, pieces of the computation can be debugged: they can be stepped through, can be set with breakpoints, etc.

Each computation is numbered; its number is displayed in angled brackets, e.g. $\langle 3 \rangle$.

4.1 Events

A computation can go through a sequence of events, which appear in its events stream. Possible events include:

started
: The computation has started. The first event in a computation.

suspended
: The computation has been suspended.

resumed
: The suspended computation has been resumed.

aborted
: The computation has been aborted. A terminal event.

terminated
: The computation ended successfully. A terminal event.

failed(Service#Goal,ErrorCode)
: *Goal* in *Service* has failed, due to error *ErrorCode*. The failed process is extracted from the computation, and the computation is suspended. The computation (without the failed process) can then be resumed or aborted at the user's discretion.

4.2 Computation management

The following commands relate to computation management. Their description uses the following conventions. Output arguments of commands are annotated with a postfix $\uparrow$. Arguments are assumed to be omitted from the right. If an output argument is omitted its value is displayed on the screen.

The *No* argument refers to computation $\langle No \rangle$. If a *No* input argument is omitted the current (most recently referenced) computation is assumed.

The *Service* argument refers to a service name. If it is omitted, the current (most recently referenced) service name is assumed.

The constant *all* can be given instead of a number argument to affect all computations, or all processes, as appropriate.

The full version of the commands is used mainly from inside computations via the shell server, as explained in Section 7 on system services; most output and default arguments are omitted in interactive use.

Invocation:

start(Service#Goal,No↑,Events↑,Ok↑)
: Start a computation of *Goal* by *Service*, unify *No* with its number, and unify *Events* with its stream of events.

Service#Goal
: Same as above, where ⟨*No*⟩ and *Events* are displayed on the screen.

#Goal
: Same as above, where the current *Service* is assumed.

Inspection:

state(No,Goal↑,Events↑,Ok↑)
: Unify *Goal* with the initial goal of computation ⟨*No*⟩, which includes values assigned so far to its output variables.
: Unify *Events* with the list of events of the computation so far.

events(No,Events↑,Ok↑)
: Unify *Events* with the stream of events of the computation.

resolvent(No,Resolvent↑,Ok↑)
: Suspend computation ⟨*No*⟩, and unify *Resolvent* with a snapshot of its resolvent.

Manipulation:

suspend(No,Ok↑)
: Suspend computation ⟨*No*⟩.

resume(No,Ok↑)
: Resume suspended computation ⟨*No*⟩.

abort(No,Ok↑)
: Abort computation ⟨*No*⟩.
: "suspend", "resume", and "abort" without arguments effect the current computation.

extract(ProcessNo,No,(Module#Goal)↑,Ok↑)
: Extract *Goal* in *Module* with serial number *ProcessNo* from computation ⟨*No*⟩. Process numbers can be determined by inspecting the resolvent.

add(Service#Goal,No,Ok↑)
: Add the computation of *Goal* by *Service* to computation ⟨*No*⟩.

Debugging:

debug(Module#Goal,No↑,Events↑,Ok↑)
: Debug a computation of *Goal* in *Module* and unify *No* with its number.

debug(ProcessNo,No,Ok↑)
: Debug process number *ProcessNo* in computation ⟨*No*⟩.
: *debug*(*ProcessNo*) debugs *ProcessNo* in the current computation.

Performance:

time(Service#Goal,Timing↑,No↑,Events↑,Ok↑)

Unify *Timing* with timing information about the computation of *Goal* in *Service*.

Interactive computations cannot be timed with this command.

rpc(Service#Goal,Rpc↑,No↑,Events↑,Ok↑)

Provides a crude measure of the parallelism in a computation, which is the average number of process reductions in one emulator cycle.

The measure is crude since their could be dependencies between reductions in the same cycle, which may prevent their true parallel execution.

profile(Module#Goal,Profile↑,No↑,Events↑,Ok↑)

Unify Profile with profiling information about the computation of *Goal* in *Module*.

Environment:

The current computation and service can be queried and set.

computation(New,Old↑)

⟨*Old*⟩ used to be the current computation.

Set the current computation to be ⟨*New*⟩.

(The current computation can be found with the command "computation(X,X)", or just "computation").

service(New,Old↑)

Old used to be the current service.

Set the current service to be *New*.

(The current service can be found with the command "service(X,X)" or just "service".)

5 Modules and Their Compilation

A module is the basic unit of compilation in Logix. A module is a set of procedures. As a convention the source of a module *foo* resides in the file *foo.cp*. When compiled, the binary form of that module is saved in the file *foo.bin*.

A module source text may begin with an optional module declaration, whose format is explained below. The procedures in a module can be in any order. Clauses of a procedure should be grouped together. A module is self-contained in the sense that it defines all procedures that it calls, with the exception of remote procedure calls and primitive procedures. Primitive procedures were listed in Section 2.5.

5.1 Remote procedure calls

A remote procedure call is a goal in the body of a clause of the form

ServiceName#Goal

For example, to reduce the process *isort(In,Out)* defined in module *sort*, use the call:

sort#isort(In,Out).

This call indicates that *isort(In,Out)* is executed with the procedure defined in module *sort*. Undefined and multiply defined procedures in a module cause compilation errors. Undefined remote procedure calls cause runtime errors.

5.2 Module management

The following commands relate to module management.

compile(Module,Options,OK↑)
: Invalidate the current binary of *Module*, if it exists, and (re)compile it.

edit(Modules)
: *Modules* can be a service name or a list of service names.
Invalidate the current binaries of *Modules*, and suspend Logix.
The modules are recompiled next time they are called.

lint(Module,Options,OK↑)
: Performs various checks on *Module*.

reload(Modules)
: Reload the binaries of *Modules*.

If the *Options* argument is omitted, default options are used.

If the *Module(s)* argument is omitted, the current service is used.

Note that if a source module that was not mentioned in an *edit* command is edited, it must be compiled explicitly to keep its binary consistent with the source.

5.3 The module declaration

The compiler can be directed by a module declaration to achieve various tradeoffs between functionality, compile time, and runtime performance.

A module can begin with declarations, identified by a prefix '–':

– AttributeName(Value).

The list of attributes of module *Service* can be found by executing the remote procedure call:

Service#attributes(List).

The following are meaningful attributes:

The *export* attribute

– export(ExportList).

where *ExportList* is a list of *Name/Arity*. The export attribute indicates the procedures in a module that can be called from the outside. If *ExportList*='*all*', or the attribute is omitted, then all procedures are exported. If the attribute is present the compiler detects dead code (procedures that are not reachable from the exported procedures) and undefined exported procedures.

The *mode* attribute

– mode(Mode).

where *Mode* is one of [*interpret, interrupt, failsafe, trust*]. The default mode is *interpret*. A process executing interpret-mode code is responsive to all the commands mentioned above: it can be inspected, debugged, suspended, resumed, aborted, and if it fails it announces this and its computation is suspended. In addition, procedures in an interpret-mode module can be meta-interpreted, as explained in Section 9.

Modules can be compiled in modes other than *interpret* to increase performance, at the expense of control, as follows.

interrupt
Pros: 3 times faster in runtime than interpret.
Cons: Processes cannot be debugged or interpreted.
failsafe
Pros: 10% to 20% faster in runtime then interrupt.
Cons: A process cannot be inspected, suspended or aborted.
trust
Pros: 10% to 30% faster in runtime then interrupt.
Cons: Processes may proceed after the computation is terminated.

There is also a significant improvement in compilation time and code size for the less-controlled modes of execution.

Other attributes

The compiler adds two attributes automatically:

– path(AbsolutePathOfSource).
– date(DateOfSource).

They can be retrieved by the *attributes* call, and used to verify the validity of the binary with respect to the source. Other attributes can be included, without affecting the compiler, for example:

– name(ModuleName).
– author(AuthorName).

The attribute:

– monitor(Start).

is discussed in Appendix 5.

6 System Services

The Logix system is a collection of communicating services. The system is connected to a directory. All modules residing in the connected directory are implicitly defined services of the system. They are opened dynamically when called. If the binary of a service that is called is available then it is loaded and activated; else the module is compiled.

A *Request* for *Service* is a remote procedure call.

Service#Request.

To call requests of a service *ServiceName* anonymously, use the call:

ServiceName#open_context(C).

Call each request:

context(C)#Request.

The following are services provided by Logix.

6.1 Shell

The shell service accepts commands that can be given interactively. These commands can be called:

shell#Request
: Where *Request* is any command that can be given interactively, except binding environment commands.

The shell understands several other commands:

X=Y	Unify *X* with *Y*.
X:=Y	Evaluate the arithmetic expression *Y*, unifying the result with *X*.
hi	The Logix system user-friendliness facility. The shell responds with *hello*.

The shell automatically submits each command to the service *user_macros* to be expanded before interpretation. See Appendix 4 for the format of requests to *user_macros*. Primitive procedure calls which are not expanded by *user-macros* may be called interactively.

6.2 Screen

The screen service can serve the following requests:

screen#display(Term,Options)
: Display *Term* on the screen, according to *Options*.

screen#display_stream(TermsStream,Options)
: Display *TermsStream*, one term per line as they become available, according to *Options*.

screen#ask(Term,Answer↑,Options)
: Display the query *Term*, according to *Options*, and read *Answer*.

When the *Options* argument is omitted, the default options are assumed.

Options is a list of terms of the form *Control*(*Controls*) or *Attribute*(*Value*).

Attributes are in [*type,read,depth,length*].

Default values of attributes can be inspected and modified with the request:

screen#option(Attribute,New,Old↑)

Modify default option *Attribute* from *Old* to *New*.

Control options are used to control the time and format of display. *Controls* are:

known(X)

Wait until *X* is assigned a non-variable value before printing.

close(L,R)

After printing, unify *L* and *R*.

prefix(T)

Print the term *T* before (each) *Term*, separated by one space.

list

Treat (each) Term as a list of terms to be printed without separation — no automatic final line-feed is added.

put(FileName)

Open the named file for writing and write (each) *Term* to the file.

append(FileName)

Append (each) *Term* to the file.

The *put* and *append* options are not relevant to queries.

Values of option attributes are described below. The description format is:

Attribute

Value

Meaning.

The *type* attribute determines the mode of display of variables.

type

freeze

Variables are printed as "_",
read-only variables as "_?".

ground

The term is printed after it is ground.

namevars

The term is frozen and variables are printed by name or, if anonymous, numbered as *_Xnnn* or *_Xnnn?*. Identical variables have the same name.

(default is "freeze").

The *read* attribute is relevant to queries only. It specifies what input is read by the query, and what is returned to *Answer*.

read

char

One character is read from input (the keyboard) and converted to a string.

string

One line is read from input, returned as a string ending with the line-feed character ($ascii\langle lf\rangle$).

chars

One line is read from input, returned as a difference list of *ascii* characters (small integers) followed by *ascii* characters $\langle space\rangle$,".",$\langle lf\rangle$.

(default is char).

Terms are printed only to a limited depth, after which sub-terms are represented by ... . Lists are inspected only to a given length. Only that many leading elements are printed, along with an indication of omitted elements if the whole list was examined. The attributes which control this are:

depth

integer (default is 8)

depth of nesting of tuples/lists.

length

integer (default is 20)

maximum length of list examined.

6.3 File

The file service interfaces to the Unix file system. It provides the basic file utilities needed by the user.

The service is connected to a Unix directory. Initially the directory is the working directory of the user when Logix is started.

The file server can serve the following requests:

file#get_file(Name,Contents,Options,Ok↑)

Read the file *Name* and return the string *Contents*.
Ok becomes "true" or "not_found".
Options is a list; it may be [ ] or contain "chars" or "string".

file#put_file(Name,Contents,Options,Ok↑)

Write the *Contents* into the file *Name*.
Ok becomes "true" or "write_error" when writing completes.
Contents can be a constant, or a stream of constants.
Options is a list; it may be [ ] or contain one of "chars", "append" or "string".

6.4 Other Services

Many other services are supplied by the Logix System. For details see [8].

The Logix System supports hierarchical static scoping of modules. The hierarchy service operates on sub-trees of the hierarchy.

The Logix System may be run as a virtual subsystem, in the same processor as its parent system, or in a separate processor. The service logix_system provides a means of spawning such sub-systems.

The transform service supports alternate languages through source-to-source transformation of modules.

The computation_utils service simplifies the spawning and control of sub-computations.

The array service provides efficient manipulation of vectors of re-assignable terms.

7 The Debug Service

The Debug Service is a meta-interpreter with a computation control mechanism, which prints goals as they are reduced, and prompts for commands to control the computation.

Commands to invoke the Debug Service appear in Section 5.2. The Debug Service provides extensive help; at any prompt the command "help" may be entered for a menu of available commands.

The Debug Service has two major modes of operation, *command* mode, and *directive* mode. In command mode (when the prompt is "*@debug*$\rightarrow$") the computation is stable, and any valid command may be entered. Any command which is not understand in command mode is delegated to Logix. In this manner commands to the Debug Service may be intermixed with commands to the system shell. The user is in directive mode (prompt is "*query*$\rightarrow$") when the system is querying the user about the validity of a reduction. This occurs only at a breakpoint. An initial breakpoint is set before the first reduction; afterwards the user must set and remove breakpoints manually. A breakpoint also occurs whenever the reduction of a goal fails, or when an invalid module name is specified.

In directive mode, only valid directives may be entered, although one of these directives is to switch to command mode. If the user switches to command mode without replying to the query break prompt, the user is left in temporary command mode. In this state, no further breakpoints occur until the original query is answered. Some responses are valid for both modes, but their effects are usually different. In general, a command applies to every active process, while a directive refers to a specific query for a single process at a breakpoint.

Breakpoints may be set prior to the reduction (pre-reduction breakpoint) or after the reduction (post-reduction breakpoint) of a specific goal. A breakpoint may be of type print or query. A print breakpoint displays the current Goal (and Body if this is a post reduction breakpoint) and continues executing, while a query breakpoint requests information from the terminal before continuing. A query breakpoint places the user in directive mode when the breakpoint is encountered.

The depth of the computation tree may be bounded, although initially it is unbounded. Any goal which exceeds it's depth bound is suspended until the bound is increased (via the depth command). The depth bound may be set to an arbitrary number or infinite (-1). Goals which have exceeded their depth bound may be viewed via the disabled command. Goals which are enabled, are currently running. The resolvent is the list of all goals in the computation.

All commands with multiple parameters have defaults which are explained in the help subsystem.

Command Summary:

abort	Abort the computation and terminate the Debug Service.
help	Print menu of valid commands.
suspend	Suspend the current computation.
resume	Resume the current computation.
depth(N)	Increase the computation depth bound by N for all processes.
resolvent	Obtain the current set of processes in the computation.
enabled	Obtain the set of processes which are active currently.
disabled	Obtain the set of processes which are held on depth bound.
break	Enable a breakpoint in all active processes.
remove	Disable a breakpoint in all active processes.
clear	Remove all breakpoints in all active processes.
query	Return from temporary command mode to directive mode.
help(Command)	Print information about *Command*.

Directive Summary:

⟨CR⟩, y, yes	Continue processing this goal.
depth	Set a depth bound for this process.
wait(Variable)	Suspend this goal until *Variable* is instantiated.
break	Set a break point for this process and its descendants.
trace	Print a trace for this process and its descendants.
remove	Remove a break point for this process and its descendants.
clear	Remove all beak points for this process and its descendants.
trace	Print each reduction for this process and its descendants.
derive	Enter all direct descendants into the break point list.
list_breaks	List all active break points for this process.
depth(NewDepth)	Change depth for this goal to *NewDepth*.
data(Data)	Further instantiate the head of a clause.
execute	Let this goal continue as an independant subcomputation.
debug	Change to temporary command mode.
help	Print a menu of valid directives.
help(Directive)	Print information about *Directive*.

8 Utilities

Logix provides several utilities, which can be accessed via remote procedure calls.

8.1 Stream

This service provides access to the fast multi-way merge and to the indexed distribute. It exports the procedures *merger/2*, *distributor/2* and *hash_table/1*.

stream#merger(In,Out)
: Each term which appears on the *In* stream is copied to the *Out* stream, except for the term *merge(OtherIn)*. In that case terms from the *OtherIn* stream are also copied to the *Out* stream. The merger remains open until all merged in-streams are closed (end-of-stream is reached) — then the *Out* stream is closed.

stream#distributor(In,Tuple)
: *Tuple* is an *n*-tuple or a list of *n* output streams.
In is a stream of requests of the form:
I # Term
For each such request, *Term* is copied to the *I*'th stream (where *I* is an integer between 1 and *n*). When *In* is closed, all of the output streams are closed.

stream#hash_table(In)
: *In* is a stream of requests to a hash-table manager.
The requests are:

lookup(Key,NewValue,Value↑,Reply↑)
: The item identified by *Key* is found (created), its old *Value* returned, and its new value set.
Reply is "old" or "new".

member(Key,Value↑,OK↑)
: The item identified by *Key* is found, and its old value returned.
OK is "true" if the item is in the table and "false" otherwise.

delete(Key,Value↑,OK↑)
: The item identified by *Key* is deleted from the talbe and its value returned.
OK is "true" if the item is deleted, and it is "false" if if no item identified by key was found.

send(Key,Term,OK↑)
: The item identified by *Key* is the variable tail of a stream.
Term is added to the stream and the new tail is remembered.
OK is "true" if *Term* was send, and is "false" otherwise.

entries(List↑)
: *List* is a list of all the items in the hash-table in the form *entry(Key, Value)*.
Key may be a string or an integer. Items which are streams must be closed explicitly (e.g. using the entries *List*).

8.2 Utils

This module is a general utilities package. It exports procedures — *chars_to_lines/2*, *append_strings/2*, *evaluate/2*, *freeze_term/3*, *ground/2*, *ground_stream/2*, *integer_to_dlist/3*.

chars_to_lines(Chars,Lines)
: Splits the ascii stream *Chars* into strings which are returned in the list *Lines*. A new string is started at each ascii ⟨*lf*⟩ or ⟨*cr*⟩ in the character stream.

append_strings(Strings,String)
: *Strings* is a list of strings, which are concatenated and returned in *String*.

evaluate(Expression,Result)
: Evaluate the arithmetic term *Expression*, returning integer *Result*.

ground(X,Y)
: Wait until X is ground, and then unify X and Y.

ground_stream(X,Y)
: Treat X as a stream, doing a "ground" for each element in turn, unifying the element with the corresponding element in Y.

integer_to_dlist(I,List,Tail)
: Convert the integer I to a list of ascii characters in *List*, ending with *Tail*.

9 Miscellaneous

When some process becomes an orphan (independent of all active processes) post-mortem information appears following normal termination, beginning with the diagnostic:

Suspended Procedures:
scanning the heap ...
scanned the heap, *ppp* processes found, *mmm* processes missing!

The processes that were found are printed on subsequent lines. The missing processes were orphaned and were subsequently garbage collected.

The system garbage collector, a part of the emulator, announces its activity by temporarily suspending all other activity, writing (GC) to the screen and when it has finished, erasing the message. This may have a peculiar looking effect if it happens near the end of a line which is being printed.

The overall capacity of the heap can be exceeded. This is diagnosed, and the system terminates.

To run with a larger heap, you may enter the command:

logix –hnumber

where number is the size of storage needed in kilo-bytes. The maximum possible size of storage is a Unix system parameter.

Variables in a call to := can be bound at runtime to integers, but not to general arithmetic expressions. To evaluate arithmetic expressions generated at runtime use the utility *evaluate*.

Appendix 1: Primitive Procedures

The following are predefined FCP procedures.

Arithmetic:

X:=E	Evaluate the arithmetic expression E and unify the result with X. Provided operators: $+$, $-$, $*$, $/$, *mod*, *max*, *min*, *abs*, *integer*, *real*,*round* and bitwise *and*, *or*, *not*.

Term inspection:

arity(T,A)	The arity of the compound term T is unified with A (the arity of a list is 2 and of a constant is 0).
arg(N,T,A)	The N'th argument of a compound term T is unified with A (for a list $[X — Xs]$, X is its first argument and Xs is its second).
length(S,L)	The length of the string or list S is unified with L.
nth_char(S,N,C)	The N'th character of string S is unified with C.
string_hash(S,H)	The integer-valued hash-code of string S is unified with H.

Term creation:

make_tuple(N,T)	T is a new tuple of arity N.
copy_skeleton(T,T1)	*T1* is a copy of the top level of the compound term T.

Type conversion:

string_to_dlist(String,ListOfAscii,E)	Convert characters of string *String* into an incomplete list of integers with tail E, and unify it with *List*.
list_to_string(ListOfAscii,String)	Convert a list of integers to a string.
tuple_to_dlist(Tuple,List,E)	Convert a tuple to a difference-list *List* with tail E.
list_to_tuple(List,Tuple)	Convert a list to a tuple.
convert_to_integer(Value,Integer)	Convert a string or a number to an integer.
convert_to_real(Value,Real)	Convert a string or a number to a real.
convert_to_string(Value,String)	Convert a string or a number to a string.

A numeric value is converted to/from its visual representation as a string.
Note: *arity*, *arg*, and *copy_skeleton* are polymorphic: T may be a list or a tuple.

Others:

freeze(Term, FrozenTerm, FrozenVariableList)

FrozenTerm is a string, whose interior represents *Term* with each writable variable replaced by a frozen writable variable and each read-only variable replaced by a frozen read-only variable. The connection between writable variables and read-only variables is preserved in their corresponding frozen variables.

FrozenVariableList contains an entry for each distinct frozen variable in *FrozenTerm*. A variable in *FrozenVariableList* is writable if some corresponding frozen variable is writable.

melt(FrozenTerm,MeltedTerm,MeltedVariableList)

MeltedTerm is a renaming of *Term*. All variables in *MeltedTerm* are local to it, except that each distinct variable in *MeltedTerm* is in *MeltedVariableList* in the same order as the corresponding variables (terms) in *Term* are in *FrozenVariableList*. Thus, *melt/3* can be used like the familiar "melt_new".
MeltedVariableList is a list of writable variables corresponding to distinct frozen variables in *FrozenTerm*. If it is unified with *FrozenVariableList*, *MeltedTerm* becomes identical to *Term*.

9.1 Arithmetic expressions

Integer and real numbers may be mixed freely in arithmetic expressions, except that the arguments of the mod function must both be integers. The result of an operation is real if any of its arguments is real, and integer if all arguments are integer. Division truncates its result if both arguments are integer.

The function $real(I)$ may be used to convert an integer expression to a real value. The functions $integer(R)$ and $round(R)$ convert their argument to integer, either truncated toward zero or rounded away from zero; both fail in case of overflow of the integer range.

Guard arithmetic expressions may not include the operators *min*, *max*, *abs*, *round*. The expression in the primitive procedure *:=/2* may include arbitrary functions declared by library or user procedures. For example the function $min(X, Y)$ is supported by the library procedure *min/3*, declared:

min(X,Y,X) $\leftarrow$ X $\leq$ Y — true.
min(X,Y,Y) $\leftarrow$ Y $\leq$ X — true.

Appendix 2: Guard Test Predicates

Below is the list of guard test predicates of FCP. Logix implements several additional guard predicates for system interface purposes.

Unification:

X = Y	X and Y unify.
X ≠ Y	X and Y fail to unify.

Type checking:

integer(X)	X is an integer.
real(X)	X is a real number.
number(X)	X is a number (either an integer or a real).
string(X)	X is a string.
constant(X)	X is a constant (either a number or a string or nil).
tuple(X)	X is a tuple.
list(X)	X is a list.
compound(X)	X is compound (either a tuple or a list).

Arithmetic:

X =:= Y	Arithmetic comparison of numeric expressions.
X > Y	ditto.
X >= Y	ditto.
X < Y	ditto.
X =< Y	ditto.

Term comparison:

X @< Y	X is less then Y according to the canonical ordering: numbers @< strings @< [ ] @< lists @< tuples. Variables are incomparable, so for two different variables X and Y tests X @< Y and $X \neq Y$ suspend. Strings are ordered lexically, tuples are ordered by arity. Two tuples of the same arity are ordered by their elements, compared pair-wise, left-to-right. Two lists are ordered first by their CARs and then by their CDRs.

Meta-logical:

unknown(X)	the value of X is unknown.
known(X)	the value of X is known.

Control:

true	succeeds.
otherwise	succeeds when all textually previous clauses fail.

Note: A test predicate suspends if it does not succeed but may succeed later.

Appendix 3: List of Operators

Operator	Position	Meaning	Priority	Associative
$\leftarrow$	infix	if	1200	no
—	infix	commit/cons	1100	right
;	infix		1100	right
,	infix	argument/goal separator	1000	right
=	infix	unify	800	no
=?=	infix	recursive equality	800	no
$\neq$	infix	recursive inequality	800	no
@<	infix	canonical ordering	800	no
@	infix		720	left
#	infix,prefix		710	right
:=	infix	arithmetic assignment	700	no
=:=	infix	arithmetic equality	700	no
==	infix	top level equality	700	no
=	infix	top level inequality	700	no
<	infix	less	700	no
>	infix	greater	700	no
$\leq$	infix	less or equal	700	no
$\geq$	infix	greater or equal	700	no
+	infix,prefix	plus	500,220	left
-	infix,prefix	minus	500,220	no
$\uparrow$	suffix		500	no
*	infix	times	400	left
/	infix	divide	400	no
div	infix	divide	400	no
/	infix	bit or	250	left
/				
	infix	bit and	240	left
	infix	mod	300	no
mod	infix	mod	300	no
˜	prefix	bit complement	220	no
`	prefix	meta-variable	210	no
?	prefix	meta-variable	210	no
$\wedge$	suffix	display	200	no

Note $\neq$ is represented by the tri-graph =
=.

Appendix 4: Line Editing Commands

In the following "point" refers to the position in a line represented by the left edge of the keyboard cursor. It is always between characters, never on them.

Back-Space	(or Delete) deletes the character to the left of point.
$\langle$control$\rangle$ D	deletes the character to the right of point.
$\langle$control$\rangle$ A	moves point to the beginning of a line.
$\langle$control$\rangle$ E	moves point to the end of a line.
$\langle$control$\rangle$ U	kills all input to the left of point.
$\langle$control$\rangle$ K	kills all input to the right of point.
$\langle$control$\rangle$ B	moves point left one character in a line.
$\langle$control$\rangle$ F	moves point right one character in a line.
$\langle$control$\rangle$ T	swaps the characters just to the left and right of point.
$\langle$control$\rangle$ R	re-display the line; point is unaffected.

In addition to the intra-line editing functions described above, Logix remembers up to 20 lines and up to 10 partial lines. Remembered lines can be recalled by:

$\langle$control$\rangle$ P — recalls the previous line (circularly);
repeat to step backward.

$\langle$control$\rangle$ N — recalls the next line (circularly);
repeat to step forward.

$\langle$control$\rangle$ I — clears the remembered line record.

$\langle$control$\rangle$ X $\langle$control$\rangle$ P
: recalls previous killed partial line (circularly);
(repeat $\langle$*control*$\rangle$ P to step backward);
see $\langle$*control*$\rangle$ K, $\langle$*control*$\rangle$ U.

$\langle$control$\rangle$ X $\langle$control$\rangle$ N
: (repeat $\langle$*control*$\rangle$ N to step forward)
recalls next killed partial line (circularly) —

$\langle$control$\rangle$ X $\langle$control$\rangle$ I
: clears the remembered partial line record.

Appendix 5: Monitors

A monitor is a server with an internal state. The state may be altered dynamically as a result of requests served by the monitor.

The monitor manages its own input stream; it serves the stream sequentially, but only when it chooses to do so. Thus any kind of request synchronization may be explicitly programmed.

We show how a *counter* monitor can be defined and implemented.

```
/*
 * counter.cp
 *
 * maintain a named value.
 */

–monitor(counter).
–export([up/1,down/1,value/1]).

counter(In) ← counter(In?,0).

counter(   [up(N)—In],Value) ←
           Value1 := Value + N,
           counter(In?,Value1).
counter(   [down(N)—In],Value) ←
           Value1 := Value – N,
           counter(In?,Value1).
counter(   [ ],_Value).
counter(   [value(Value)—In],Value) ←
           counter(In,Value).
counter(   [X—In],Value) ←
           otherwise —
           computation#error(counter,dont_understand,X),
           counter(In?,Value).
```

References

[1] Hirsch, M., Silverman, W. and Shapiro, E., Computation Control and Protection in the Logix System, Chapter 20 in [6].

[2] Houri, A. and Shapiro, E., A Sequential Abstract Machine for Flat Concurrent Prolog, Chapter 38 in [6].

[3] Mierowsky, C., Taylor, S., Shapiro, E., Levy, J. and Safra, M., The Design and Implementation of Flat Concurrent Prolog, Weizmann Institute Technical Report CS85-09, 1985.

[4] Nakashima, H., Tomura, S. and Ueda, K., What is a Variable in Prolog? *Proc. International Conference on Fifth Generation Computer Systems*, pp. 327–332, Tokyo, 1984.

[5] Shapiro, E., A Subset of Concurrent Prolog and Its Interpreter, Chapter 2 in [6].

[6] Shapiro, E. (Editor), *Concurrent Prolog: Collected Papers*, Vols. 1 and 2, MIT Press, 1987.

[7] Taylor, S., Av-Ron, E. and Shapiro, E., A Layered Method for Process and Code Mapping, Chapter 22 in [6].

[8] Silverman, W., Hirsch, M., Houri, A. and Shapiro, E., Supplement to User Manual for System 2.0, Logix System distribution documents.

[9] Kliger, S., Yardeni, E., Kahn, K., and Shapiro, E., The Language FCP(:,?), *Proc. International Conference on Fifth Generation Computer Systems*, pp. 763–773, ICOT, Tokyo, 1988, and Weizmann Institute Technical Report CS88-07, 1989.

[10] Shapiro, E., The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys* **21**:3, September 1989, pp. 271-278, and Weizmann Institute Technical Report CS89-08, 1989.