# OPRS Development Environment
# Version 1.0.0

Oprs Software
felix@laas.fr
http://softs.laas.fr/openrobots/tools/openprs.php

June 8, 2004

# Contents

# List of Figures

# List of Tables

# Part I

# Licensing Information

# Part II

# Overview

# Overview of the OPRS Development Environment

The OPRS Development Environment is a set of programs designed to help users create applications using Procedural Reasoning [IGR92]. The concept of Procedural Reasoning appeared a few years ago at SRI International in Menlo Park, California. To date, it has been the subject of many research projects and numerous publications have been written on this new programming paradigm (see Bibliography 426).

OPRS Development Environment is, to our knowledge, the first complete Procedural Reasoning development environment written in C and available under Unix. Its graphic interface is based on MIT's X Window version 11 ( X11), and Open Software Foundation's Motif widget set (Motif). It is the first implementation of OPRS intended as a complete software product and environment, unlike other previous versions which were research prototypes usually written in different Lisp dialects.

## 0.1   What is Procedural Reasoning?

Procedural Reasoning is a set of tools and methods representing and executing plans and procedures. These plans or procedures are conditional sequences of actions which can be run to achieve given goals or to react in particular situations. Procedural Reasoning differs from other commonly used knowledge representation (rules, frames, etc.)

To a degree, procedural representation is a trade-off between purely declarative representations and strictly imperative representations. Declarative representations suffer from a lack of control on the execution of their rules, imperative suffer from limited modularity. Procedural Reasoning is particularly well suited for problems where implicit or explicit knowledge is already formalized as procedures or plans. The control information (i.e. the sequence of actions and tests) embedded in these procedures or plans is actually preserved.
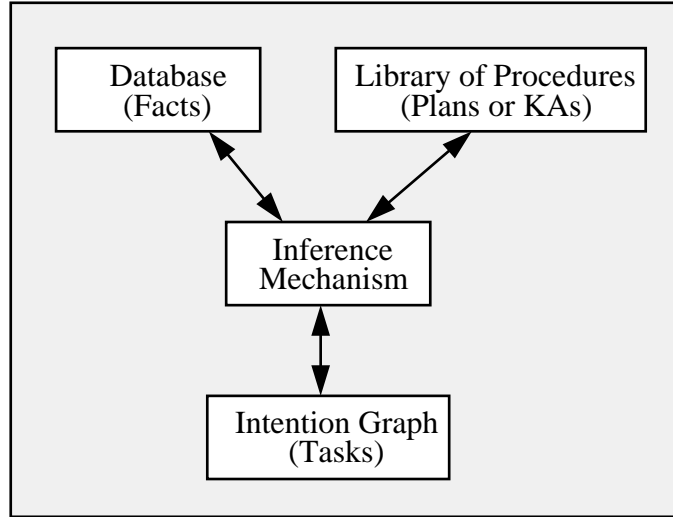
Figure 1: OPRS Global Architecture

## 0.2 Overall Description of OPRS

The C Procedural Reasoning System (OPRS) is a generic architecture for representing and reasoning about actions and procedures in a dynamic domain. It has been applied to various tasks with real-time demands, including malfunction monitoring for different subsystems of NASA's space shuttle [GI90b], the diagnosis, monitoring and control of telecommunications networks [WI91], the control of mobile robots [Rev92] and system control for a surveillance aircraft [IGL89].

As shown in Figure 1, the architecture of a OPRS Kernel consists of (1) a database containing the system of current beliefs about the world; (2) a library of plans (or procedures), called Knowledge Areas (OPs), that describe particular sequences of actions and tests that may be performed to achieve given goals or to react to certain situations; and (3) an intention graph, consisting of a [partially] ordered set of all plans chosen for execution at runtime. An interpreter (inference mechanism) manipulates these components, selecting an appropriate plan (OP) based on system beliefs and goals, placing those selected OPs on the intention structure, and finally executing them.

The OPRS Kernel interacts with its environment through its database, which acquires new beliefs in response to changes in the environment, and through the actions it performs as it carries out its intentions. Different instances of OPRS, running asynchronously, can be used in an application that requires the cooperation of more than one subsystem.

In OPRS, goals are descriptions of desired tasks or behaviors. In the logic used by OPRS, the goal to achieve a certain condition C is written as (! C);

the test for a condition is written as `(?  C)`; to wait until the condition is true is written as `(^ C)`; to passively maintain `C` is written as `(# C)`; to actively maintain `C` is written as `(% C)`; to assert the condition `C` is written as `(=> C)`; and to retract the condition `C` is written as `( > C)`. For example, the goal to close valve `v1` could be represented as `(!  (position v1 cl))`, and to test for it being closed as `(?  (position v1 cl))`.

Knowledge about how to accomplish given goals or to react to certain situations is represented in OPRS by declarative procedure specifications called *Knowledge Areas* (OPs). Each OP consists of a *body*, which describes the steps of the procedure, and an *invocation condition*, which specifies under which situations the OP is useful. Together, the invocation condition and body of a OP express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions [GL86b].

The set of OPs in a OPRS application system not only consists of procedural knowledge about a specific domain, but also includes *meta level* OPs — that is, information about the manipulation of the beliefs, goals, and intentions of OPRS itself. For example, typical meta level OPs encode various methods for choosing among multiple applicable OPs. They determine how to achieve a conjunction or disjunction of goals, and compute the amount of additional reasoning that can be undertaken, given the real-time constraints of the problem domain. In achieving this, meta level OPs make use of information about OPs that are contained in the system database or in the property slots of the OP structures.

OPRS has several features that make it particularly powerful as a situated reasoning system, including: (1) the semantics of its plan (procedure) representation, which is important for verification and maintenance; (2) its ability to construct and act upon partial (rather than complete) plans; (3) its ability to pursue goal-directed tasks while at the same time be responsive to changing patterns of events in bounded time; (4) its facilities for managing multiple tasks in real-time; (5) its default mechanisms for handling stringent real-time demands of its environment; and (6) its meta level (or reflective) reasoning capabilities. Some of these features have been discussed in earlier reports and papers [GI89a, GI90a, GI90b].

## 0.3   Example of Procedure/OP in OPRS

The procedure or OP which is presented on Figure 2 belongs to a library of procedures in the control and supervision of a tank truck filling station. Its goal is to appropriately close or open the filling valve. To control the proper execution, it monitors two position sensors placed on the valve. It generates an emergency stop of the station in case of malfunction.

The points shown on the figure 2 are now presented and developed:

1. The *name* of the procedure allows the user to designate it in the different selection menus of the system.

# Move Valve

**INVOCATION:**
(! (POSITION VALVE $X))

**CONTEXT:**
(AND (ALARM YES)
     (DELAY TWO-GOOD $DEL-2TBS)
     (DELAY ONE-GOOD $DEL-1TB))

**EFFECTS:**
((=> (POSITION VALVE $X)))

**DOCUMENTATION:**
"This KA tries to put the valve in position $x.
It waits ($del-2tbs) time units if the two
sensors are good, or ($del-1tb) time units
if only one is good.
After this time, it shutdowns if the trusted
sensor(s) is not in good position. "

```
                                      START
                          (! (POSITION SWITCH-VALVE $X))
                                       N1
                          (! (= @SWITCH-TIME (CURRENT (TIME))))
                                       N2
                          (? (& (TYPE TB $TB1)
                                (STATUS $TB1 FAILED)))
                                   T  N5  F
                                        (^ (V (ELAPSED-TIME
                                               @SWITCH-TIME $DEL-2TBS)
                                            (& (POSITION SENSOR1 $X)
                                               (POSITION SENSOR2 $X))
                                            (& (TYPE TB $TB1)
                                               (STATUS $TB1 FAILED))))
                                                N4
                                        (? (& (TYPE TB $TB1)
                                              (STATUS $TB1 FAILED)))
                                            T  N3  F
      (^ (& (V (ASSOCIATED-TB $TB1 $TB2)                 (? (& (POSITION SENSOR1 $X)
               (ASSOCIATED-TB $TB2 $TB1))                       (POSITION SENSOR2 $X)))
            (V (ELAPSED-TIME
               @SWITCH-TIME $DEL-1TB)                          T  N6  F
               (POSITION $TB2 $X))))
                      (^ (& (V (ASSOCIATED-TB $TB1 $TB2)
                               (ASSOCIATED-TB $TB2 $TB1))
                            (V (ELAPSED-TIME
                               @SWITCH-TIME $DEL-1TB)
                               (POSITION $TB2 $X))))
                                       N12
                          (? (POSITION $TB2 $X))        (! (SHUTDOWN (TWO-SENSORS $DEL-2TBS)))
                                   T  N7  F                      END1
                          (! (SHUTDOWN (TRUSTED $TB2 $DEL-1TB)))
                                      END0
```
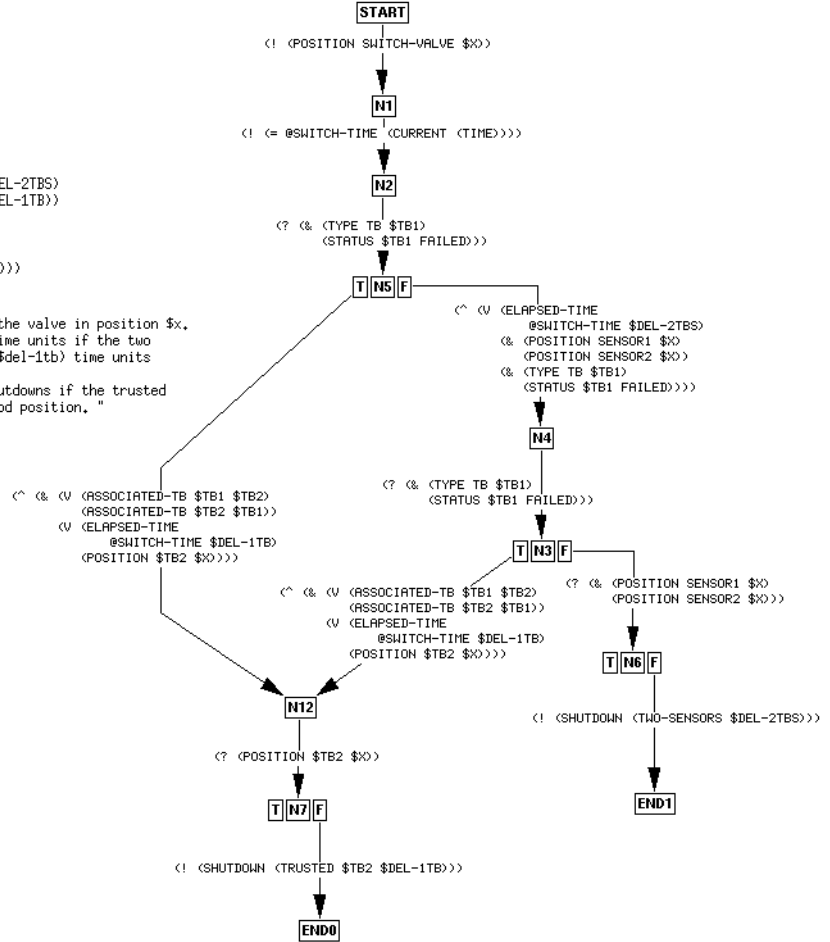
Figure 2: Example of a OP

2. The *invocation part* specifies which goals or which events may make it applicable. This particular procedure is applicable whenever the system has the goal to put the valve in position `$X` (`$X` is a variable which can take two values: "open" or "closed", which will be known at run-time).

3. The *context part* further specifies the conditions of applicability of the procedure. In this case, it will determine the acceptable response delays on the valve position sensors.

4. The *effects part* specifies the facts you want to add or retract from the database upon successful execution of the procedure. Here, if the procedure successfully executes, it will conclude the new position of the valve in the database.

5. The *documentation* field is self explanatory.

   Then, there is the "procedural" part which specifies the sequence of tests and actions to evaluate when the procedure is executed.

6. The "`START`" node is the starting point of the procedure. To successfully execute a procedure, one must satisfy all the actions and conditions which lead to an "`END`" node. This is done by jumping from node to node, while satisfying the condition which label the edge connecting this two nodes. For the "Move Valve" procedure, the first action to be done is to put the "switch" in the proper position: `$X`. There is only one node "`START`" in a procedure.

7. There can be more than one outgoing edge from a specific node. In this case, the system will try to satisfy one condition after another. As soon as a condition is satisfied, we can make the transition to node at the head of the edge (the node at which the edge points).

8. Execution proceeds from node to node until it reaches an "`END`" node. When one "`END`" node is reached, the execution of the procedure is considered successful. If it was goal invoked, then this goal is considered achieved. In our case, the valve will indeed be in position `$X`. If no "`END`" node can be reached, then the execution is considered to be failed, and the goal to be achieved remains to be satisfied.

In a typical OPRS application, one defines a library of such procedures and OPs. These procedures are then loaded in a OPRS Kernel which will execute them whenever they are applicable, i.e., when their invocation and context parts are true.

## 0.4   The OPRS Development Environment

The OPRS Development Environment is designed to allow the user to implement applications in control and supervision of complex systems, automatic execution
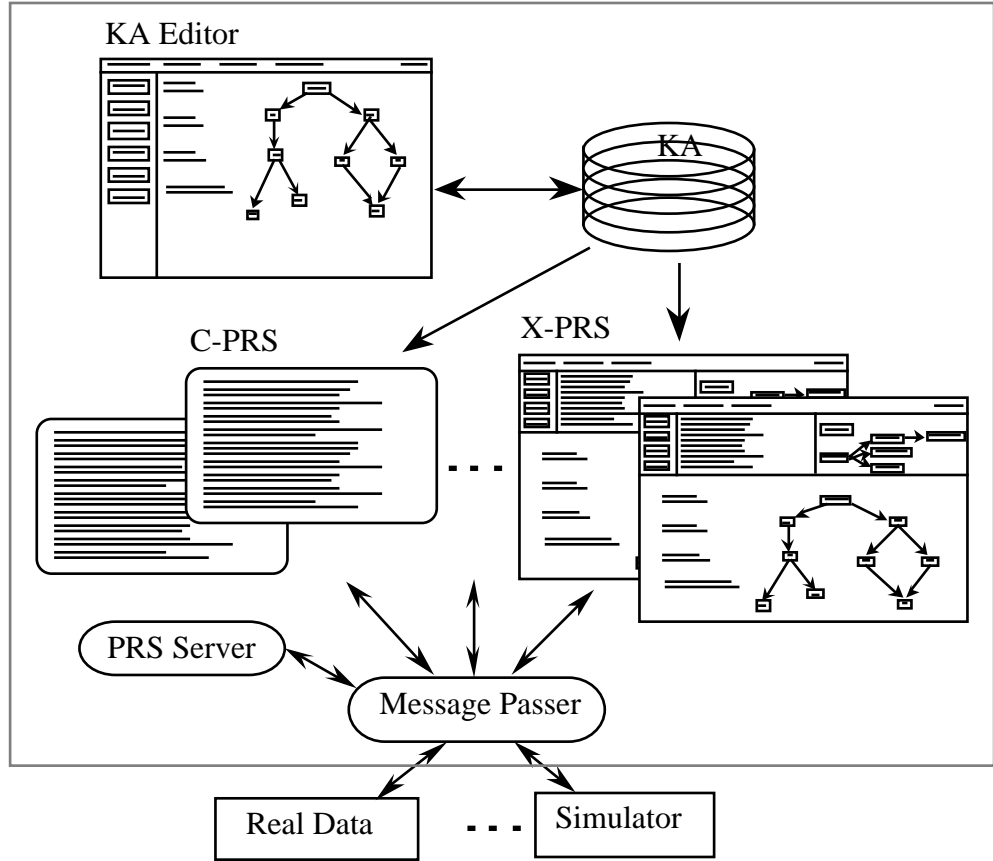
Figure 3: OPRS Development Environment

of predefined procedures, etc. Thanks to its modular architecture, it is easy to integrate an application developed with OPRS in already existing systems.

The OPRS Development Environment is composed of different programs and modules:

- A OP Editor, which enables the user to create, edit and modify its application procedures.

- A OPRS-Server, which enables the user to asynchronously manages a number of OPRS Kernels and X-OPRS Kernels.

- A Message Passer , which is the module which enables an application and external modules to communicate with the different kernels of the OPRS application.
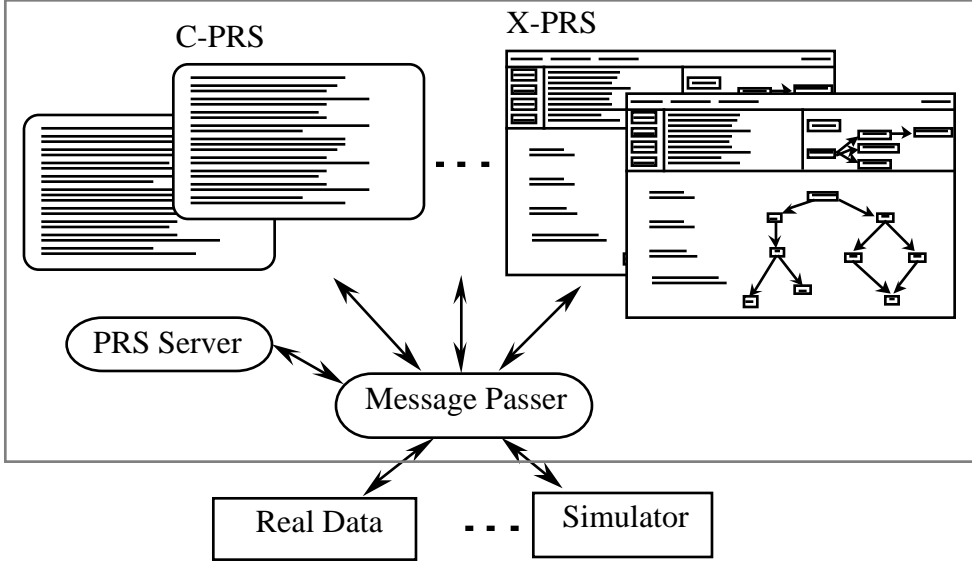
Figure 4: OPRS Application Environment (graphical version)

- Some OPRS Kernels and X-OPRS Kernels, which execute the procedures of your application.

A OPRS application is organized around a "Message Passer" module and a "OPRS-Server" module. However, one can run as many OPRS Kernels or X-OPRS Kernels as required by the application on any machine of the network.

## 0.5 The OPRS Application Environment

The OPRS Application Environment is designed to run OPRS applications developed under the OPRS Development Environment. It allows the user to execute the procedures exactly as they are executed in the OPRS Development Environment. However, it does not allow the user to modify or edit the existing procedures. This environment is particularly well suited for a site using a OPRS application developed by a third party.

According to the needs of your application, the OPRS Application Environment exists in two versions:

A graphic version, under X11/Motif, which enables the user to follow the graphic execution of the procedures and the evolution of the task graph in the X-OPRS Kernels (Figure 4).

An ASCII version which enables the user to execute the procedures in a standard Unix environment (Figure 5). This version, functionally identical to
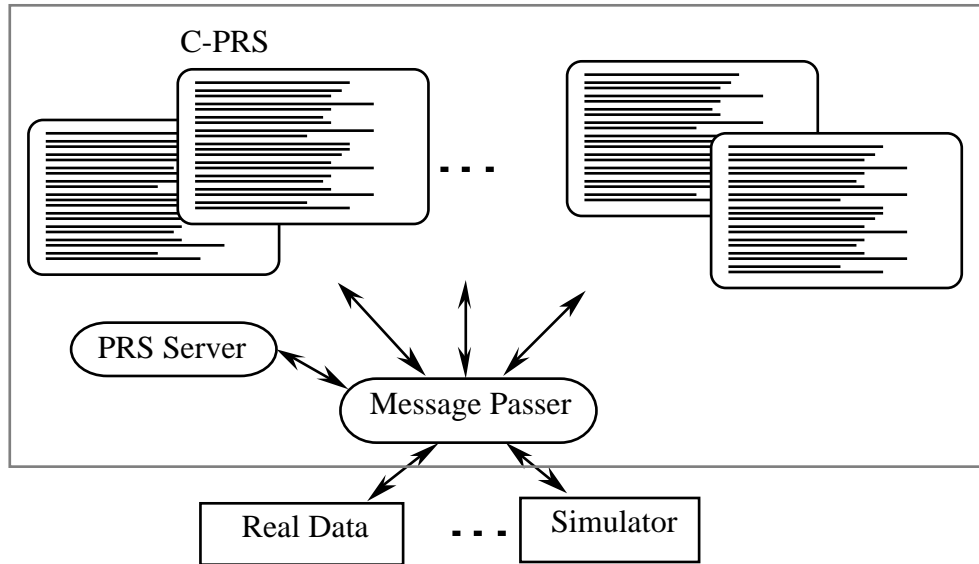
Figure 5: OPRS Application Environment (ASCII version)

the previous one, does not allow the user to follow the graphical execution of procedures.

Whatever version is chosen, the licensing mechanism stays the same; the "Message Passer" and the "OPRS Server" are the central and unique modules around which as many as required by the application, OPRS Kernels or X-OPRS Kernels are run.

## 0.6 The Structure of this Manual

This manual is available in hard copy or as an on-line help function that you can access while using the OPRS Development Environment. The book is a standard manual, with parts, chapters, sections, appendices, etc. . . The on-line help is structured in a similar way with each section accessible through a series of menus or hypertext links. Additionally, the appropriate section of the on-line manual is directly accessed when you request help from the OPRS Development Environment. For example, most dialog boxes in the Motif interface have a HELP button. When you activate any HELP the system will show the appropriate documentation page from which you can then access any part of the manual.

This manual is organized in eight parts to help you easily find the information you need. Each part presents one particular program or module of the OPRS Development Environment. Because of the many interconnections and cross references between these different parts, the manual is structured to explain these cross references whenever possible.

Different parts can be read in any order, although the order in which they are presented is certainly the preferred one. It is suggested that you read a part describing a module before reading one describing the same module's X11/Motif interface. For example, one should read the OPRS Kernel part before the X-OPRS Kernel part.

**Overview:**

**OPRS Kernel:** This part introduces the most important modules of the OPRS Development Environment. The OPRS Kernel is really the central program of the OPRS Development Environment. However, it needs the other modules and programs to be used as a real application. It executes the procedures produced with the OP Editor. It is created and can be interacted with, using the OPRS-Server. It can be run under X11/Motif using the X-OPRS interface. It can communicate with the external world and other OPRS kernels using the Message Passer. It represents the most important and biggest part of the present manual.

**OPRS-Server:** This part introduces the OPRS-Server program which can create, kill and allow the user to interact with OPRS Kernels.

**Message Passer:** This part introduces the module which allows the OPRS and X-OPRS kernels to communicate with one another and with external programs.

**X-OPRS Kernel:** This is the X11/Motif companion of the OPRS Kernel module. In this interface, you can graphically trace the procedures which are executing and follow the various tasks the kernel is working on.

**OP Editor:** This part describes the program which is used to create, edit and modify OPs and procedures. It is a graphical editor based on X11/Motif.

**Using OPRS:** This part describes how to use the OPRS Development Environment. A step by step OP Editor session is presented, as well as an example of how to run an X-OPRS Kernel application. This part also goes through the process of explaining the various choices you will need to make to set-up a OPRS application, and gives some example of such application.

**Appendices:** The appendices describe various topics about the OPRS Development Environment: installation (see [How to Install the OPRS Development Environment], §J, page 369), differences from SRI PRS (see [Principal Differences from SRI PRS], §**??**, page **??**), default OPs (see [Default OPs], §F, page 315), the Grammar Used in the OPRS Development Environment (see [Grammar Used in the OPRS Development Environment], §K, page 375), etc., as well as a glossary, a bibliography and various indices which point you back to the appropriate section of the manual.

Some chapters or sections have arbitrarily been put in a particular part, although they could equally well be presented in elsewhere. For example, the OP syntax section (see [OP Syntax and Semantic], §4, page 57) is part of the OPRS Kernel part, although it could be in the OP Editor part. In any case, the various indices and cross references will always point you to the proper section when necessary.

Comments and problems about this documentation should be reported to the following electronic mail address: felix@laas.fr or entered in the Bugzilla database: http://softs.laas.fr/.

# Part III

# OPRS Kernel

# Overview of the OPRS Kernel

The OPRS Kernel is certainly the most important program of the OPRS Development Environment. It is the program which executes the OPs and the procedures. It is the core of the OPRS technology. A OPRS Kernel is composed of:

- A database containing the facts loaded in the system,

- A library of plans and procedures describing the conditional sequences of actions which must be done to achieve specific goals or to react to particular events,

- A tasks graph composed of a partial order of the plans chosen for execution.

OPRS is the implementation in C of the OPRS technology. The OPRS main assets are:

- advanced language for procedural representation,

- possibility of redefining the control algorithm of the main loop in this same language,

- real-time main loop which guarantees a bound on reaction time,

- possibility of defining or redefining evaluable predicates, evaluable functions and actions to suit your application,

- possibility of tuning the kernel to your application,

- advanced and non-monotonic use of the database,

- each OPRS Kernel is an independent Unix process,

- available under various Unix (Solaris, Ultrix, Sun OS, etc) and real-time Unix system (VxWorks),

- reduced size of the kernel (few hundreds of kilo bytes), compared to Lisp and systems developed in Lisp.

You can call this program directly from a Unix shell or you can call it directly from the OPRS-Server when you execute the `make` command.

The OPRS Kernel is also a "part" of the X-OPRS program. In fact, the kernel part of X-OPRS is the OPRS Kernel. In one case, the main loop runs alone; in the other case, it runs interleaved with the Xt Application Main Loop. Note that if the OPRS Kernel is running alone (not in the Xt Application Main Loop), then the performances are increased. Any performance study should be made with the kernel alone (except, of course, if the goal is to evaluate the performance of the X interface).

# Chapter 1

# How to Use the OPRS Kernel

The OPRS can be called directly from the keyboard or, if you prefer, you can create, kill or modify a OPRS Kernel using commands of the OPRS-Server (see [Commands of the OPRS-Server], §11.3, page 150, for details). The OPRS Kernel can be run without being connected to the OPRS-Server but still needs to be connected to the Message Passer. Connection to the Message Passer is mandatory, but automatically performed by the kernel. If there is already a Message Passer running on the specified host/port, then the kernel will connect to it. Otherwise, it will attempt to start one and then to connect to it.

If you start a OPRS Kernel from the OPRS-Server, connections to the OPRS-Server are made automatically. However, if you start it from a Unix shell (on another host for example) then you need to connect it to the OPRS-Server (unless you specify the `-a` argument. This is done by issuing the `accept` command in the OPRS-Server.

## 1.1   How to Start a OPRS Kernel

You can start a new OPRS Kernel either by typing the command `oprs` at the Unix prompt:

```
% oprs FOO
```

This will only work if there is already a OPRS-Server running, or you will get this error message: `''client:  connect:  Host is unreachable''`.

If there is a OPRS-Server, you will be reminded to issue the `accept` command in the OPRS-Server with the following message: `''Go in the oprs-server, and execute the accept command.''`

or you can start a new OPRS Kernel with the `make` command of the OPRS-Server (the `make_x` command start a X-OPRS Kernel instead):

```
OPRS-Server> make foo
```

Note that if you start OPRS from the Unix shell, you may want to specify a number of options or arguments, for example, if the OPRS-Server and the Message Passer don't use the default socket port for their socket communication, then you need to specify in the `oprs` command the port number on which both programs expect connections. Unless you started the kernel with the `-a` argument, you are then required to register this kernel from the OPRS-Server with the `accept` command.

Upon start-up, and just before entering the OPRS Kernel main loop, the kernel will execute the: `start_kernel_user_hook` function (see [User Hooks], §10.2, page 135). This function can be used for example to initialize some data structures, or to install an intention scheduler (see [Intention Graph Scheduling], §10.4, page 138). This function is executed before any loading command specified with the `-x` argument.

## 1.2    Arguments to the `oprs` Command

If a OPRS Kernel is created from the Unix shell, then you can specify a number of arguments. There is a large number of possible arguments, but most of them are seldom used.

Usage:

```
oprs [-s server-hostname] [-i server-port-number] [-a]
    [-m message-passer-hostname] [-j message-passer-port-number]
    [-x include-filename]* [-c oprs-command]*
    [-I size-symbol-hash-table] [-P size-pred-hash-table]
    [-D size-database-hash-table] [-F size-function-action-hash-table]
    [-d oprs-data-path] [-p] [-l upper|lower|none] [-L en|fr] [-n] client-name
```

All the arguments are optional except for the name of the OPRS Kernel. It is preferable to use upper case for the name. In fact, the name is upper casified automatically, unless it is surrounded by |.

**-s** to specify the hostname on which the server is running. If the kernel cannot connect to this hostname on the specified port, then the program exits with an error message.

**-i** to specify the port on which the OPRS-Server expects a connection. Do not forget to issue the `accept` command in the OPRS-Server.

**-a** to specify the kernel should runs alone without being connected to the OPRS-Server. In this case you do not need to issue the `accept` command in the OPRS-Server.

**-m** to specify the hostname on which the Message Passer runs or will be started (usually the same hostname as the OPRS-Server). If the OPRS Kernel cannot connect to this hostname on the specified port (even after trying to start the Message Passer), then the program exits with an error message.

-j to specify the port on which the Message Passer is expecting a connection (or will be started if necessary).

-x to specify an include file to load upon start-up. This will save you a `connect` in the OPRS-Server, followed by an `include` and a `disconnect` in the OPRS Kernel. Reminder: include files can contain other `include` directives. This option can appear more than once, in which case, files are loaded in the they order are specified.

-c to specify a command to execute upon start-up. This option can appear more than once, in which case, the commands are executed in the order they are specified.

-d to specify a data path, i.e. a colon separated list of directories where the kernel will look for data files (*'.inc'*, *'.opf'* and *'.db'*) (see [OPRS Kernel Environment Variables], §1.3, page 24).

-I to specify the size of the symbol hash-table (default size: 1024).

-D to specify the size of the database main hash-table (default size: 1024).

-P to specify the size of the predicates hash-table (default size: 64).

-E to specify the size of the evaluable functions, actions and evaluable predicates hash-tables (default size: 128).

-F to specify the size of the function/action hash-table (default size: 64).

-p can be used to print the temporal operator in English instead of the single letter. It will print `achieve` instead of !, and `wait` instead of ^ and so on.

-l upper|lower|none can be used to print and parse all the symbol and id in upper case, lower case or in no particular case. You may specify this option by setting the OPRS_ID_CASE environment variable: Example:

```
setenv OPRS_ID_CASE none
```

-L en|fr can be used to select the language of the interface (French or English). Note that by default your kernel is in English. Note also that for the applications with an X interface (i.e. X-OPRS Kernel and the OP Editor the choice of the *'app-defaults'* file will select the language (see [Xt/Motif Widgets Hierarchy and Resources], §L, page 385). In this case, selecting a different value with the option will lead to a warning and to a mix of language in the interface.

-n can be used to specify the name of the kernel. The -n is only required if the name is not the last arguments.

## 1.3 OPRS Kernel Environment Variables

A number of environment variables can be used to customize the OPRS Kernel or to define default arguments. Arguments passed using the command line have precedence on those acquired from environment variables.

OPRS DATA PATH is used to specify a data path, i.e. a colon separated list of directories where the kernel will look for data files (*'.inc'*, *'.opf'* and *'.db'*). It is used by the OPRS Kernel and the X-OPRS Kernel. It is equivalent to the -d command line argument. Example:

```
setenv OPRS_DATA_PATH ./data:/usr/local/oprs/data:${HOME}/data
```

OPRS DOC DIR is used to specify the location of the online OPRS Development Environment documentation. It is used by the X-OPRS Kernel and the OP Editor. Example:

```
setenv OPRS_DOC_DIR /usr/local/oprs/doc
```

OPRS MP PORT is used to specify the port on which the Message Passer will listen to connection. It is used by the OPRS Kernel, the X-OPRS Kernel, the OPRS-Server and the Message Passer. It is equivalent to the -j command line argument. Example:

```
setenv OPRS_MP_PORT 3456
```

OPRS MP HOST is used to specify the host on which the Message Passer will listen to connection. It is used by the OPRS Kernel, the X-OPRS Kernel, the Message Passer and the OPRS-Server. It is equivalent to the -m command line argument. Example:

```
setenv OPRS_MP_HOST alf.laas.fr
```

OPRS SERVER PORT is used to specify the port on which the OPRS-Server will listen to connection. It is used by the OPRS Kernel, the X-OPRS Kernel and the OPRS-Server. It is equivalent to the -i command line argument. Example:

```
setenv OPRS_SERVER_PORT 3457
```

OPRS SERVER HOST is used to specify the host on which the OPRS-Server will listen to connection. It is used by the OPRS Kernel and the X-OPRS Kernel. It is equivalent to the -s command line argument. Example:

```
setenv OPRS_SERVER_HOST alf.laas.fr
```

OPRS_ID_CASE is used to specify if the program should upper case, lower case
or should not change the case of the parsed Id. This is equivalent to the
-l option. The possible values are `lower`, `upper` or `none`: Example:

```
setenv OPRS_ID_CASE none
```

The various _PORT and _HOST environment variables are very useful when
different users are using the OPRS Development Environment on the same host.
By setting these variables to the proper value, they can make sure that their
application will not interact with each other.

## 1.4 How to Kill a OPRS Kernel

You can kill a OPRS Kernel with the `kill` command of the OPRS-Server:

```
OPRS-Server> kill foo
```

If you kill a OPRS Kernelthat is already dead, you get a warning.

You can also kill a OPRS Kernel with the `quit`, `q` or `EOF` commands. In
this case, you need to be connected to this OPRS and issue the command at
the prompt:

```
FOO> quit
```

## 1.5 OPRS Kernel over Network

All the communications between the various components of the OPRS Develop-
ment Environment are made using Internet sockets. If your license agreement
allows it, you can run a OPRS Kernel on a machine different from the one
on which the OPRS-Server runs. On some machines with notably slow process
switching it is strongly advised to run the OPRS-Server and the Message Passer
on a host different from the one upon which the OPRS Kernel is running. In
any case, you can start a OPRS Kernel from any host, providing you specify the
proper arguments to enable the OPRS Kernel to establish connection with the
OPRS-Server and the Message Passer. Note that you need to issue an `accept`
command in the OPRS-Server to establish the connection.

## 1.6 How to Connect to a OPRS Kernel

When you start a OPRS Kernel, it enters its main loop and is in *run* mode. In
this mode, the kernel checks for messages from the Message Passer and com-
mands from the OPRS-Server, but does not interact directly with the user. If
you want to interact directly with a OPRS Kernel (to consult the database, or
load some OPs, and so on), you can do it using the `connect` command of the
OPRS-Server. This will put the kernel in *command* mode. Note: This is not
true for the X-OPRS Kernel with which you can interact using the X/Motif
interface.

For example, assuming you have created a OPRS Kernel named `FOO`, you can connect to it by issuing the following command from the OPRS-Server window.

`OPRS-Server> connect foo`

If the designated kernel has been started from the server, then the prompt changes from:

`OPRS-Server>`

to:

`FOO>`

in the same window or terminal.

If the designated kernel has been started from a Unix shell, then the new prompt appears in the window or tty where it was started, and the OPRS-Server is blocked, waiting until the OPRS Kernel leaves its *command* mode.

Note that at this point, the user is connected through the keyboard to the designated OPRS Kernel, and everything typed is in fact interpreted by the OPRS Kernel. Consequently, the OPRS-Server is now disconnected from the keyboard and will wait until the OPRS Kernel releases it.

One problem with this scheme is that during this time, the OPRS Kernel is deaf to the rest of the world. For example, it does not check for messages from the Message Passer, and it does not even run its top level loop... However, the messages are queued (as much as possible) and the main loop will parse them upon restart. In any case, the connection to a OPRS Kernel should be used as seldom as possible and the kernel should then be left alone to run its own life. Nevertheless its use is not prohibited, and is usually appropriate to load database, or OPs just after the OPRS Kernel has been started.

To disconnect the user and the keyboard from the OPRS Kernel, the user can type:

`FOO> disconnect`

The prompt then becomes:

`OPRS-Server>`

if the kernel was in the same window, otherwise this prompt appears again in the window where the OPRS-Server is running.

You can then interact with the OPRS-Server again, and the OPRS Kernel returns in *run* mode.

If you happen to break the command parser, you need to reset it. Do this by typing a dot . at the beginning of a new line, i.e. 'RET', '.', and 'RET'. See [OPRS Kernel Parser], §2.1, page 27 for more on this subject.

Keep in mind that you do not need to connect to a OPRS Kernel to execute a command such as loading a database, or an include file. You can perform all of these from the OPRS-Server using the `transmit` command (see [OPRS-Server Communication Commands], §11.3.2, page 151). The main advantage of the command mode is that the interaction with the user is direct, but the main advantage of the `transmit` command of the OPRS-Server is that it does not stop the OPRS Kernel main loop, it merely slows it down, as the command will be executed by the main loop itself.

# Chapter 2

# OPRS Kernel Commands

The OPRS Kernel has a number of commands which can either be executed by the user (when he is connected to the OPRS Kernel of his choice) or put into an *include file*. Most commands have self-explanatory names however, their syntax is very rigorous and the command parser will detect any inconsistency. If you put your command in an include file, keep in mind that it can contain any command except the `connect` and `disconnect` commands.

The OPRS Kernel commands are classified in different categories and are presented in the following sections. Some commands appear in more than one category and are presented in each category with the appropriate cross reference.

## 2.1   OPRS Kernel Parser

The OPRS Kernel uses a parser to parse the user commands (so does the OPRS-Server and to some extend the OP Editor and the X-OPRS Kernel). If for some reasons the parser gets confused, presumably because of a syntax error, you can type a dot at the beginning of a new line, followed by a carriage return to reset it (i.e. 'RET', '.', and 'RET'). In any case, when you reset the parser, you must get the prompt after a warning.

```
FOO> echoo (asd)
FOO: warning: syntax error near ECHOO
.
FOO: warning: Parsing error, unknown command, reseting the parser near .

FOO> echo (asd)
( ASD )

FOO>
```

If you do not get the prompt, it means that the parser is still expecting a token to parse. (In some very peculiar situations, the parser needs to get close

parenthesis `)` to accept the reset token `\n.\n`).

## 2.2   OPRS Kernel Database Commands

The database commands allow the user to conclude, consult, delete facts, and modify some properties of the database.  These commands are usually used whenever the user wants to interact directly with the database. However, while the OPRS Kernel is running, facts will be concluded/deleted/consulted by the kernel itself.

- `delete expression`. Delete the `expression` (see [General Expressions], §3.6, page 50) from the database.  Note that you can delete expressions containing variables. Variables in this context are universally quantified, i.e. all the matching expressions will be deleted.  Note that you cannot delete gexpressions containing logical operators. Example: `delete (foo a b $x) delete (bar 12.0 (+ 4 5))`

- `consult gexpression`. Consult the `gexpression` (see [General Expressions], §3.6, page 50) in the database. Note that the consult command accepts a gexpression (i.e. expression combined with logical operators), not only an expression. Example: `consult (bar $x b c) consult (& (foo 4 "String") (bar $x b c)) consult (|| (foo $x "String") (bar $x b c))`

- `conclude expression`. Conclude the `expression` (see [General Expressions], §3.6, page 50) in the database.  You cannot conclude an expression with unbound variables, and you can only conclude expression, not gexpression. Example: `conclude (bar a b c)`, and `conclude (  (foo 2 3 (+ 2 3)))`, are accepted, however `conclude (bar $x b c)` is not allowed because of the variable, nor is `conclude (& ( bar boo) (boo bar))` because of the conjunction.

- `show db`. This command displays all the expressions which are currently contained in the database. They are not displayed in any particular order, so it can be rather tedious to look for a particular fact.

- `save db 'file_name'`. This command saves the contents of the database in a the file 'file_name'. It is saved in a format suitable to be read by the kernel.  Example: `save db "my-database.db" save db "/usr/name/oprs/application.db"`

- `empty fact db`. Empty the database. It clears and frees all the contents of the database. Functional facts, closed world predicates and basic events *declarations* are preserved.

- `load db 'file_name'`. Load the content of 'file_name' in the database. See [OPRS Kernel Loading Commands], §2.4, page 30 for more information on this command.

## 2.3   OPRS Kernel OP Library Commands

The OP library contains all the OP/procedures which have been loaded in the OPRS Kernel. There are a number of commands to load/unload OP files in the OP library of a OPRS Kernel, as well as to clear/consult the OP library, and to set/unset trace on specific OP or OP files (i.e. set of OPs loaded from the concerned file).

- `delete op op_name`. Delete the specified OP from the OP library.

- `delete opf file_name`. Delete all the OP which were contained in the OP File specified as argument.

- `show op op_name`. Print the specified OP from the OP library.

- `list op`. List all the OPs loaded in the kernel.

- `list opfs`. List all the OP Files loaded in the kernel.

- `trace graphic op op_name on|off`. Set the graphic trace on or off for the specified OP.

- `trace step op op_name on|off`. Set the step status on or off for the specified OP.

- `trace text op op_name on|off`. Set the text trace on or off for the specified OP.

- `trace graphic opf file_name on|off`. Set the graphic trace on or off for all the OPs in the specified loaded OP file.

- `trace step opf file_name on|off`. Set the step status on or off for all the OPs in the specified loaded OP file.

- `trace text opf file_name on|off`. Set the text trace on or off for all the OPs in the specified loaded OP file.

- `load opf 'file_name'`. Load all the OPs contained in the OP File specified as argument. See [OPRS Kernel Loading Commands], §2.4, page 30 for more information on this command.

- `reload opf 'file_name'`. Unload and then load all the OPs contained in the OP File specified as argument. See [OPRS Kernel Loading Commands], §2.4, page 30 for more information on this command.

- `consult relevant op goal|fact`. Return the OPs which are relevant for the goal or the fact given as an argument. Relevant OPs are not applicable. These are just the OPs which may be considered for applicability.

- `consult applicable op goal|fact`. Return the OPs which are applicable, with their binding environment (see [Frames and Binding Environments], §3.4, page 49) for a particular goal or fact. Note that this applicable OPs will not be executed... Therefore this command is not equivalent to the `add` command (see below). However, the fact (if a fact is used for this consultation) is concluded in the database and then retracted from the database, as if it was a basic event fact (see [Basic Events], §5.7, page 88).

## 2.4   OPRS Kernel Loading Commands

File name can be given relative to the current directory or absolute (starting with a `/`). This rule also applies to file loaded by means of commands in include files. Remember that files are searched in the `OPRS_DATA_PATH` path (see [OPRS Kernel Environment Variables], §1.3, page 24).

- `set oprs_data_path string`. Set the OPRS_DATA_PATH to the given value. This command has priority over the OPRS_DATA_PATH given as environment variable or as argument to the `oprs` command.

- `show oprs_data_path`. Print the current OPRS_DATA_PATH value.

- `include file_name`. Execute all the commands in `file_name`. The recommended extension for these files is *'.inc'*. Include file can contain other include directives. Only two commands are forbidden in include files: `connect` and `disconnect` (see [Include File Format], §2.14, page 40). Include files can be loaded at start-up time with the `-x` option (see [Arguments to the oprs Command], §1.2, page 22).

- `require file_name`. Execute all the commands in `file_name`. This command is used to load an include file. It is exactly like the `include` command (see above), except that it will check that the file has not already been loaded with another require call.

- `load db file_name`. Load all the facts contained in `file_name`. Note that the format used in OPRS Development Environment is slightly different from the one used in the SRI version of OPRS. The database file contains a list of facts, not just the facts in a file (see [Database File Format], §5.1, page 79). If you have used the Lisp version of OPRS, you can convert your already existing database file by just adding an open parenthesis '(' at the beginning, and a closing parenthesis ')' at the end. Functional facts and closed world predicate declarations are no longer in the database file, but are issued as commands (see [OPRS Kernel Database Commands], §2.2, page 28).

- `load opf file_name`. Load all the OPs contained in the file `file_name`. `file_name` can be *'.opf'* OP file or a *'.dopf'* file. In fact, if a newer *'.dopf'*

file exist, OPRS Kernel will load it instead of the *'.opf'*. The OP Editor accepts different OP File formats (see [OP File Format], §17, page 241). However, the OPRS Kernel (to keep it as small and as efficient as possible) only accepts OPF File format (*'.opf'* or *'.dopf'* suffix). If you wish to load another format, it is necessary to load it under the OP Editor and save it under OPF File format. For more information on the compilation process (see [Procedure and Expression Compilation and Parsing], §4.4, page 76).

- `delete opf file_name`. Delete all the OPs which were contained in the OP File specified as argument.

- `reload opf 'file_name'`. Delete and then load all the OPs contained in the OP File specified as argument. This command keeps track of the OP which were graphic/text traced. If the specified OP file was not loaded, then it is equivalent to a load.

- `empty op db`. Empty the OP Library. It clears and frees all the OPs loaded in the kernel. Memory is reclaimed and will be used by the system. This command can be used while OP are being executed.

## 2.5 OPRS Kernel Trace Commands

There are many ways to trace various parts of the OPRS Kernel execution. Most of these traces can be set/unset in the X-OPRS Kernel using the trace menu (see [OPRS Trace], §13.3.4, page 193).

The general syntax is:
`trace feature on|off`

- `trace relevant op on|off` Turn on or off information on relevant OPs (quite verbose).

- `trace load op on|off` Turn on or off information on the compilation of OPs. (very verbose)

- `trace applicable op on|off` Turn on or off information on the set of applicable OPs.

- `trace conclude on|off` Turn on or off information on fact concluded in the database from the parser. This can be useful if you conclude fact from the parser using internal functions such as `send_command_to_parser`.

- `trace fact on|off` Turn on or off information on facts posted in the kernel.

- `trace goal on|off` Turn on or off information on the goal posted in the kernel.

- `trace suc_fail on|off` Turn on or off information on the success or failure of OPs.

- `trace intend on|off` Turn on or off information in the intention operation in the kernel.

- `trace intention failure on|off` Turn on or off information on intention failure. When an intention fails, the kernel will print the chain of goals which failed and lead to the intention failure.

- `trace send on|off` Turn on or off information on messages sent by the kernel.

- `trace receive on|off` Turn on or off information on messages received by the kernel.

- `trace graphic on|off`. Will set the graphic trace on or off (general flag).

- `trace text on|off`. Will set the text trace on or off for (general flag).

- `trace db on|off` Turn on or off traces on the database operations.

- `trace db frame on|off` Turn on or off the printing of the frames while consulting the database.

- `trace thread on|off` Turn on or off traces on thread creation and merging.

- `trace all on|off` Equivalent to turn on or off: `db`, `receive`, `send`, `intend`, `suc_fail`, `goal`, `relevant op`, `load op`, `applicable op` and `fact`.

All these trace options can be set or unset by the appropriate command in an include file.

## 2.6   OPRS Kernel Run Option Commands

The option commands allow the user to set or unset some options of the OPRS Kernel. Most of these options can be set/unset in the X-OPRS Kernel using the options menu (see [OPRS Run Option], §13.3.5, page 197).

The general syntax is:
`set option on|off`

- `set eval post on|off`. Turn on or off the current-quote mechanism, (see [Current and Quote], §10.7, page 141). Default value: `on`.

- `set parallel post on|off` Turn on or off the parallel posting of goals. When this option is on, one goal for each thread active in the current intention will be posted. Default value: `on`.

- `set parallel intend on|off` Turn on or off the parallel intending of op-instance. Default value: `on`.

- `set parallel intention on|off` Turn on or off the parallel intention execution. When it is `on`, all the intention at the root of the graph are executed. For this option to be really useful, the `parallel intend` option should be on. Default value: `on`. **'This option has some very important consequences on the standard behavior of the kernel'**. In any case the kernel always checks that a particular OP Instance has not been already intended before intending it. This is to make sure, for example, that you do not intend again from a meta level OP, a OP instance already intended by the main loop or another meta level OP. Moreover, the kernel always check that a OP Instance intended for a particular goal is intended in the proper place, i.e. it is not intended if there is already another OP instance which has been intended for the same goal. Note however, that it may be intended later if it is still applicable and if the other one has failed.

- `set time_stamping on|off` Turn on or off the time stamping of various date (for facts, goals, creation date, soak date, etc). Time stamping is quite expensive, which is the reason why this option exists. Default value: `off`.

All these trace options can be set or unset by the appropriate command in an include file.

## 2.7 OPRS Kernel Meta Level Option Commands

The option commands allow the user to set or unset some options of the OPRS Kernel. Most of these options can be set/unset in the X-OPRS Kernel using the options menu (see [OPRS Meta Level Option], §13.3.5, page 199).

The general syntax is:
`set option on|off`

- `set meta on|off`. Turn on or off the meta-level mechanism, (see [Meta Level Reasoning], §9, page 131). Turning it `off` greatly increases the performance of the system. This flag is a general flag. When it is `on`, then you need to select which specific meta fact you want the kernel to conclude or not.

- `set soak on|off`. Turn on or off the posting of the meta fact (SOAK), (see [SOAK and other Meta Facts], §9.1, page 131). Default value: `on`.

- `set meta fact on|off`. Turn on or off the posting of the meta fact (FACT-INVOKED-OPS), (see [SOAK and other Meta Facts], §9.1, page 131). Default value: `on`.

- `set meta goal on|off`. Turn on or off the posting of the meta fact (GOAL-INVOKED-OPS), (see [SOAK and other Meta Facts], §9.1, page 131). Default value: `on`.

- `set meta fact op on|off`. Turn on or off the posting of the meta fact (`APPLICABLE-OPS-FACT`), (see [SOAK and other Meta Facts], §9.1, page 131). Default value: `off`.

- `set meta goal op on|off`. Turn on or off the posting of the meta fact (`APPLICABLE-OPS-GOAL`), (see [SOAK and other Meta Facts], §9.1, page 131). Default value: `off`.

All these trace options can be set or unset by the appropriate command in an include file. For efficiency reasons, it is strongly recommended to only use the meta-level options required by the application.

## 2.8   OPRS Kernel Compiler/Parser Option Commands

The checking commands allow the user to set or unset some compiler/Parser checking of the OPRS Kernel. Most of these checkings can be set/unset in the X-OPRS Kernel using the options menu (see [OPRS Compiler/Parser Option], §13.3.5, page 198).

The general syntax is:

`set checking on|off`

- `set action on|off` Turn on or off some checking on the action compilation/aoprsing. In particular, it will check if an action calls a function declared as an evaluable function. See [Action Checking], §4.4.1, page 76. Default value: `on`.

- `set function on|off` Turn on or off some checking on the function compilation/parsing. Note that it will require all symbols used in place of a function to be declared as a function. Symbols declared as evaluable functions in the kernel are declared as function de facto. See [Function Checking], §4.4.3, page 77. Default value: `on`.

- `set predicate on|off` Turn on or off some checking on the predicate compilation/parsing. Note that it will require all symbols in place of a predicate to be declared. Symbols declared as functional fact or basic events or closed world predicates are declared as predicate de facto. See [Predicate Checking], §4.4.2, page 76. Default value: `on`.

- `set symbol on|off` Turn on or off some checking on the symbol compilation/parsing. All symbols declared by other means (evaluable functions, predicates, actions, cwp, basic events, etc) are declared as symbol de facto. See [Symbol Checking], §4.4.4, page 77. Default value: `on`.

All these checking can be set or unset by the appropriate command in an include file. In development phase, it is stronly recommended to keep all these checkings on.

## 2.9   OPRS Kernel Declaration Commands

- `declare be predicate`. Declare the `predicate` as a basic event predicate (see [Basic Events], §5.7, page 88). This declaration is local to the OPRS Kernel the user is connected to. Example: `declare be foo`

- `declare cwp predicate`. Declare the `predicate` as a closed world predicate (see [Closed World Predicates], §5.5, page 83). This declaration is local to the OPRS Kernel the user is connected to. Example: `declare cwp foo`

- `declare ff predicate integer`. Declare the `predicate` as functional fact for the argument integer (see [Functional Facts], §5.6, page 86). This declaration is local to the OPRS Kernel the user is connected to. Keep in mind that all functional fact predicate are also considered as closed world predicate. Example: `declare ff position 1` `declare ff connection_status 2`

- `declare function function`. Declare `function` as a function (see [Function Checking], §4.4.3, page 77). Example: `declare function pressure-of` × `declare function status`

- `declare id symbol`. Declare the symbol as an id, therefore, the OP compiler does not complain if it encounters this symbol. This command is useful if and only if you perform it before the first occurrence of the symbol in the OPs or in the database. As a consequence, most likely it will appears at the beginning of the include file. See [Symbol Checking], §4.4.4, page 77.

- `declare op_predicate predicate`. Declare the `predicate` as OP predicate (see [OP Predicates], §5.9, page 94). This declaration is not mandatory at all, but considerably improve the speed of the kernel. Example: `declare op_predicate foo` `declare op_predicate print`

- `declare predicate predicate`. Declare `predicate` as a predicate (see [Predicate Checking], §4.4.2, page 76). Example: `declare predicate foobar` `declare predicate connection`

- `undeclare be predicate`. Undeclare the `predicate` as a basic event predicate (see [Basic Events], §5.7, page 88). Example: `undeclare be foo`

## 2.10   OPRS Kernel Listing Commands

- `list action`. List all the functions which have been declared as action (see [Using Action OPs], §7.7, page 114). Example: `list action`

- `list all`. Equivalent to a `list be`, `list cwp`, `list ff`, `list predicate`, `list evaluable predicate`, `list op_predicate`, `list function`, `list evaluable function`, `list action`. Example: `list all`

- `list be`. List all the predicates which have been declared as basic event predicate (see [Basic Events], §5.7, page 88). Example: `list be`

- `list cwp`. List all the predicates which have been declared as closed world predicate (see [Closed World Predicates], §5.5, page 83). Example: `list cwp`

- `list evaluable function`. List all the functions which have been declared as evaluable function (see [Evaluable Functions], §6, page 95). Example: `list evaluable function`

- `list evaluable predicate`. List all the predicates which have been declared as evaluable predicate (see [Evaluable Predicates], §5.8, page 88). Example: `list evaluable predicate`

- `list ff`. List all the predicates which have been declared as functional fact predicate (see [Functional Facts], §5.6, page 86). Example: `list ff`

- `list function`. List all the functions which have been declared. Example: `list function`

- `list op_predicate`. List all the predicates which have been declared as op predicate (see [OP Predicates], §5.9, page 94). Example: `list op_predicate`

- `list predicate`. List all the predicates which have been declared. Example: `list predicate`

## 2.11   OPRS Kernel Dumping/Loading Commands

Since version 1.4, OPRS provides a mechanism to save OP and fact database in a binary compiled dump format. This binary dump format present a number of advantages.

**Faster reload**  The main advantage of using OP and database dump/load is the load speed compare to loading OP or database files. For large applications, with hundreds of OP, the startup time can be critical, and it is always much faster (5 or more times faster) to load a dump file than the equivalent OP or database files. In fact, when one load a OP file, the kernel recompiles the OP in some internal structure, while if you load a dump OP file, the structures are just loaded in memory, avoiding a costly parsing and compilation.

**More than just OP files** . The OP dump files contains in fact more than just the OP description, they also contain all the declarations of predicates, functions, and symbols mentioned in the OPs.

**Architecture and indian independant** The dump format is intentionaly architecture and endian independant. One can create a dump file under a Pentium running Windows NT and reload it under a Sparc running Solaris or a 680x0 board under VxWorks.

**Graphic and non graphic version of OP** . Graphic information is dumped according to the kernel from which you dump the OPs (i.e. it is dumped from an X-OPRS Kernel, while it is not from a OPRS Kernel). Similarly, when loading OP dump, the graphic information is loaded if an only if it is present in the dump and if the kernel may use it (i.e. an X-OPRS Kernel).

There are some drawbacks to the use of the dump/load facilities, and also some misunderstandings:

**Unreadable format** The dump format is almost unreadable to the common user (and even programmer)... Therefore, all OP files should also be kept under source format (*'.opf'*).

**Memory consumption** The dump and reload of OP and database dump files require some memory allocation which in some situations is sparse (under VxWorks for example). Some tables are required to perform the dump and the load. These tables are temporarily allocated and are freed afterward.

**No C compiled code included** C Compiled code, i.e. code linked by the user in the OPRS Kernel is not dumped in the OPRS dump files. The main reason is that contrary to the OPRS compiled code, this code is not architecture independant. Therefore, when you load a OPRS dump file, the kernel will complain if it cannot find the evaluable functions/predicates/actions code it needs.

The currently supported commands are:

- `dump op` *'file_name'*. Dump in binary format all the OP currently loaded in the kernel. The suggested suffix for the dump OP file is *'.dopf'*. If this command isexecuted from the X-OPRS Kernel, graphical information is saved in the file. Example: `dump op "op-ex.dopf"`

- `dump all opf`. Dump in binary format all the OP files currently loaded in the kernel. Each file in its own *'.dopf'* file. This command is very useful when an application is ready and one want to compile all the OPs at once. Example: `dump op "op-ex.dopf"`

- `dump db` *'file_name'*. Dump in binary format all the facts currently in the database. The suggested suffix for the dump db file is *'.ddb'*. Example: `dump db "op-ex.ddb"`

- `load dump op` *'file_name'*. Load all the OP in the dump OP file which must be in binary format, and add them in the kernel (except for the OP already loaded). The suggested suffix for the dump OP file is *'.dopf'*. If this command is executed from the X-OPRS Kernel, and the dump OP file was produced from an X-OPRS Kernel, the graphical information is restored, otherwise, it will be unavailable (but the OP will still remain executable). However, if it is executed from a OPRS Kernel, graphical information if present is discarded. Example: `load dump op "op-ex.dopf"`

- `load dump db` *'file_name'*. Load and add in the database all the fact in the dump fact binary format file. The suggested suffix for the dump db file is *'.ddb'*. Example: `load dump db "db-ex.ddb"`

The dump/load mechanism is the first step of a broader mechanism which will, in future releases, provide other facilities. For example, future version of OPRS will provide warm boot capabilities (i.e. one will be able to dump a kernel state and later, reload it and continue from the point where it was dumped). In fact, the warm boot feature is more or less already present in the current version but is not currently supported (i.e. use at your own risk).

- `dump kernel` *'file_name'*. Dump in binary format the complete kernel (i.e. the database, the OPs, and all the intentions). The suggested suffix for the dump kernel file is *'.dkrn'*. If this command is executed from the X-OPRS Kernel, graphical information is saved in the file. Example: `dump kernel "krn-ex.dkrn"`

- `load dump kernel` *'file_name'*. Load the kernel contained in the dump kernel file which must be in binary dump kernel format, i.e. add the loaded fact in the database, add the loaded OP in the OP library (except for the OP already loaded), and replace the intention graph with the one loaded from the file. The suggested suffix for the dump OP file is *'.dkrn'*. If this command is executed from the X-OPRS Kernel, and the dump kernel file was produced from an X-OPRS Kernel, the graphical information is restored, otherwise, it will be unavailable (but the OP will still remain executable). However, if it is executed from a OPRS Kernel, graphical information, if present, is discarded. Example: `load dump kernel "krn-ex.dkrn"`

## 2.12   OPRS Kernel Status and Control Commands

A number of commands exist to display the status of OPRS Kernel and to control its execution.

- `show run status`. Display all the current status of the kernel. This command is usually sent by the OPRS-Server, with a `transmit` command.

- `step`. This command will step the execution of the OPRS Kernel. This command is usually sent by the OPRS-Server, with a `transmit` command. This will lead to step through one loop of the OPRS Kernel. Note that one loop execution does not always produce any noticeable or visible effect....

- `next`. This command will next the execution of the OPRS Kernel. This command is usually sent by the OPRS-Server, with a `transmit` command. Executing this command will lead the kernel to run until the control hits an edge of step traced OP. This is very useful when you graphic trace OPs. At each click, the execution goes from one traced edge to the next traced edge (whatever execution happens in between).

- `halt`. This command will halt the execution of the OPRS Kernel. This command is usually sent by the OPRS-Server, with a `transmit` command. While the kernel is halted, you can perform most command, such as consulting the fact or OP Library, adding new fact or new goal (in which case they will be taken into account whenever the kernel is restarted), displaying a particular OP, load new OPs, etc.

- `run`. This command will run the execution of the OPRS Kernel. This command is usually sent by the OPRS-Server, with a `transmit` command. It will restart a stopped OPRS Kernel.

- `reset kernel`. Reset the kernel by flushing all the intentions in the intention graph as well as flushing all the goals and facts in the input buffers of the kernel. This does not stop the kernel and this does not flush the fact database and the OP Library.

## 2.13 OPRS Kernel Miscellaneous Commands

- `show intention`. Display all the intentions in the intention graph. It gives an extensive status of the intentions and their component (thread, status, waiting condition, joining, etc).

- `show condition`. Display all the conditions currently active in th OPRS Kernel. It gives an extensive status of the conditions (waiting or preserve condition, as well as their "evolving" status).

- `show memory`. Display some information on the memory usage of the kernel.

- `show variable`. Display all the currently used global variables and their binding.

- `disconnect`. Instruct the connected OPRS to leave the `stdin` and give it back to the OPRS-Server. The OPRS client returns in run mode, and execute its main loop again.

- `unify expression expression`. Unify two expressions (see [General Expressions], §3.6, page 50). This can be used to check the result of the unification mechanism.

- `echo gexpression|gtexpression|gmexpression`. Echo the general expression, general temporal expression or general meta expression to the screen. This is used to check that an expression is properly parsed.

- `send name message`. Send the message (an Expression) to the OPRS named `name`.

- `add goal|fact`. Add a goal (a Temporal Expression) or a fact (an Expression) in the current OPRS Kernel. Most of the time, the user would prefer to use the similar command from the OPRS-Server (which is the same except it take the name of the kernel as first argument). Note that the kernel will not execute any applicable OP because of this fact or goal before it is returned to run mode with the `disconnect` command.

- `stat db`. Print a report on the use of the various hashtables in the database (occupation, collision, etc). It can be used to tune your application by using the appropriate arguments to the `oprs` or `xoprs` command (see [Arguments to the oprs Command], §1.2, page 22).

- `stat id`. Print a report on the use of the symbol hashtables in the kernel (occupation, collision, etc). It can be used to tune your application by using the appropriate arguments to the `oprs` or `xoprs` command (see [Arguments to the oprs Command], §1.2, page 22).

- `stat all`. Print a report on the use of all the various hashtables in the kernel (occupation, collision, etc). It can be used to tune your application by using the adequate arguments to the `oprs` or `xoprs` command (see [Arguments to the oprs Command], §1.2, page 22).

- `q|quit|exit|EOF`. Quit the kernel. This command also disconnects you. It will execute the: `end_kernel_user_hook` function which can be usedto do some cleanup (see [User Hooks], §10.2, page 135).

- `show version`. Print the version of the OPRS Kernel.

- `show copyright`. Print the copyright notice.

- `help|h|?`. Print an on-line help.

## 2.14   Include File Format

*Include files* are files containing command to be executed by a OPRS Kernel. The recommended, but not enforced, extension for these files is *'.inc'*. Include file can contain other include directives. Only two commands are forbidden in include file: `connect` and `disconnect`. Include files can be loaded at start-up

time with the **-x** option (see [Arguments to the oprs Command], §1.2, page 22).
Lines beginning with the ; character are ignored and considered as comments. A
number of include files are provided with the OPRS Development Environment
distribution.

Here is an example of an include file:

```
;;; This is a comment
declare cwp property-p
trace load op off
include "/usr/local/oprs/data/new-default.sym"
load opf "/usr/local/oprs/data/new-default.opf"
declare id factorial
declare id recursive
declare id prefer-iterative
declare id prefer-recursive
declare id print-factorial
declare id sssoak
declare id tas-fact
declare id foo
declare id test_par
declare id TT
declare cwp prefer-recursive
declare cwp prefer-iterative
load opf "/usr/local/oprs/data/fact-meta.opf"
trace graphic opf "/usr/local/oprs/data/fact-meta.opf" on
trace text opf "/usr/local/oprs/data/fact-meta.opf" on
trace graphic op |Print Factorial| off
```

# Chapter 3

# Syntax and Semantic Used in the OPRS Development Environment

The syntax and the associated semantic is the same in the OPRS Kernel, X-OPRS Kernel, OP Editor and OPRS-Server modules. The same grammar is used and syntax checking is done and enforced whenever possible. In other words, unless you bother with the OP files themselves, it is very unlikely that a OP produced by the OP Editor will be rejected by the OPRS Kernel.

A Lex/Yacc like format grammar can be found in [Grammar Used in the OPRS Development Environment], §K, page 375. The grammar is recursive, therefore, it cannot be easily presented with a bottom up approach without making the assumption that some elements are known.

Note that although the syntax is very much like Lisp, but is no Lisp interpreter behind the reader (see [LISPLIST], §H.1, page 361).

## 3.1 Variables

A *variable* is a symbol which can or cannot be bound to different values depending on the context (see [Frames and Binding Environments], §3.4, page 49). This allows the user to specify an expression with part of it unspecified. Depending on the context, a variable can be existentially or universally quantified. These quantifications are usually implicit but obvious. Most often however, variables are existentially quantified, i.e. the expression containing the variable is interpreted with one possible value of all the possible values this variable can take. We will point out the situation where the variables are to be interpreted in a universally quantified way. Finally, there are operators and functions to force the universally quantification (see [Universal Quantification of Variables], §10.9, page 142).
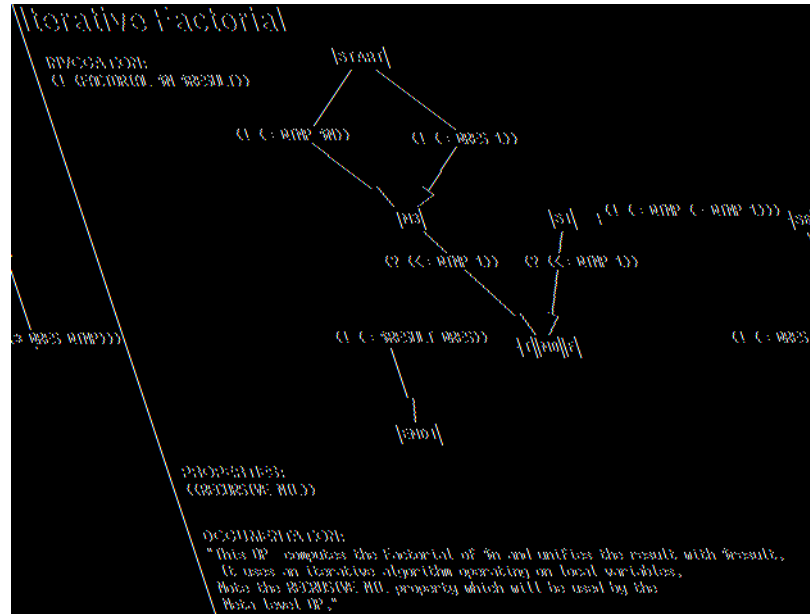
Figure 3.1:  A OP to Compute Factorial with an Inner Loop and Program Variables.

There are two types of variables in OPRS. The *logical variables* (their name starts with the dollar character $) and the *program variables* (their name starts with the at sign character @).

### 3.1.1   Logical Variables

Logical variables start with the dollar character $. Example: $x, $y and $FOO.

The behavior of these variables is quite standard in logic programming (like in PROLOG [CM84]). They are bound once and remain bound. If the execution of the branch on which they were bound is successful, then they stay bound (this is commonly used to bring values back to a caller) .

The scope of these variables is the procedure or the consultation expression in which they appear. Moreover, in recursive procedure call, a new "copy" of the variables is done.

### 3.1.2   Program Variables

Program variables start with the at sign character @. Example: @x, @y and @FOO.

These variables are provided for convenience. They can be bound and re-bound at any time, even if they are already bound to some value. They are not

used in invocation or context parts due to their inability to keep a value. They are mostly used inside the body of a procedure in construction such as a local loop (See Figure 3.1).

The scope of these variables is the procedure or the consultation expression in which they appear. Moreover, in recursive procedure call, a new "copy" of the variables is done.

### 3.1.3   Global Variables

Global variables start with two at sign characters `@@`.

These variables are provided for convenience. They can be bound and rebound at any time (like Program Variables see [Program Variables], §3.1.2, page 44), even if they are already bound to some value. Their main characteristic is that they are visible all over the OPRS Kernel or X-OPRS Kernel in which they are used. The scope of these variables is the whole kernel.

Note that this mechanism should not be used in place of the database itself, it should only be used for information widely used, and ever changing, and which are difficult a priori to compute.

## 3.2   Terms

Terms are an important component of the OPRS syntax. They are of various types and are the building blocks used to build more complex expressions. Terms are generic, and can be of the following types. For each of them, we indicate in parenthesis the C type as defined in *'oprs-type-pub.h'*.

### 3.2.1   Integer as a Term

A term can be an *integer*. Its C type is then `INTEGER`. In the OPRS Kernel, integers become C `int`. A direct consequence is that they are not bignums. Therefore, there are maximum values for these numbers. Check the C compiler and your machine architecture for specific maximum and minimum values.

Example of integers: `0 1 -3 4732 +34 -0 +0 -023`. Example of numbers which are probably not recognized as integers: `123123123123123123 --4`.

### 3.2.2   Float as a Term

A term can be a *float*. Its C type in the Term structure is then `TT_FLOAT` or `FLOAT` (except under Windows 95-NT). In the OPRS Kernel these floats are casted in C `double`. Check your C compiler and your machine architecture for specific maximum and minimum values.

Example of floats: `123.234, -45.0, +34.1, 123E-4, -45.90e+23, 45.90e23` Example of numbers which are probably not recognized as floats: `+.345 234 e34`.

### 3.2.3   String as a Term

A term can be a *string*. Its C type in the Term structure is then `STRING`. These strings are standard strings, as commonly encountered in C. They use the same backslash notation as the one used in C. To insert a " in a string you must backslash it with a
.

Example of strings: `"This is a string $#%$^%$^ "`, `"This is another`
`string with a`
`" embedded and even a`
`n which will become new line."`  Example of non strings: `"This is a "`
`not a string because of the char " in it"`.

### 3.2.4   Symbol as a Term

A term can be a *symbol*. Its C type in the Term structure is then `TT_ATOM` or `ATOM` (except under Windows 95-NT). Symbols case may be changed automatically according to the option used (unless they are surrounded by vertical bars (`|`)). In some situations, (such as printing node names, or scripts names, the vertical bars are not printed around the symbol).

Example of symbols: `nil, t, |Foo Bar|, :BOO, bar`.  Example of non symbols: `123il, |#:|123|, $g45, @x, $32a, .asd`.

### 3.2.5   Variable as a Term

A term can be a *variable*. Its C type in the Term structure is then `VARIABLE`. Variables can be bound, in this case they are always bound to a Term (another variable for example). This depends of the binding environment in which they are considered.

Example of variables: `$g45, $x, $32a, @foo, @BoO`.

### 3.2.6   Variable List as a Term

A term can be a *variable list*. Its C type in the Term structure is then `LENV`. Variable lists are used for different purpose like when using `n-all` functions (see [Universal Quantification of Variables], §10.9, page 142), or in multi variable special actions (see [Multi Variable Special Action], §4.3.3, page 67).

Example of variable list: `($g45 $x $32a @foo @BoO)`.

### 3.2.7   Gtexpression as a Term

A term can be a *gtexpression*. Its C type is then `GTEXPRESSION`. See [General Temporal Expressions], §3.7, page 51. Gtexpression as terms are used in `test-and-set` or `apply-subst-in-gtexpr` functions.

Example of gtexpression: `(?  (foo a $x))`, `(achieve (position $x closed))`.

### 3.2.8   Gexpression as a Term

A term can be a *gexpression.* Its C type is then `GEXPRESSION`. See [General Expressions], §3.6, page 50. Gexpression as terms are used in `all` and `n-all` functions. From the reader, only `LEXPRESSION` are allowed, to be able to distinguish them from `TERM_COMP`.

Example of lexpression: `(& (foo $x b) (bar $x b))`.

### 3.2.9   Composed Term as a Term

A term can be a *composed term.* Its C type is then `TERM_COMP`. Beware, these terms look like lisp lists, but they are not (see below for a description of the lisp list). Composed terms are a list which first element is a function name and the rest of the element of the list are terms. The function appearing in this context are not always evaluable. Of course, composed terms can contain composed terms recursively.

Example of composed terms: `(+ 1 2)`, `(foo a b)`, `(l_list 1 2 3 4)`, `(cdr (.2 4 7 .))`.

### 3.2.10   Lisp List as a Term

A term can be a lisp list. Its C type is then `LISP_LIST`. Lisp lists are provided for upward compatibility with the Lisp version of OPRS. However, we do not encourage the use of Lisp functions and lispisms in OPRS. Therefore, to distinguish lisp lists from other lists like structure (such as composed terms), we use a different parenthesis scheme. The Lisp lists are surrounded with parenthesis followed or preceded with the dot char `.`. Moreover, when read by the parser, we only allow terms in the list (so you can still have lists of lists, etc...). However, the kernel uses this data structure to handle list of OP Instances, Goals, Intentions or other internal data structures when necessary. In this case the printed form is the name of the structure followed by the address of the object in memory. See [LISPLIST], §H.1, page 361 for more on this subject.

Example of lisp lists: `(.  2 4 7 .)`, `(.  2 (.  2 (.  2 4 7 .)  4 7 .)  4 7 .)`.

### 3.2.11   User Pointers as a Term

A term can be a user pointer. Its C type is then a `U_POINTER`. A user defined pointer is a pointer on a user created object which he wants to manipulate. For example, an evaluable function may return such pointer (like the pointer to a complex data structure), which can then be used by other user defined evaluable functions to access specific fields of the data structure, or by user-defined evaluable predicates or actions. The kernel will only consider the pointer itself, it will not consider the object pointed by the `U_POINTER`. Therefore, it will not free it, nor will it copy it or compare it with other similar object. Last, the comparison of `U_POINTER` is a pointer comparison. `U_POINTER` are written with

the hexadecimal printout format of C. U_POINTER can be read by the OPRS
Kernel and the X-OPRS Kernel. For this, use the 0x prefix and an hexadecimal
number. This can be useful if you want to consult the database or post a fact
or a goal with an expression containing a U_POINTER for which you know a
corresponding object exist. See [User Pointers], §10.10, page 142 for more on
this subject.

Example of user pointer: 0xff30, 0xA120.

### 3.2.12   Array of Integers as a Term

A term can be an array of integers. Its C type is then INT_ARRAY. Arrays of
integers are read by the various programs of the OPRS Development Environ-
ment using the [ and a matching ] square bracket. In this case, the type of the
first element determine the type of the whole array. If it is an INTEGER, then
the array is an INT_ARRAY, if it is a FLOAT, then the array is a FLOAT_ARRAY
(see below). Any subsequent element is properly casted if possible, if not it is
set to 0. The size of arrays read is the number of element read. Arrays can
be created/accessed/set with the appropriate evaluable functions (see [Array
Manipulation Evaluable Functions], §6.1.2, page 97) and actions (see [Array
Manipulation Actions], §7.7.1, page 116). Arrays can also be created/consulted
using the appropriate functions (see [Array Manipulation Functions], §G.1.5,
page 340).

Example of array of integers: [ 1 2 4 5 0 3 4], [1 3 4.5 5 7] (note that
the 4.5 will be casted in 4).

### 3.2.13   Array of Floats as a Term

A term can be an array of floats. Its C type is then FLOAT_ARRAY. Arrays of
floats are read by the various programs of the OPRS Development Environment
using the [ and a matching ] square bracket. In this case, the type of the first
element determines the type of the whole array. If it is an INTEGER, then the
array is an INT_ARRAY, if it is a FLOAT, then the array is a FLOAT_ARRAY (see
above). Any subsequent element is properly casted if possible, if not it is set
to 0.0. The size of arrays read is the number of element read. Arrays can
be created/accessed/set with the appropriate evaluable functions (see [Array
Manipulation Evaluable Functions], §6.1.2, page 97) and actions (see [Array
Manipulation Actions], §7.7.1, page 116). Arrays can also be created/consulted
using the appropriate functions (see [Array Manipulation Functions], §G.1.5,
page 340).

Example of array of floats: [ 1.5 4.2 4.0 5.888 0.0 3e4 3.1415], [1.0
3.0 4 5.123 7123.123] (note that the 4 will be casted in 4.0).

### 3.2.14   C List as a Term

A term can be a C list. Its C type is then OPRS_LIST. C lists are only used
when you want to return more then one value from an action, i.e. for a multi

variable special actions (see [Multi Variable Special Action], §4.3.3, page 67). These C lists are always lists of `Term *`. These lists look like composed terms, or Lisp Lists, but they are not. They are not readable, although they are, to some extend, printable.

   Example of C lists, as you could build one to return from a multi variable special action: `(1 2 3 nil foo (foo "asd" bar))`.

### 3.2.15   Other Objects as Term

There are other object which can be terms: `TT_FACT`, `TT_GOAL`, `TT_INTENTION`, `TT_OP_INSTANCE`. These objects are manipulated by the user but cannot be read from the parser as is.

## 3.3   Special Symbols

There are a number of predefined symbols in the OPRS Kernel. These symbols can be useful to the user to write evaluable functions (see [Evaluable Functions], §6, page 95) or actions (see [Using Action OPs], §7.7, page 114). They are defined in the file *'oprs-type-pub.h'*.

```
extern Symbol lisp_t_sym;        /* The lisp t symbol. */
extern Symbol wait_sym;          /* The :wait symbol. */
extern Symbol nil_sym;           /* The lisp nil symbol */
```

### lisp_t_sym                                    **Special Symbol**

   `extern Symbol lisp_t_sym` is the `T` symbol as returned by evaluable functions and actions upon success.

### nil_sym                                       **Special Symbol**

   `extern Symbol nil_sym` is the `NIL` symbol as returned by evaluable functions and actions upon failure.

### wait_sym                                      **Special Symbol**

   `extern Symbol wait_sym` is the `:WAIT` symbol as returned by actions when they have not completed their computation.

## 3.4   Frames and Binding Environments

Frames (also called Binding Environments) are a very important component of the unification mechanism. The reasons why this mechanism is important is of little interest to the end user, however, it is important that the user be able to "read" and "interpret" such frame constructs. One has to remember that frames give the binding of variables. Frames can be installed or not. When they are installed, the variables are bound to the values specified by the frame. Note that

they can be bound to a value specified by another frame. As a consequence, a variable cannot be bound in two different frames at the same time.

A frame looks like this:

```
{ installed-boolean ( variables-binding ) previous-frame }
```

Here is a real example commented:

```
{0                ; \r{it is not installed}
 ([{$Z->(Term *)NULL } >> C ]   ; \r{3 uninstalled bindings, $Z to C}
  [{$Y->(Term *)NULL } >> B ]   ; \r{$Y to B}
  [{$X->(Term *)NULL } >> A ] ) ; \r{$X to A}
 {                ; \r{The previous frame}
  1               ; \r{is installed,}
  ()              ; \r{but does not have any binding to install,}
  {}              ; \r{and point on the empty frame.}
 }
}
```

## 3.5   Properties

Properties are pairs: `(symbol term)`. Properties are used in OPs to handle user-defined information which can be relevant to subsequent computation in particular for meta level reasoning. For example, if you want to implement a priority mechanism, then you can define a priority slot in the properties of your OP and then write a meta level OP which, when necessary, will look in this slot and intend the OP with the appropriate priority. Note that we can have a term which is evaluated at run time, if the function is an evaluable function.

Example of properties: `(priority 12)`, `(decision-procedure t)`, `(importance (+ 4 $x))`.

## 3.6   General Expressions

Expressions are used to represent facts in OPRS. General expressions (gexpression) embed expressions and logical expressions (lexpression). Logical expressions are just a combination of general expressions using the standard logical operators.

General expressions (gexpression) are defined as follows:

gexpression := expression | lexpression lexpression := ( expr_logical_operator gexpression+ ) expression := ( predicate term* ) | (   ( predicate term* ) ) predicate := SYMBOL expr_logical_operator := & | V

Example of gexpressions:   `(foo a b)`, `(factorial 5 120)`, `(> 3 2)`, `(& (foo a $x) (bar $x 7))`, `(   (toto 45 (+ 3 4)))`.

## 3.7 General Temporal Expressions

Temporal expressions (in short texpression) are used to represent goals in OPRS. A texpression is a gexpression with a temporal operator. General temporal expressions (gtexpression) embed texpressions and logical temporal expressions (ltexpression). Logical temporal expressions are just a combination of general temporal expressions using the standard logical operators.

General Temporal Expressions (gtexpression) are defined as follows:
`gtexpression := texpression | ltexpression ltexpression := ( texpr_logical_operator gtexpression+ ) texpression := ( temporal_operator gexpression )` ×
`temporal_operator := ! | ? | ^ | # | % | => | > | achieve | test | wait | preserve | maintain | conclude | assert | retract texpr_logical_operator := & | V`

According to the temporal operators, the gexpression contained is treated differently.

The different temporal operators are the *achieve* operator, the *test* operator, the *wait* operator, the *passive preserve* operator, the *active preserve* operator, the *assert/conclude* operator and the *retract* operator.

Example of gtexpression: `(! (factorial 5 $x))`, `(wait (tank full))`, ×
`(? (> $x 5))`, `(test (! (position valve close)) (# (< (pressure-of tk1) 320)))`.

Note that not all gtexpressions have a defined associated semantic. We shall see examples of such gtexpression at the end of this section.

### 3.7.1 Achieve Operator

The achieve operator is represented with the symbol: `!`, or with the symbol `achieve` (in upercase or lowercase).

The semantic of this gexpression is to try anything "possible" to make the gexpression true. Either it is already true in the database, if not and if any OP is applicable to satisfy this goal, it will be attempted. If the OP executes successfully, then the goal is achieved.

Example: `(! (position valve close))` is true of a sequence of states in which the position of the valve is closed at the end. However, the kernel is unable to handle goals such as: `(& (! (position valve close)) (! (pressure tk1 200)))`, because it does not know how to handle this type of conjunction. However, a goal such as `(! (& (position valve close) (pressure tk1 200)))` may be achieve if `(& (position valve close) (pressure tk1 200))` is already true in the database. Last, `(& (! (position valve close)) (# (pressure tk1 200)))` is allowed.

### 3.7.2 Test Operator

The test operator is represented with the symbol: `?`, or with the symbol `test` (in upercase or lowercase).

The semantic of this gtexpression is to check if the gexpression is currently true. Either it is already true in the database, or if any OP is applicable to satisfy this goal, it will be attempted. If it is not true, or cannot be achieved by the execution of a OP, it is considered as failed.

Example: `(?  (position valve close))` is true if and only if the valve is currently closed. Note that this information may not be directly available in the database, but may be concluded from other information, such as the fact that two sensors agree on the fact that it is closed, or by calling an external function to check it. In both cases, a OP responding to this particular goal should be provided and would perform the appropriate computation. Note that as for the achieve goal, the kernel is unable to handle goals such as: `(& (?  (position valve close)) (?  (pressure tk1 200)))`, because it does not know how to handle this type of conjunction. However, a goal such as `(? (& (position valve close) (pressure tk1 200)))` may be achieved if `(& (position valve close) (pressure tk1 200))` is true in the database. In other words, the database is able to handle conjunction and disjunction, but not the OP retrieval mechanism. This problem can be solved using multi thread execution (by putting each goal on a separate thread).

### 3.7.3   Wait Operator

The wait operator is represented with the symbol: `^`, or with the symbol `wait` (in upercase or lowercase).

The semantic of this gtexpression is to wait until the gexpression becomes true. Either it is already true in the database, or the kernel waits until it becomes true by putting the intention or task, from which this goal comes from, to sleep. Note that this goal never fails... The intention or thread sleeps until it becomes true, so at worst, the intention sleeps forever.

Example: `(^ (position valve close))` will always return true. It is just a matter of when it will return. The kernel will suspend the thread executing this goal until the condition is satisfied, so it can never fail (but it can remain suspended for ever too). Waiting goal are often combined to make watchdog, such as: `(^ (|| (position valve close)) (elapsed-time (time) 10))`, which will wait until either the valve is closed or 10 seconds elapsed. Note that there are no way to know which condition satisfied, but usually it is just a matter of putting a test after this goal to check which condition is true or not.

### 3.7.4   Passive Preserve Operator

The preserve operator is represented with the symbol: `#`, or with the symbol `preserve` (in upercase or lowercase).

The semantic of this operator is to preserve the truth of the gexpression. This is a passive preservation, i.e. when the gexpression becomes false, automatically, this goal and presumably any conjunction of goals it is part of is considered as failed.

Example: The passive preserve operator is never used alone, it is always used in a conjunction such as in: `(& (!  (position valve close)) (# (pressure tk1 200)))` which will attempt to close the valve while checking that the pressure in `tk1` remains equal to `200`. Note that the passively preserved condition are checked continuously (as new events are received by the system). Moreover, if the preserved condition fails, the conjunction fails. It is then up to the user to figure out what went wrong and take any appropriate action accordingly.

### 3.7.5  Active Preserve Operator

The active preserve operator is represented with the symbol: `%`, or with the symbol `maintain` (in upercase or lowercase).

The semantic of this operator is to actively preserve the truth of the gexpression. This is an active preservation, i.e. when the gexpression becomes false, the system tries automatically to re-achieve it by invoking a OP which has this goal in its invocation part. When the condition is failed, the execution thread in which this goal appears is stopped and the system will try to reachieve the condition.

Example: The active preserve operator is never used alone, it is always used in a conjunction such as in: `(& (!  (position valve close)) (% (pressure tk1 200)))` which will attempt to close the valve while checking that the pressure in `tk1` remains equal to `200`. Note that the passively preserved condition are checked continuously (as new events are received by the system). Moreover, if the preserved condition fails, the kernel will attempt to reestablish it by calling a OP which match this active preserve goal. If all attempts to reestablish the fail condition fail, then the conjunction is failed.

### 3.7.6  Assert/Conclude Operator

The conclude operator is represented with the symbol: `=>`, or with the symbol `conclude`, or with the symbol `assert` (in upercase or lowercase).

When executed, this goal has for effect to assert the gexpression in the database. To do so, the gexpression is transformed as a fact and is pushed into the kernel (and can lead to OP execution). This goal always succeeds.

Example: `(=> (position valve close))`. One can also put more then one conclude or retract goal in a conjunction: `(& (=> (position valve close)) (=> (pressure tk1 200)) ( > (ALARM))))`.

### 3.7.7  Retract Operator

The retract operator is represented with the symbol:  `>`, or with the symbol `retract` (in upercase or lowercase).

When executed, this goal has for effect to retract the gexpression from the database. It always succeeds.

Example: `( > (position valve close))`. One can also put more then one conclude/assert or retract goal in a conjunction: `(& ( > (position valve close)) ( > (pressure tk1 200)) ( > (ALARM)))`.

## 3.8 General Meta Expressions

A general meta expression (gmexpression) is defined as follows:

Gmexpressions are mainly used in *invocation part* and other OP text fields. There is no real reason for them to exist under this form, and they are provided, as is, more for upward compatibility with SRI Lisp OPRS than for real syntactic reasons. The parser has been modified to allow directly gexpressions and gtexpressions. The parser does nore recognizes the old `*FACT` and `*GOAL` markers (see [Principal Differences with SRI OPRS], §**??**, page **??**).

`gmexpression := mexpression | lmexpression lmexpression := ( mexpr_logical_operator gmexpression+ ) mexpression := gexpression | gtexpression  mexpr_logical_operator := AND | OR | NOT`

### 3.8.1 FACT Meta Expressions

The FACT Meta Expressions are used to designate a gexpression in the OP applicability fields.

Example of FACT Meta Expressions: `(FOO a b)`, `(> 4 3)`, `(foo 32 "this is a string")`, `(elapsed-time (time) 32)`.

### 3.8.2 GOAL Meta Expressions

The GOAL Meta Expressions are used to designate a gtexpression in the OP applicability fields.

Example of GOAL Meta Expressions: `(?  (bar a b))`, `(!  (factorial $x $y))`, `(?  (> 3 4))`, `(^ (elapsed-time (time) 10))`.

## 3.9 Facts

Facts are gexpressions concluded in the system. For example, the effects part of a OP is list of Texpressions (only conclude and retract operator). Each of the concluded Gexpressions (only Expressions) is concluded in the system as a new fact. However, facts contain much more information than just the gexpression from which they originate. For example they keep track of various dates, like their creation date, or the date at which they have been completely parsed by the applicable OP mechanism (see [Fact and Goal Manipulation Functions], §G.1.6, page 341).

Facts, as opposed to goals, are not linked in any way to the procedure, intention or external module they come from. Facts are perfectly anonymous in the sense that they are just a piece of information which is concluded in the database and which may start some procedures. However, if a fact is a message,

then its data structure stores the module from which it originated. Nevertheless, for any fact or message, the success or failure of the procedures applicable because of them is of no interest to anybody (i.e. any OP or intention)...

Facts cannot contain any unbound variable. They would be meaningless as these facts are not associated with any binding environment.

## 3.10 Messages

Messages are facts. They are called messages because they come from "outside" (presumably from a different agent such as another OPRS Kernel or an external program such as a monitor). Moreover, they usually come on the Message Passer socket. As soon as they are received by the OPRS Kernel, they are treated as Facts. When received, the message contains the name of the sender, and if the appropriate trace flag is on, a message on the screen advises you of the arrival of the message as well as of the name of the sender. This information is kept in the fact which will be created from the message, and can be retrieved with the appropriate access function (`fact_sender`, see [Fact and Goal Manipulation Functions], §G.1.6, page 341).

## 3.11 Goals

Goals (as opposed to facts) are linked to an already existing and executing procedure and intention (the intention in which this procedure executes). They can contain unbound variables, which are often used to return values. For example, when the goal: (! (factorial 4 $X)) is posted, you most likely expect the $X to be bound to 24 upon success of this goal.

Although it is technically possible, it is contrary to OPRS philosophy to post a goal in a kernel. Goals should only come from the execution of procedures. An agent usually does not directly give a goal to another agent. It merely passes a message containing a request to achieve a goal, in which case a message, i.e. a fact, is passed to the OPRS Kernel. Moreover, if you directly post a goal from the "keyboard" you may lose it, if the applicable OP selection mechanism does retain its OP Instance for execution (if the goal is posted by the kernel, it will be automatically reposted until failure or success).

# Chapter 4

# OP Syntax and Semantic

OPs must follow a very rigorous syntax. The OP Editor and the OPRS Kernel will enforce this syntax as much as possible. As a matter of fact, any OP properly parsed by the OP Editor should be parsed and compiled (for the syntax part) without any problem by the OPRS Kernel.

OPs are composed of two parts. A number of text fields which define the "declarative" part of the procedure, and the Graph part (in some OPs, it is just an action), which defines the operative or the procedural part of the OP. The declarative part is composed of various fields. Three of these fields are controlling the applicability of the OP, the *invocation part*, the *context part*, and the *setting part*. To be applicable, these three parts must be true, but in a different way.

## 4.1 OP Applicability Fields

The applicability fields are the field controlling the applicability of the OP. They are: the invocation part, the context part, and the setting part. For a OP to be applicable, these three fields must be true in the same binding environment. However, each field has a particular semantic, which should be respected to take full advantage of OPRS OP triggering mechanism. The invocation part is a logical expression describing the *events* that must occur for the OP to be executed. Usually, these consist of some *changes* in system goals (in which case, the OP is invoked in a goal-directed fashion) or system beliefs (resulting in data-directed or reactive invocation), and may involve both. The context part is a logical expression specifying those conditions that must be true of the current state for the OP to be executed. The setting part is a logical expression specifying conditions that must be true, but for which the truth value can be determined before the system runs (after the database and the OP have been loaded). In other words, the conditions appearing in the setting part are not run-time dependent. One can see that this separation between the invocation part, the context part, and the setting part is only justified by performance and

efficiency reasons.

### 4.1.1  Invocation Part

The Invocation part is a General Meta Expression (see [General Meta Expressions], §3.8, page 54). It specifies which facts, goals or any combination of both can trigger the applicability of a OP. Keep in mind that only the facts and goals specified in the invocation part can trigger the OP applicability. In other words, a OP is considered relevant, i.e. considered for applicability, only if the system has a new goal or a new fact which is specified in the invocation part of this OP. Note that for this reason there is no point at all in putting an evaluable predicate in the invocation part of a OP.

Example of invocation part:  `(ALARM)`, `(position $x BP)`, `(OR (ALARM)` `(FIRE))`, `(AND (ALARM) (OVERPRESSURIZED $TANK))`.

### 4.1.2  Context Part

The Context part is either a General Meta Expression (see [General Meta Expressions], §3.8, page 54), or nothing.

It contains information which must be true for the OP to be applicable, but the difference with the invocation part is that the occurrence of this information does not trigger the OP applicability.

Example of context part:  `(POSITION $T $POS)`, `(> $X 245)`, `(OR (FOO` `$x) (BAR $Y))`, `(AND (POSTION $T OPEN) (STATUS $T GOOD))`.

### 4.1.3  Setting Part

The Setting part is either a General Meta Expression (see [General Meta Expressions], §3.8, page 54), or nothing.

It contains information which must be true to make the OP applicable, but this information is not supposed to change over time. In other words, the truth value of this part (or more accurately, the frames in which this gmexpression is true) can be defined at OP compile time (providing the fact database is loaded).

Example of setting part:  `(CONNECTED $PIPE $TANK)`, `(ASSOCIATED-SENSORS` `$S1 $S2)`, `(OR (ASSOCIATED-SENSORS $S1 $S2) (ASSOCIATED-SENSORS $S2` `$S1))`, `(AND (ASSOCIATED-XDCR $TK1 $XDR1) (ASSOCIATED-XDCR $TK2 $XDR2))`.

## 4.2  OP Other Fields

Other fields used in OP include:

### 4.2.1  Effects Part

The Effects part is either a list of General Temporal Expressions (see [General Temporal Expressions], §3.7, page 51), or nothing.

It contains a list of conclude or retract texpressions which are concluded or retracted upon successful execution of the OP.

Example of effects part: `()`, `((=> (POSITION $T $POS)))`, `(( > (POSITION $T $POS)))`, `((=> (POSITION $T1 $POS1)) (=> (POSITION $T2 $POS2)) (=> (POSITION $T3 $POS3)))`, `(( > (FOO $x)) (=> (FOO $y)) (=> (BAR $X $Y)))`.

### 4.2.2 Properties Part

Important properties of the OP are represented in the properties slot. The Properties part is either a list of properties (see [Properties], §3.5, page 50), or nothing. It contains a list of properties which can be used by the appropriate predicates (see [OP Instance Related Evaluable Predicates], §5.8.1, page 90 and functions see [OP Instance Related Evaluable Functions], §6.1.3, page 98). Properties are usually used by meta level OP (see [OP Properties], §10.1, page 135) to retrieve specific information about OPs.

Example of properties part: `()`, `((DECISION-PROCEDURE T))`, `((ID FOO))`, `((RESOURCE-USED PLATINE))`, `((SPEED (evaluate-speed $x $y $z)) (RELIABILITY 23))`, `((PRIORITY $X))`.

### 4.2.3 Documentation Part

The Documentation is a string. Its purpose is to document the OP.

Example of documentation part: `""`, `"This string document the OP."`, `"You can put variable names such as $x in documentation strings."`, `"This OP will kill its own intention after 10 seconds. 10, 9, 8, 7, 6, 5, 4, 3, 2, 1... Ophbooum..."`, `"This is a test OP. Please ignore."`.

## 4.3 Execution Part

There are two different types of OPs in OPRS. Action OPs are the basic or low level actions of the system, and Graph OPs are the real procedures or plans of the system.

### 4.3.1 Graph OP

The body of a OP is represented as a graphic network and can be viewed as a plan or plan schema. Each arc of the network is labelled with a goal.

A typical example of a OP is given in Figure 4.1, which describes a procedure to isolate close a valve in the truck demo presented in See [Truck Loading Example], §23.1, page 273. The invocation part describes useful conditions for this OP. In this case, the OP is considered useful whenever the system acquires the goal to close or open a valve, provided the various facts given in the context part are true. (In determining the truth value of the invocation part, some of the variables appearing in the invocation part will be bound to specific identifiers. Indeed, in this case, all the time out values will be so bound.)

## Move Valve

**INVOCATION:**
(! (POSITION VALVE $X))

**CONTEXT:**
(AND (ALARM YES)
     (DELAY TWO-GOOD $DEL-2TBS)
     (DELAY ONE-GOOD $DEL-1TB))

**EFFECTS:**
((=> (POSITION VALVE $X)))

**DOCUMENTATION:**
"This KA tries to put the valve in position $x.
It waits ($del-2tbs) time units if the two
sensors are good, or ($del-1tb) time units
if only one is good.
After this time, it shutdowns if the trusted
sensor(s) is not in good position. "

**START**

(! (POSITION SWITCH-VALVE $X))

**N1**

(! (= @SWITCH-TIME (CURRENT (TIME))))

**N2**

(? (& (TYPE TB $TB1)
      (STATUS $TB1 FAILED)))

**T NS F**

(^ (V (ELAPSED-TIME
        @SWITCH-TIME $DEL-2TBS)
      (& (POSITION SENSOR1 $X)
         (POSITION SENSOR2 $X))
      (& (TYPE TB $TB1)
         (STATUS $TB1 FAILED))))

**N4**

(? (& (TYPE TB $TB1)
      (STATUS $TB1 FAILED)))

**T N3 F**

(^ (& (V (ASSOCIATED-TB $TB1 $TB2)
         (ASSOCIATED-TB $TB2 $TB1))
      (V (ELAPSED-TIME
           @SWITCH-TIME $DEL-1TB)
         (POSITION $TB2 $X))))

(? (& (POSITION SENSOR1 $X)
      (POSITION SENSOR2 $X)))

**T N6 F**

(! (SHUTDOWN (TWO-SENSORS $DEL-2TBS)))

**END1**

(^ (& (V (ASSOCIATED-TB $TB1 $TB2)
         (ASSOCIATED-TB $TB2 $TB1))
      (V (ELAPSED-TIME
           @SWITCH-TIME $DEL-1TB)
         (POSITION $TB2 $X))))

**N12**

(? (POSITION $TB2 $X))

**T N7 F**

(! (SHUTDOWN (TRUSTED $TB2 $DEL-1TB)))

**END0**

Figure 4.1: Another Example of a OP

# Move Valve

**INVOCATION:**
(! (POSITION VALVE $X))

**CONTEXT:**
(AND (ALARM YES)
     (DELAY TWO-GOOD $DEL-2TBS)
     (DELAY ONE-GOOD $DEL-1TB))

**EFFECTS:**
((=> (POSITION VALVE $X)))

**DOCUMENTATION:**
"This KA tries to put the valve in position $x.
It waits ($del-2tbs) time units if the two
sensors are good, or ($del-1tb) time units
if only one is good.
After this time, it shutdowns if the trusted
sensor(s) is not in good position. "

START
(! (POSITION SWITCH-VALVE $X))

N1
(! (= @SWITCH-TIME (CURRENT (TIME))))

N2
(? (& (TYPE TB $TB1)
      (STATUS $TB1 FAILED)))

T N5 F
(^ (V (ELAPSED-TIME
        @SWITCH-TIME $DEL-2TBS)
      (& (POSITION SENSOR1 $X)
         (POSITION SENSOR2 $X))
      (& (TYPE TB $TB1)
         (STATUS $TB1 FAILED))))

N4
(? (& (TYPE TB $TB1)
      (STATUS $TB1 FAILED)))

T N3 F
(? (& (POSITION SENSOR1 $X)
      (POSITION SENSOR2 $X)))

(^ (& (V (ASSOCIATED-TB $TB1 $TB2)
         (ASSOCIATED-TB $TB2 $TB1))
      (V (ELAPSED-TIME
           @SWITCH-TIME $DEL-1TB)
         (POSITION $TB2 $X))))

(^ (& (V (ASSOCIATED-TB $TB1 $TB2)
         (ASSOCIATED-TB $TB2 $TB1))
      (V (ELAPSED-TIME
           @SWITCH-TIME $DEL-1TB)
         (POSITION $TB2 $X))))

T N6 F
(! (SHUTDOWN (TWO-SENSORS $DEL-2TBS)))

END1

N12
(? (POSITION $TB2 $X))

T N7 F
(! (SHUTDOWN (TRUSTED $TB2 $DEL-1TB)))

END0

Figure 4.2: An Example of a OP

Figure 4.1 presents an older version of this OP. The OP body describes what to do if the OP is chosen for execution. Execution begins at the `start` node in the network, and proceeds by following arcs through the network. Execution completes if it reaches a finish node (a node with no exiting arcs). If more than one arc emanates from a given node, any one of the arcs emanating from that node may be traversed. To traverse an arc, the system must either (1) determine from the database that the goal has already been achieved or (2) find a OP (procedure) that achieves the goal labelling that arc. For example, to cross the arc emanating from the start node requires either that the system has already flipped the switch or that some OP to do it be successfully executed. If the system fails to go through an arc emanating from some node, other arcs emanating from that node may be tried. If, however, the system fails to achieve any of the goals on arcs emanating from the node, the OP as a whole will fail. For instance, since only one arc emanates from the `START` node in Figure 4.2, if all attempts to flip the switch fail, this procedure for opening or closing the valve will also fail.

## 4.3.2   New Graph OP Construction

Since release 1.1 of the OPRS Development Environment, there are some new graph construction allowed: the "IF-THEN-ELSE" node; to express conditional branching, and the "split" and "join" node; to express parallel execution. These two new constructions are presented in the next sections.

### IF-THEN-ELSE Node

The "IF-THEN-ELSE" node was introduced in OPRS to simplify the construction of OPs. To illustrate this we show a OP written the old way, and then the same one with the new way.

The OP shown on Figure 4.3 computes factorial recursively. One can see that the OP starts with a test to check if `$n` is either greater than `1` or smaller.

The OP shown on Figure 4.4 performs exactly the same operation: computes factorial recursively. However, one can see that the OP `S1` and `S3` nodes have been replaced by a `NO` node which is surrounded by two nodes labeled `T` and `F`. This node `NO` is a "IF-THEN-ELSE" node. It works as follows: when an edge as an "IF-THEN-ELSE" outgoing node, the goal labeling this edge never fail. In other words, the transition is always possible. If the goal is achieved (in our example if the goal `(?  (<= $n 1))` is true), then the execution proceeds from the `T` node, otherwise, it proceeds from the `F` node. This construction is in fact very similar to the old *test-and-set* construction (see [Miscellaneous Actions], §7.7.1, page 121).

Figure 4.5 and Figure 4.6 show another example of an old form and a new form of procedure.

Apart from the visual aspect, this "IF-THEN-ELSE" construction is also more efficient as you will evaluate less goals (1 goal instead of 1.5 goal in average).

**Recursive Factorial**
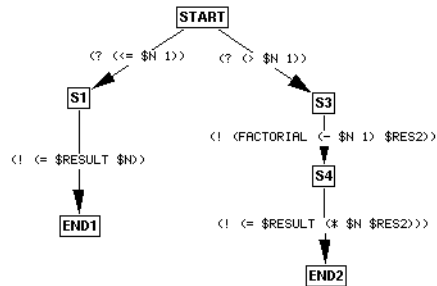
**INVOCATION:**
(! (FACTORIAL $N $RESULT))



**PROPERTIES:**
((RECURSIVE T))

**DOCUMENTATION:**
"This KA computes the Factorial of $n in a recursive manner.
Note the RECURSIVE T property which will be used by the
Meta KA to decide which KA to intend."

Figure 4.3: A OP to Compute Factorial Recursively (Old if-then-else Form).
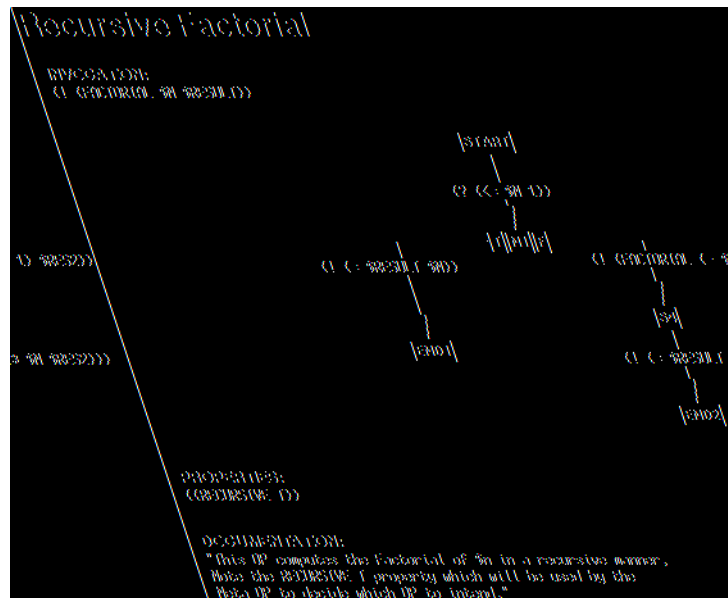


Figure 4.4: A OP to Compute Factorial Recursively (New if-then-else Form).

**Iterative Factorial**
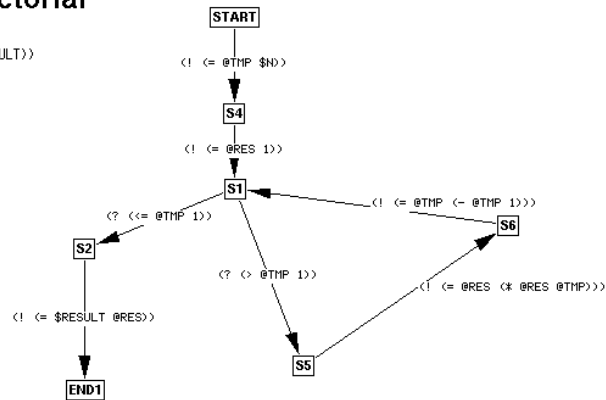
INVOCATION:
(! (FACTORIAL $N $RESULT))

```
                                              START

                                  (! (= @TMP $N))

                                        S4

                                  (! (= @RES 1))

                                        S1                    (! (= @TMP (- @TMP 1)))

            (? (<= @TMP 1))                                              S6

        S2

                            (? (> @TMP 1))            (! (= @RES (* @RES @TMP)))

    (! (= $RESULT @RES))

                                                  S5
        END1
```

PROPERTIES:
((RECURSIVE NIL))

DOCUMENTATION:
"This KA  computes the Factorial of $n and unifies the result with $result.
 It uses an iterative algorithm operating on local variables.
Note the RECRUSIVE NIL property which will be used by the
Meta level KA."

Figure 4.5: A OP to Compute Factorial Iteratively (Old if-then-else Form).
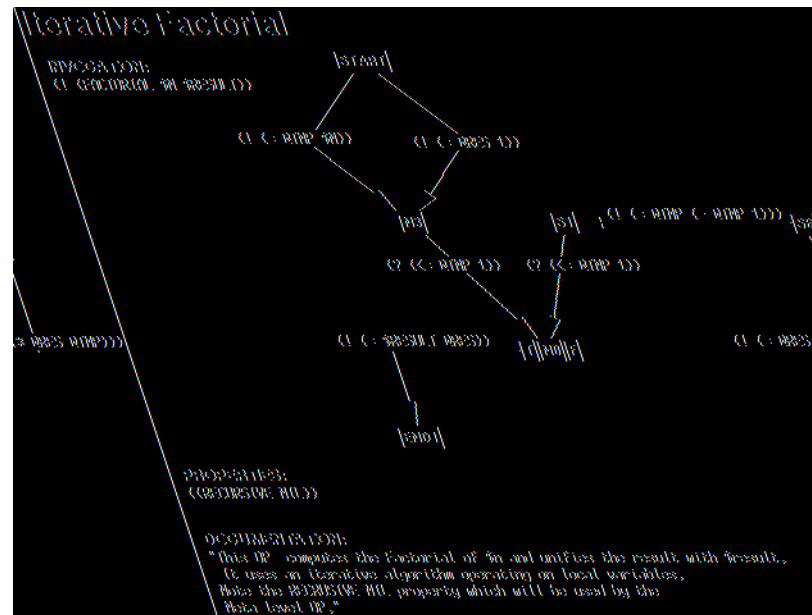


Figure 4.6: A OP to Compute Factorial Iteratively (New if-then-else Form).
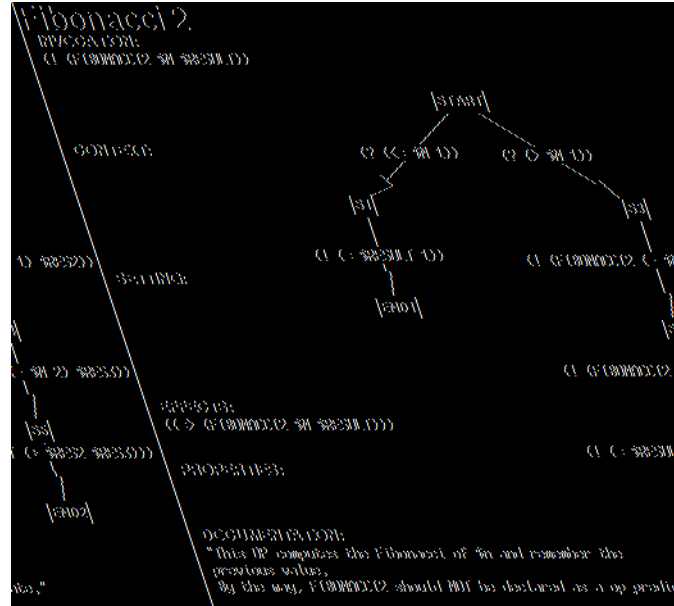
Figure 4.7: A OP to compute Fibonacci (without parallelism).

**Split and Join Node**

The split and join node is a construction which is linked to the use of parallel execution (see [Parallel Execution of OPs in OPRS], §8, page 127).

Here also we illustrate this new construction with concrete examples. Figure 4.7 shows a OP which computes Fibonacci. In this particular case, the two recursive calls can be done in parallel. Figure 4.8 shows an example of such construct. In this particular example we mixed the "IF-THEN-ELSE" construction with the split node. The F node of the N0 "IF-THEN-ELSE" node is a split node (this is represented with the thick bottom of the node). Similarly, the S4 node is a join node. Basically, a split node split as many thread as it has outgoing edges, and a join node merge as many thread as they are ingoing edges. See [Parallel Execution of OPs in OPRS], §8, page 127 for more on this subject.

## 4.3.3 Action OPs

Some OPs have no bodies. These are the primitive OPs of the system and are associated to a primitive action that is directly performable by the system. Clearly, execution of any OP must eventually reduce to the execution of sequences of primitive OPs (unless, of course, each of the subgoals of the OP has already been achieved).

There are two types of action OPs. They correspond to 2 types of behavior. With Standard Action, you expect the performed action to return a symbol
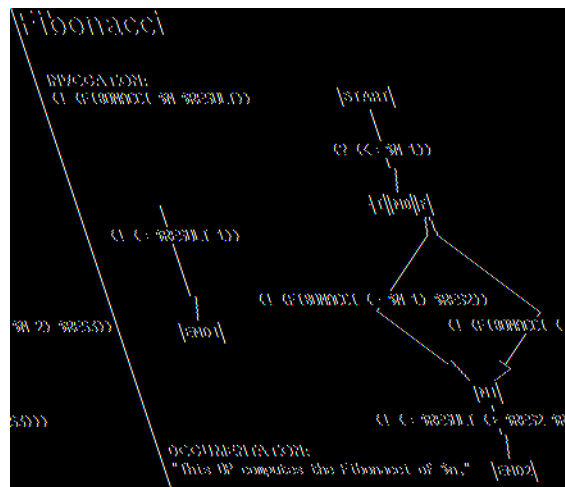
Figure 4.8: A OP to compute Fibonacci (with parallelism).
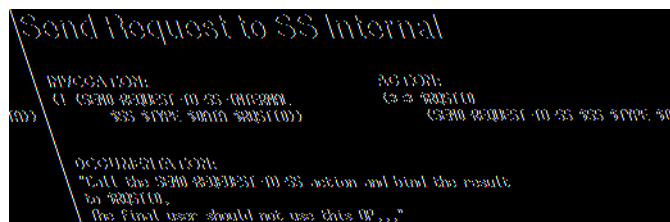


Figure 4.9: A Standard Action OP

(usually `T`, `NIL` or `:WAIT`, see [Important Variables], §G.1.2, page 338, and [Special Symbols], §3.3, page 49) to inform the system of the success or the failure of the execution. However, in many application one want to grab the value(s) returned by the action and eventually bind it/them or unify it/them with a variable or a list of variables, this is the role of the Special Action and Multi Variable Special Action. Last, not least, in both cases, the action can return before completion (there is a special value `:WAIT` for this purpose) and will be automatically called again after the system has checked for new applicable OPs (see [Action Slicing], §10.11, page 143).

### Standard Action

The standard action is a *Composed Term* (see Figure 4.9). The function part of this composed term must be declared as an action (see [How to Define your Own Actions], §7.7.2, page 122). The value returned by the evaluation of this
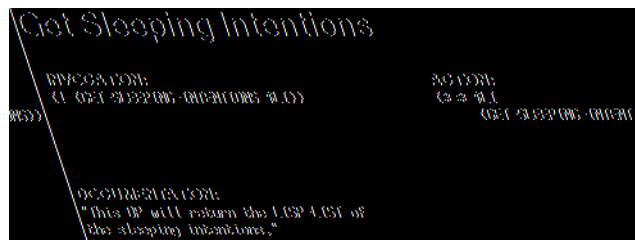
Figure 4.10: A Special Action OP



Figure 4.11: A Multi Variable Special Action OP

function is meaningful. It must be a pointer to term, and this term will be freed by the caller. If it returns the term symbol `:wait`, the function has not completed its execution and it should be called again later. If it returns the term symbol `NIL`, then the action is considered as failed and the OP failed the goal it was working on. Any other term value returned is considered as a success, and the action OP is successful.

Example of standard action: `(print $x)`, `(send-message $x $y)`, `(init-robot)`, `(goto-location $x $y)`.

### Special Action

Special actions are provided to allow the binding of the result from the action evaluation. Their syntax is different (see Figure 4.10). A special action must be of the following form: `(*=* <variable> <composed-term>)`.

Here also, the function part of the composed term must be an action (see [Using Action OPs], §7.7, page 114) or an evaluable function (see [Evaluable Functions], §6, page 95). The only difference with standard action is that the result of the evaluation is unified with the variable. The success or the failure of the OP itself depends on the result of the unification. If the unification succeeds, then the OP is considered as successful, it is a failure otherwise. Note that if the action returns the term symbol: `:WAIT`, the action will be called again later.

Example of special action: `(*=* $y (read))`, `(*=* $status (init-system $y))`, `(*=* T (confirm $y))`.

### Multi Variable Special Action

New in version 1.1, one can bind a list of terms/variables to a list of terms returned by an action (see Figure 4.11).

The newly allowed form for special action is thus:  (*=* (<variable>*)
<composed-term>). The old form remains valid. The behavior is exactly the
same, except that the action is expected to return a list of terms. Each term will
be unified against the corresponding term/variable in the list. If the number of
terms differ (i.e. there are two variables in the list and the action returns three
terms), or if one of the unification fails, the action fails. Note that if the action
returns the term symbol :WAIT, the action will be called again later.

Example of multi variable special action: (*=* ($x $y $theta) (position
robot)), (*=* ($result $status) (init-system $y)).

Example of a form which is not a multi variable special action (it must be a
list of variables, not terms): (*=* (T NIL T) (confirm-three-responses $y
$z $w)).

One can find an example of an action returning a list of terms in the file:
'user-action.c'.

### 4.3.4   Text OPs

Text OPS have been introduced in recent versions of OPRS to allow user to write
OP as standard programing language. Text OPs language provide standard if-
then-else, while, do-while, goto, break construct, as well as parallel operator.
See [Grammar Used in the OPRS Development Environment], §K, page 375 for
a description of this grammar. Like action OP and graphic OP, they can be
traced.

**General Presentation of the Text OPs**

The best way to illustrate text OPs is to study some examples.

The following file presents text OPs which implement factorial.

```
;;;
;;; File: /users/oprs/C/ACS/oprs-1.3/data/fact-text.opf
;;; $Id: fact-text.opf,v 1.1 2003/10/21 08:12:20 felix Exp $
;;; Date: Wed Feb 15 18:02:01 1995
;;; Host: cascade
;;; OP Editor Version : 1.3.1 Beta
;;; Compiled on cascade (Wed Nov 16 12:06:21 MET 1994)
;;; OP File Format: 2
;;;
(
(defop FOO
:invocation (|| (foo) (bar))
:action (PRINT $N)
:documentation "This OP prints its term."
)
(defop BAR
:invocation (& (foo) (bar))
```

```
:action (PRINT $N)
:documentation "This OP prints its term."
)
(defop PRINT
:invocation (! (PRINT $N))
:action (PRINT $N)
:documentation "This OP prints its term."
)
(defop |Iterative Factorial|
:invocation (! (FACTORIAL $N $RESULT))
:body
(;;; init in parallel
 (// ((! (= @TMP $N))
     )
     ((! (= @RES 1))
     )
 )
 (WHILE (? (> @TMP 1))
    (! (= @RES (* @RES @TMP)))
    (! (= @TMP (- @TMP 1)))
 )
 (! (= $RESULT @RES))
)
:properties ((RECURSIVE NIL))
:documentation "This is another example"
)
(defop |Meta Factorial|
:invocation (APPLICABLE-OPS-GOAL $GOAL $X)
:context (EQUAL (LENGTH $X) 2)
:body
((IF (? (PREFER-ITERATIVE))
    ;;; Look for the one which has property recursive
    (IF (? (PROPERTY-P RECURSIVE (FIRST $X)))
       (! (INTENDED-OP (SECOND $X)))
     ELSE
       (! (INTENDED-OP (FIRST $X)))
    )
  ELSEIF (? (PREFER-RECURSIVE))
    ;;; Look for the one which has property recursive
    (IF (? (PROPERTY-P RECURSIVE (FIRST $X)))
       (! (INTENDED-OP (FIRST $X)))
     ELSE
       (! (INTENDED-OP (SECOND $X)))
    )
  ELSE
    ;;; We do not have any preference...
```

```
    ;;; Choose randomly
    (! (INTENDED-OP (SELECT-RANDOMLY $X)))
 )
)
:properties ((DECISION-PROCEDURE T))
:documentation "This Meta OP chooses which Factorial OP to intend according
to the presence or not of the recursive property on the
applicable OPs.
Do not use this Meta OP in other applications."
)
(defop |Meta Intend|
:invocation (! (INTENDED-OP $X))
:action (INTEND-OP $X)
:documentation "The simplest way to intend an applicable OP
($X a Op Instance LISP_CAR)."
)
(defop |Print Factorial|
:invocation (! (PRINT-FACTORIAL $X))
:body
(;;; Example of simple sequence
 (! (FACTORIAL $X $RES))
 (! (PRINT $RES))
)
:documentation "Print factorial example using text op."
)
(defop |Recursive Factorial|
:invocation (! (FACTORIAL $X $N))
:body
((IF (? (> $X 1))
    (! (FACTORIAL (- $X 1) $N2))
    (! (= $N (* $X $N2)))
  ELSE
    (! (= $N 1))
 )
)
:properties ((RECURSIVE T))
:documentation "This is an example"
)
)
```

The OP on Figure 4.12 is the OP Editor representation of the `Meta Factorial`
presented above.

The following file presents text OP which implement fibonacci.

```
;;;
;;; File: /users/oprs/C/ACS/oprs-1.3/data/fib-text.opf
;;; $Id: fib-text.opf,v 1.1 2003/10/21 08:12:20 felix Exp $
```

Figure 4.12: Meta Factorial Text OP

```
;;; Date: Wed Feb 15 18:00:45 1995
;;; Host: cascade
;;; OP Editor Version : 1.3.1 Beta
;;; Compiled on cascade (Wed Nov 16 12:06:21 MET 1994)
;;; OP File Format: 2
;;;
(
(defop PRINT
:invocation (! (PRINT $N))
:action (PRINT $N)
:documentation "This OP prints its term."
)
(defop |Fibonacci|
:invocation (! (FIBONACCI $N $RESULT))
:body
((IF (? (< $N 1))
    (! (= $RESULT 1))
  ELSE
    ;;; $N is greater than 1
    ;;; compute the two values of fibonacci in parallel
    (// ((! (FIBONACCI (- $N 1) $RES2))
        )
        ((! (FIBONACCI (- $N 2) $RES3))
        )
    )
    ;;; Implicit rendez-vous
    ;;; Compute the final result
    (! (= $RESULT (+ $RES2 $RES3)))
 )
)
:documentation "This is a // fiboncci"
)
(defop |Print Fibonacci|
:invocation (! (PRINT-FIBONACCI $X))
:body
((! (FIBONACCI $X $N))
 (! (PRINT $N))
)
:documentation "Print fibonacci text op."
)
)
```

   The OP on Figure 4.13 is the OP Editor representation of the `Fibonacci`
presented above.

   Text OPs are editable with the op-editor, but can also be created with your
preferred text editor. In fact, the op-editor does not store any position, nor text

Figure 4.13: Fibonacci Text OP

filling information about these OPs and will not allow you to move the fields of a text OP.

One can mix graph op and text OP in the same OP file. Text OPs are put in OP file with the same extension file *'.opf'*. As long as all the OPs in a OP files are text OPs you can edit this file emacs or your prefered text editor. However if at least one OP in a OP file is a graphic OP, then you can still grab or edit the text OP in this file, but it is more difficult, and it is at your own risk.

The body field of a text OP cannot be empty... it must at least contain the empty instruction list, i.e. ().

Actions OP created under the OP Editor are by default considered as graphic OP.

There is one major advantage in editing text OP in the OP Editor, the lexical and syntaxic parsing is continuously done. Otherwise, if you edit them under Emacs, you will have to wait until you load them in the OP-Editor or OPRS to check them.

Text OP selected for graphic trace, show up on the screen, and you can see the body, the other interesting fields, adn hilighted the currently posted goals.

### IF-THEN-ELSE Instruction

The `IF-THEN-ELSE` instruction and `IF-THEN-ELSEIF...` instructions are standard branching operations.

The syntax is the following:

```
if_inst: ( IF if_part_inst )

if_part_inst: gtexpr list_inst
        | gtexpr list_inst ELSE list_inst
        | gtexpr list_inst ELSEIF if_part_inst
```

list_inst is a possibly empty sequence of instruction.
here is an example of use:

```
(IF (? (PREFER-ITERATIVE))
    ;;; Look for the one which has property recursive
    (IF (? (PROPERTY-P RECURSIVE (FIRST $X)))
        (! (INTENDED-OP (SECOND $X)))
     ELSE
        (! (INTENDED-OP (FIRST $X)))
    )
  ELSEIF (? (PREFER-RECURSIVE))
    ;;; Look for the one which has property recursive
    (IF (? (PROPERTY-P RECURSIVE (FIRST $X)))
        (! (INTENDED-OP (FIRST $X)))
     ELSE
        (! (INTENDED-OP (SECOND $X)))
    )
  ELSE
    ;;; We do not have any preference...
    ;;; Choose randomly
    (! (INTENDED-OP (SELECT-RANDOMLY $X)))
    (! (PRINT "Intending randomly"))
 )
```

### WHILE Instruction

The `WHILE` instruction is a standard while operations.
    The syntax is the following:

```
while_inst: ( WHILE gtexpr list_inst )
```

list_inst is a possibly empty sequence of instruction.
    here is an example of use:

```
(WHILE (? (> @TMP 1))
    (! (= @RES (* @RES @TMP)))
    (! (= @TMP (- @TMP 1)))
 )
```

One can use the `BREAK` instruction to make a local exit of the while loop.

### DO-WHILE Instruction

The `DO-WHILE` instruction is a standard do-while.
    The syntax is the following:

```
do_inst: ( DO list_inst WHILE gtexpr )
```

list_inst is a possibly empty sequence of instruction.
    here is an example of use:

```
(DO
 (! (FOO $X))
 (// ( (! (BAR $Y)) (! (BOO $Z)))
     ( (! (BAR $A)) (! (BOO $W)))
     ( (! (BAR $B)) (! (BOO $U)))
 )
 WHILE (? (> (X-OF $Y) 35)))
```

Here also, one can use `BREAK` to exit the loop.

### Parallel Instruction

The `//` instruction executes all the branches in parallel.
    The syntax is the following:

```
par_inst: ( // <par_list_inst>* )
```

```
par_list_inst: ( list_inst )
```

list_inst is a possibly empty sequence of instruction.
    here is an example of use:

```
 (// ((! (= @TMP $N))
     (! (=@FOO @TMP))
     )
     ((! (= @RES 1))
     )
 )
```

### GOTO-LABEL Instruction

The `GOTO-LABEL` instruction is a standard goto operations.
    The syntax is the following:

```
goto_inst: GOTO id
```

```
label_inst: LABEL id
```

here is an example of use:

```
(;;; comment
 LABEL test
 (! (foo @x))
 (! (= @x (- @x 1)))
 (IF (? (> @x 0))
     GOTO test
  ELSE
     GOTO fin)
  LABEL fin
)
```

**Comment Instruction**

Although if is not really an instruction, comments can only be used and presented while they are present in an instruction position. As noted on the examples above, comments are presented with one or more semicolon ; in instructions sequences.

## 4.4   Procedure and Expression Compilation and Parsing

Before being executed, procedures are compiled. This is done automatically when you load a OP file in a OPRS Kernel. Most of the time, this compilation goes without any problem (because procedures have been created with the OP Editor which is fairly rigorous regarding the syntax allowed for a OP). However, one could imagine that, for some reason, a OP may not be parsable by the OPRS Kernel and could be rejected by the OP compiler.

In any case, the OP compiler does more than just syntax checking. Some semantic checking is done too. For example, it can check that all the symbols you use have been declared. It can also check that the actions which appear in the action part of an Action OP are indeed declared as evaluable functions. The compilation and syntax checking does not only apply to procedure themeselves, but also to expression parsed by the kernel.

### 4.4.1   Action Checking

If a symbol is considered to be associated to an action definition (because it appears as an action call in an action part of a OP), then the system will check that such action has been defined (linked) in the kernel. A warning will be issued if no definition can be found for this action. Special action can be defined as evaluable functions too. In other words, if you have declared the function `FOO` as an evaluable function, you can call it in place of a special action. The system will also check that the number of arguments of the call is consistent with the number of arguments of the declaration.

Part of this checking can be turned on or off using the `set action on|off` command (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

### 4.4.2   Predicate Checking

Predicates are also subject to some checking at parsing time. First, if the predicate checking is on, then the system will check that the predicate has been declared before its first occurrence in an expression. Predicate can be declared with a `declare predicate` command or one of the `declare ff`, `declare be`, `declare op_predicate`. Note that an evaluable predicate is automatically declared (see [OPRS Kernel Declaration Commands], §2.9, page 35). This pred-

icate checking can be turned on or off using the `set predicate on|off` command (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34). Moreover, the number of arguments of predicates is now fixed. Therefore, whenever the system encounters a predicate for the very first time, it will set its number of argument and will then check that subsequent use of this predicate is done with the same number of argument.

### 4.4.3   Function Checking

Functions are also subject to some checking at parsing time. First, if the function checking is on, then the system will check that the function has been declared before its first occurrence in an expression. Function can be declared with a `declare function` command. Note that an evaluable function is automatically declared (see [OPRS Kernel Declaration Commands], §2.9, page 35). This checking can be turned on or off using the `set function on|off` command (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

### 4.4.4   Symbol Checking

While compiling OP and parsing expresion, the system will check for new symbol. In other words¡, if it encounters a symbol for the first time, a warning will be displayed to inform the user of this newly used symbol. This facility can be disabled but it is usually very useful to catch typos... Nevertheless, it is advised to create a symbol file, (containing `declare id` commands for each symbol you want to declare in your application). Note that all symbols declared by other means.

This checking can be turned on or off using the `set symbol on|off` command (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

# Chapter 5

# Database

The content of the OPRS database represents the current beliefs of the system. Some of these beliefs are provided initially by the system user. Typically, they include facts about static properties of the application domain, such as the structure of some subsystems or the physical laws that must be obeyed by certain mechanical components. Other beliefs are derived by OPRS itself as it executes its OPs. They are typically current observations about the world or conclusions derived by the system from these observations, and they may change over time. For example, at some times OPRS may believe that the pressure of a tank is within acceptable operating limits, at other times not. Updates to the database therefore necessitate the use of consistency maintenance techniques.

The database is one of the most important components of the OPRS Kernel. Its role is basically to "remember" which expressions or evaluable predicates are true and, to give the expressions back upon consultation. This consultation can be done by the user with a `consult` command (see [OPRS Kernel Database Commands], §2.2, page 28), or by the kernel itself when it tries to execute a procedure (see [Procedure Execution and Run Time], §7, page 109).

On top of these basic functionalities, several related functionalities are provided by the database and are very often critical in the application using OPRS. They are all presented in this chapter.

The database uses a term indexing mechanism [Sti87], which has been extended in various ways to handle some specific operations, and allows consultation in constant time.

## 5.1   Database File Format

The internal format of the database is of no interest to the end user. However, a user may need to load a certain number of facts at once. This can be done by issuing a series of `conclude` commands (see [OPRS Kernel Database Commands], §2.2, page 28), but this can be rather tedious... More easily, this can be done by loading a database file (see [OPRS Kernel Loading Commands],

§2.4, page 30).

The database file format is rather simple. It is a list of expressions. As in Lisp, any line starting with a semicolon ; is considered as a comment.

Here is an example of such a file:

```
;;; This is a comment
(
(foo a b)
(bar 23 (+ 4 5))
(boo (f g h) "this is a string" 23.44)
(foo a c)
(bar 3 (+ 74 5))
(foo (f g h) "this is a string" 23.44)
(fo a b)
(ba 23 (+ 4 5))
(bo (f g h) "this is a string" 23.44)
)
```

It is recommended to use the '.db' extension for these database files.

Note that you need to declare any closed world predicate and functional fact before you load a database containing facts referring to them. Evaluable functions and predicates are defined beforehand in the kernel at link time, so no precaution has to be taken for them.

## 5.2   Unification

Although not specific to the database, the unification used in the OPRS Kernel is mostly used in database operations (but not only in database operations). For example, when you consult an expression such as (foo $x 6), then at some point the $x variable may be unified to a term if necessary.

The unification mechanism considers two kinds of variables (see [Variables], §3.1, page 43), logical variables and program variables. Keep in mind that logical variables are bound once onward and stay bound on a successful execution (this is similar to PROLOG variable). However program variables, which are provided for convenience, can be rebound at any time.

Last, but not least, it is in the unification (but not only in the unification) that evaluable functions (see [Evaluable Functions], §6, page 95), are evaluated. This means that a term containing an evaluable function is evaluated whenever one attempts to unify it. It used to be that the evaluation of terms appearing in a goal were delayed until unification was required. Recent versions of the kernel still allow this mechanism, but it is not the default anymore. Now to prevent the evaluation of terms at posting time, you need to surround the term in a quote form (see [Current and Quote], §10.7, page 141).

Nevertheless, if a term containing an evaluable function is in the scope of a quote, this can lead to a very peculiar behavior particularly if this evaluable function runs with considerable overhead, or if it has side effects (such as

printing). You may get the feeling that your function is evaluated more than it should be. To avoid this mechanism, you can use the `current` mechanism in the scope of the quote.

## 5.3   Conclude

The conclude operation is the operation by which you add something to the database. Only expressions, without variables, are allowed. All the evaluable functions found in the expression are evaluated and the result is concluded, not the original expression. For example, if you conclude `(foo (+ 1 2) (- 5 4))`, then `(foo 3 1)` ends up in the database.

You can conclude facts with negation, `(  (foo 3 1))`, for example. Moreover, when an expression is concluded, its negation, if found, is automatically retracted from the database.

Of course, concluding the negation of closed world predicates is a ıno-op (see [Closed World Predicates], §5.5, page 83). Similarly, concluding an evaluable predicate produces a warning. For example, if you conclude `(> 3 1)`, the system reports a non fatal error, even if the expression is true.

## 5.4   Consultation

One can consult general expression (see [General Expressions], §3.6, page 50), with variables. Conjunctions and disjunctions are treated, as expected, with the variables having scope over the whole general expression. In `(& (foo $x 6) (bar $x 7))`, the `$x` variable is the same in both expressions. If you do not want this behavior, you should use two different variables. When a disjunction is consulted, all the possible sub general expressions which are satisfied are returned. If you have `(foo 1 6)` and `(bar 1 7)` in the database, and consult `(|| (foo 1 6) (bar 1 7))`, you get three possible solutions: `(|| (foo 1 6) (bar 1 7))`, `(|| (bar 1 7))` and `(|| (foo 1 6))`. One can easily imagine that consulting a big disjunction with lot of simple true expressions leads to a huge number of solutions. For example, N disjunctions, each being true for M values, would lead to (M**N)!/N! solutions! No need to say that extreme caution should be exerted when writing disjunction in OPs.

When you consult a general expression in the database, the kernel returns a list of facts. If the appropriate flag is on (`trace db frame on|off`) (see [OPRS Kernel Trace Commands], §2.5, page 31), it will return each fact with the frames which, when applied to the consulted fact, make each returned fact be true. Note that for internal consultation, facts themselves are of little if no interest for the kernel. Only the frame is important. The deepest frame in the hierarchy is the frame in which the fact you consult has been created (most likely an empty frame with the variables present in the consulted gexpression).

If you have `(foo 1 6)` and `(foo 2 6)` in your database, and you consult `(foo $x 6)`, then it returns something like:

```
FOO> conclude (foo 1 6)
The expression:( FOO 1 6 )  has been concluded in the database.


FOO> conclude (foo 2 6)
The expression:( FOO 2 6 )  has been concluded in the database.


FOO> consult (foo $x 6)
The user consultation of: ( FOO {$X->(Term *)NULL } 6 )
gives the following result:
( FOO 2 6 )
( FOO 1 6 )


FOO> trace db frame on

FOO> consult (foo $x 6)
The user consultation of: ( FOO {$X->(Term *)NULL } 6 )
gives the following result:
[ ( FOO 2 6 ) { 0 ([{$X->(Term *)NULL}>>2 ] )
                { ({$X->(Term *)NULL} ) 1 () {} } } ]
[ ( FOO 1 6 ) { 0 ([{$X->(Term *)NULL}>>1 ] )
                { ({$X->(Term *)NULL} ) 1 () {} } } ]
```

Note that the format of the returned information is a list of:

`fact`

   or

`[ fact frame-in-which-the-fact-is-true ]`

depending of the **db_frame** flag (see [OPRS Kernel Trace Commands], §2.5, page 31).

Multiple frames are returned when the consultation returns more than one possible value. Moreover, if the consultation leads to a tree search because of a conjunction or a disjunction in the consulted gexpression, then you get more than one level of nested frames.

```
FOO> conclude (foo a b)
The expression:( FOO A B )  has been concluded in the database.


FOO> conclude (bar b c)
The expression:( BAR B C )  has been concluded in the database.


FOO> conclude (bar b d)
The expression:( BAR B D )  has been concluded in the database.


FOO> consult (& (foo $x $y) (bar $y $z))
The user consultation of:
```

```
( & ( FOO {$X->(Term *)NULL } {$Y->(Term *)NULL } )
    ( BAR {$Y->(Term *)NULL } {$Z->(Term *)NULL } ) )
gives the following result:
[ ( & ( FOO A B ) ( BAR B C ) )
  { 0 ([{$Z->(Term *)NULL } >>C ] )
   { 0 ([{$Y->(Term *)NULL } >>B ] [{$X->(Term *)NULL } >>A ] )
    { 1 ()
     {} } } } ]
[ ( & ( FOO A B ) ( BAR B D ) )
  { 0 ([{$Z->(Term *)NULL } >>D ] )
   { 0 ([{$Y->(Term *)NULL } >>B ] [{$X->(Term *)NULL } >>A ] )
    { 1 ()
     {} } } } ]

FOO> consult (|| (foo $x $y) (bar $y $z))
The user consultation of:
( || ( FOO {$X->(Term *)NULL } {$Y->(Term *)NULL } )
    ( BAR {$Y->(Term *)NULL } {$Z->(Term *)NULL } ) )
gives the following result:
[ ( || ( BAR B C ) )
  { 0 ([{$Z->(Term *)NULL } >>C ] [{$Y->(Term *)NULL } >>B ] )
   { 1 ()
    {} } } ]
[ ( || ( BAR B D ) )
  { 0 ([{$Z->(Term *)NULL } >>D ] [{$Y->(Term *)NULL } >>B ] )
   { 1 ()
    {} } } ]
[ ( || ( FOO A B ) )
  { 0 ([{$Y->(Term *)NULL } >>B ] [{$X->(Term *)NULL } >>A ] )
   { 1 ()
    {} } } ]
[ ( || ( FOO A B ) ( BAR B C ) )
  { 0 ([{$Z->(Term *)NULL } >>C ] )
   { 1 ()
    {} } } ]
[ ( || ( FOO A B ) ( BAR B D ) )
  { 0 ([{$Z->(Term *)NULL } >>D ] )
   { 1 ()
    {} } } ]
```

## 5.5   Closed World Predicates

A closed world predicate, in short CWP, is a mechanism by which one can
specify what to do in the absence of information about some predicate.  In
OPRS Kernel database, facts are known to be true if they are in the database

and are considered false if they are absent. As a consequence, in the absence of any information about an expression and its negation. Both are considered to be false. However, in many situations, the absence of some information is considered as if it were false. For example, assume you have built a database containing information about flights between cities. If your database does not contain the fact that there is a direct flight between Toulouse and San Francisco, most likely, it means that no such flight exists. Therefore, if you ask `consult (direct-flight TLS SFO)`, it returns FALSE. But if you ask `consult (  (direct-flight TLS SFO))`, then it also returns FALSE by default, although it is TRUE. In fact, one solution would be to enter all the possible flights which do not exist, by adding explicitly (  (direct-flight TLS SFO)) and all the other ones. This is tedious and will clutter your database with unnecessary information.

The solution to this problem in OPRS Kernel is to declare the predicate `direct-flight` as CWP with the following command: `declare_cwp direct-flight` (see [OPRS Kernel Database Commands], §2.2, page 28). It means automatically that any consultation of the negation of this predicate (such as `consult (  (direct-flight TLS SFO))`) returns TRUE if the positive ((`direct-flight TLS SFO`)) is not in the database.

When you declare evaluable predicate (see [Evaluable Predicates], §5.8, page 88), you can also specify if they are CWP or not. All the evaluable predicates predefined in the kernel are CWP.

A side effect of this declaration is that any attempt to conclude the negation of a CWP is ignored by the database. Indeed, what would be the point of concluding (  (direct-flight TLS SFO)) as it is true anyway (providing of course that (`direct-flight TLS SFO`) is not in the database).

Here are some examples to illustrate the CWP mechanism:

```
FOO> consult (foo a)
The user consultation of: ( FOO A ) gives the following result:
NULL

FOO> consult (~ (foo a))
The user consultation of: ( ~ (FOO A ) ) gives the following result:
NULL

FOO> declare cwp foo

FOO>  consult (foo a)
The user consultation of: ( FOO A ) gives the following result:
NULL

FOO> consult (~ (foo a))
The user consultation of: ( ~ (FOO A ) ) gives the following result:
[ ( ~ (FOO A ) ) { 1 () {} } ]
```

```
FOO> consult (~ (foo $x))
The user consultation of:
( ~ (FOO {$X->(Term *)NULL } ) )
gives the following result:
[ ( ~ (FOO {$X->(Term *)NULL } ) ) { 1 () {} } ]

FOO> conclude (foo a)
The expression:( FOO A )  has been concluded in the database.

FOO> consult (foo a)
The user consultation of:
( FOO A )
gives the following result:
[ ( FOO A ) { 0 () { 1 () {} } } ]

FOO> consult (foo $x)
The user consultation of:
( FOO {$X->(Term *)NULL } )
gives the following result:
[ ( FOO A )
  { 0 ([{$X->(Term *)NULL } >>A ] ){ 1 () {} } } ]

FOO> consult (~ (foo a))
The user consultation of:
( ~ (FOO A ) )
gives the following result:
NULL

FOO>  consult (~ (foo $x))
The user consultation of:
( ~ (FOO {$X->(Term *)NULL } ) )
gives the following result:
NULL

FOO> conclude (~ (foo a))
The expression:( ~ (FOO A ) )  has been concluded in the database.

FOO> consult (foo a)
The user consultation of: ( FOO A ) gives the following result:
NULL

FOO> consult (foo $x)
The user consultation of: ( FOO {$X->(Term *)NULL } )
gives the following result:
NULL
```

```
FOO> consult (~ (foo a))
The user consultation of: ( ~ (FOO A ) )
gives the following result:
[ ( ~ (FOO A ) ) { 1 () {} } ]


FOO>  consult (~ (foo $x))
The user consultation of: ( ~ (FOO {$X->(Term *)NULL } ) )
gives the following result:
[ ( ~ (FOO {$X->(Term *)NULL } ) ) { 1 () {} } ]
```

Note the absence of binding on the consultation of CWP expression with
variables.

## 5.6   Functional Facts

Functional facts are predicates which can be expressed as a function of a subset
of their arguments which gives the rest of their arguments as a result. In some
cases, there is only one possible result:   For example: (`factorial 5 120`)
(because `factorial(5) => 120`).  This result being unique (`factorial 5` is
always `120`), there is no reason a priori to make any bookkeeping on it. But
in some situations or applications, the result is not unique and can change over
time. For example:   (`pressure tk1 245`) (indeed, we can express the fact
that `pressure(tk1) => 245`).  Most likely, there is only one possible value of
pressure at one time, therefore any previously recorded value for `tk1` should be
discarded. In this case, if we had received (`pressure tk1 250`) few minutes (or
seconds) ago, we must remove it from the database, otherwise the consultation
of (`pressure tk1 $x`) would return both facts, which is wrong.

You can certainly write your own procedure to clean up automatically. How-
ever, the OPRS Kernel provides an automatic mechanism to handle this type
of cleanup. It is called functional fact, and you can declare any predicate as a
functional fact with the following command: `declare ff predicate position`
(see [OPRS Kernel Declaration Commands], §2.9, page 35). `predicate` is the
name of the predicate.  `position` is the argument position at which the ar-
gument becomes the result of the functional evaluation. For example, in the
pressure example above, you would invoke: `declare ff pressure 1`.

For a predicate like:   (`position-robot building-E`), it is  `declare ff
position-robot 0`, because `position-robot() =>building-E`. For a predi-
cate like (`connection-status paris toulouse up`), it is `declare ff connection-status
2`, because `connection-status(paris, toulouse) => up`.

This mechanism requires the order of the arguments to be organized in such
a way that the value arguments of the functional evaluation come after the
arguments required for the evaluation. If you said (`pressure 250 tk1`), then
you cannot use this mechanism (or we would have to use position pattern to
declare the functional facts).

Note that in this section, we mention that the predicate is considered as
functional, which does not mean it is evaluable (see [Evaluable Predicates],

§5.8, page 88), but merely that it could be expressed as functional.

All functional fact predicates are also considered as closed world predicate (see [Closed World Predicates], §5.5, page 83).

Many users have asked us to extend this mechanism to an *history* mechanism. You could then declare that you want to keep the last five values of a functional fact. This notion of history is heavily linked to the notion of *time stamping* the information stored in the database. As usual, there are pros and cons of such a mechanism. However, we are currently considering adding it. As of now, you need to do the bookkeeping yourself when you need more previous values, to do trend analysis for example.

Here are some examples to illustrate the functional fact mechanism:

```
FOO> consult (foo $x)
The user consultation of: ( FOO {$X->(Term *)NULL } )
gives the following result:
NULL

FOO> conclude (foo a)
The expression:( FOO A )  has been concluded in the database.

FOO> consult (foo $x)
The user consultation of: ( FOO {$X->(Term *)NULL } )
gives the following result:
[ ( FOO A ) { 0 ([{$X->(Term *)NULL } >>A ] )
             { 1 () {} } } ]

FOO> conclude (foo b)
The expression:( FOO B )  has been concluded in the database.

FOO> consult (foo $x)
The user consultation of: ( FOO {$X->(Term *)NULL } )
gives the following result:
[ ( FOO A ) { 0 ([{$X->(Term *)NULL } >>A ] )
             { 1 () {} } } ]
[ ( FOO B ) { 0 ([{$X->(Term *)NULL } >>B ] )
             { 1 () {} } } ]

FOO> declare ff foo 0

FOO> conclude (foo c)
The expression:( FOO C )  has been concluded in the database.

FOO> consult (foo $x)
The user consultation of: ( FOO {$X->(Term *)NULL } )
gives the following result:
[ ( FOO C ) { 0 ([{$X->(Term *)NULL } >>C ] )
```

```
            { 1 () {} } } ]

FOO> conclude (foo d)
The expression:( FOO D )  has been concluded in the database.

FOO> consult (foo $x)
The user consultation of: ( FOO {$X->(Term *)NULL } )
gives the following result:
[ ( FOO D ) { 0 ([{$X->(Term *)NULL } >>D ] )
            { 1 () {} } } ]
```

## 5.7   Basic Events

Basic events are used to handle facts which are "transients", i.e. facts which must be noticed by the OPRS Kernel but should not be kept in the database. Keep in mind that any fact or message which is received by a OPRS Kernel is stored in the database by default.

For example, you may want your application to notice facts such as (`alarm`), but do not want it to remember this fact. You can declare the `alarm` predicate as a basic event. To do so, you need to issue the `declare be alarm` (see [OPRS Kernel Declaration Commands], §2.9, page 35), command in this kernel (or put it in the include file which will be loaded in this kernel). Similarly, you may want your application to monitor some pressure values but do not want to keep these values (even the last one). You could then issue the command `declare be pressure` (assuming that the pressure predicate is used to carry this information).

Note that you can declare any predicate as a basic event, and some predefined predicates are actually basic event predicates. Here is the list of the predicates which are, by default, declared as basic events in the kernel.   ×
`SOAK`, `APPLICABLE-OPS-FACT`, `FACT-INVOKED-OPS`, `DB-SATISFIED-GOAL`, ×
`APPLICABLE-OPS-GOAL`, `GOAL-INVOKED-OPS`, `FAILED-GOAL`, `FAILED`, `REQUEST`, `ACHIEVED`, `INTENTION-WAKE-UP`, `READ-RESPONSE-ID`, `READ-RESPONSE`.

You can undeclare a basic event with the command: `undeclare_be` (assuming it was declared as a basic event).

## 5.8   Evaluable Predicates

OPRS provides mechanisms to define, incorporate and use evaluable predicates. By default there are a number of predefined evaluable predicates. However, the user can add its own definition of evaluable predicates, and even redefine the one which are defined by default in the kernel.

Evaluable predicates are used whenever you want to have a predicate to correspond to some C code (or code linked in the OPRS Kernel), which evaluation describes the extension of the predicate.

Evaluable predicates are evaluated at the database level (which explain why we present them in this Chapter). In fact, from the final user point of view, the mechanism is completely transparent, they look like standard predicates. The database somehow "recognizes" them as being evaluable, evaluates their arguments and the predicates themselves instead of looking in the expression table . Evaluable predicates can be consulted as well as other predicates.

However, consulting `(> (+ 3 4) (- 3 4))` in the database (with a the `consult` command), evaluates `(+ 3 4)` and `(- 3 4)` which respectively returns `7` and `-1` (see [Evaluable Functions], §6, page 95, for more on this subject). Then, it evaluates the `(> 7 -1)` and it returns TRUE.

One can see from this mechanism that it is not possible to use an evaluable predicate in an environment where the terms are undefined (or unbound). It is usually meaningless, and very often an error, to consult evaluable predicates with unbound variables. Such as: `(> 4 $x)` when `$x` is unbound. In fact the `>` predicate will print an error message.

Evaluable predicates can be used whenever a predicate is appropriate, however, you cannot use evaluable predicates in an Invocation Part. If you use a predicate in an Invocation Part, it will not be able to trigger the relevance of this OP as any predicate of the Invocation Part should.

## 5.8.1   Predefined Evaluable Predicates

All the evaluable predicates return a `PBoolean`, i.e. `TRUE` or `FALSE` (see [Important Constants], §G.1.3, page 340), not a Term. Besides, all the evaluable predicates take a `TermList` as an argument. Whenever it is possible, we will specify the number of arguments and the type of the `Term` for each element.

To help the reader understand the descriptions for the evaluable predicates in the following section, consider the `>` (greater than) evaluable predicate:

`>`                                                        **Evaluable Predicate**

> `PBoolean > (INTEGER or FLOAT INTEGER or FLOAT)` is the greater than function. It is defined for two terms (subsequent terms in the list are ignored). It can compare any numbers (`INTEGER` or `FLOAT`).

The `PBoolean` before the predicate name `>` is the type returned by the evaluable predicate. The `(INTEGER or FLOAT INTEGER or FLOAT)` after the predicate name specifies the type of the object contained in the TermList which is the argument to the `>` predicate. In this case, it means at least two arguments which must be `FLOAT` or `INTEGER Term *`. See [liblist.a library], §G.3, page 350, for examples of how to access different types of objects contained in TermList, as well as [How to Define your Own Evaluable Predicates], §5.8.2, page 93.

Evaluable predicates can be classified in different categories which are presented in the following section.

**Arithmetic Evaluable Predicates**

These predicates deal with numbers.

**>**                                                                    **Evaluable Predicate**

   `PBoolean > (INTEGER or FLOAT INTEGER or FLOAT)` is the greater
than function. It is defined for two terms (subsequent terms in the
list are ignored). It can compare any numbers (`INTEGER` or `FLOAT`).

**>=**                                                                   **Evaluable Predicate**

   `PBoolean >= (INTEGER or FLOAT INTEGER or FLOAT)` is the greater
than or equal function. It is defined for two terms (subsequent terms
in the list are ignored). It can compare any numbers (`INTEGER` or
`FLOAT`).

**<**                                                                    **Evaluable Predicate**

   `PBoolean < (INTEGER or FLOAT INTEGER or FLOAT)`  is the less
than function. It is defined for two terms (subsequent terms in the
list are ignored). It can compare any numbers (`INTEGER` or `FLOAT`).

**<=**                                                                   **Evaluable Predicate**

   `PBoolean <= (INTEGER or FLOAT INTEGER or FLOAT)`  is the less
than or equal function. It is defined for two terms (subsequent terms
in the list are ignored). It can compare any numbers (`INTEGER` or
`FLOAT`).

### OP Instance Related Evaluable Predicates

These predicates deal with OP Instance and the properties of the OP they are
an instance of.

**PROPERTY-P**                                        **Evaluable Predicate**

   `PBoolean PROPERTY-P (ATOM property,TT_OP_INSTANCE)`  is the
function used to check if a property is defined. It is defined for two
terms, a term symbol, the property name, and a `TT_OP_INSTANCE`
(subsequent terms in the list are ignored). It returns `TRUE` if the
specified property is non-nil for the specified OP Instance. Note
that this does not return the value of the property, but merely if
it is defined and non-nil. If you want to access the value of the
property, use the evaluable function: `property-of`.

**NOT-AN-INSTANCE-OF-ME**                         **Evaluable Predicate**

   `PBoolean NOT-AN-INSTANCE-OF-ME (TT_OP_INSTANCE)`  is defined
for 1 term a `TT_OP_INSTANCE`. It returns `TRUE` if the OP, in which
this predicate appears in the invocation, or context part, is not the
OP from which the op-instance parameter is an instance. This is very
useful for Meta level OPs to prevent them of looping on themselves...

**IS-FACT-INVOKED**                               **Evaluable Predicate**

PBoolean IS-FACT-INVOKED (TT_OP_INSTANCE) is defined for 1 term
a TT_OP_INSTANCE. It returns TRUE if the op-instance parameter is
invoked by a fact.

## IS-GOAL-INVOKED                          Evaluable Predicate

PBoolean IS-GOAL-INVOKED (TT_OP_INSTANCE) is defined for 1 term
a TT_OP_INSTANCE. It returns TRUE if the op-instance parameter is
invoked by a goal.

### Time Related Evaluable Predicates

These predicates deal with time and elapsed time.

## ELAPSED-TIME                             Evaluable Predicate

PBoolean ELAPSED-TIME (INTEGER t1, INTEGER t2) is the pred-
icate used to check if the number of seconds t2 have elapsed since t1.
It is mainly used in a construct such as (^ (elapsed-time (time)
5)).

## ELAPSED-MTIME                            Evaluable Predicate

PBoolean ELAPSED-MTIME (INTEGER t1, INTEGER t2) is the pred-
icate used to check if the number of milliseconds t2 have elapsed since
t1. It is mostly used in a construct such as (^ (elapsed-mtime
(mtime) 5)). Keep in mind that if the OPRS Kernel has noth-
ing to do, except checking this predicate, then it will wake up every
main_loop_pool_sec * 1000000 + main_loop_pool_usec micro sec-
onds (this value can be reduced if needed) (see [Important Variables],
§G.1.2, page 338).

**'Due to possible overlapping reasons, do not use this predi-
cate with big values of t2. For values greater than 10 000 or
so, use elapsed-time and time instead which count seconds'**.

### Miscellaneous Evaluable Predicates

These miscellaneous predicates deal with various objects and types.

## =                                        Evaluable Predicate

PBoolean = (AnyTerm AnyTerm) is the assignment predicate. This
predicate used to be satisfied by a OP, it is now defined as an evalu-
able predicate. (Note: that you can still define it as a OP). It is
defined for two terms. It returns TRUE if it manage to assign (or
unify) the value of the second term to the first term. Most of the
time, the first term is a variable.

## ==                                       Evaluable Predicate

`PBoolean == (AnyTerm AnyTerm)` is the unification predicate. This predicate used to be satisfied by a OP, it is now defined as an evaluable predicate. (Note: that you can still define it as a OP). It is defined for two terms. It returns `TRUE` if it managed to unify the second term with the first term. One advantage of having this predicate defined as an evaluable predicate is that you can use it in complex statement such as: `(?  (|| (== $x (foo a b)) (== (foo a b) (foo $x $y))))`

### NULL                                                          **Evaluable Predicate**

`PBoolean NULL (LISP_LIST)` is the Lisp List NULL function. It is defined for one term (subsequent terms in the list are ignored). It returns `TRUE` if the Lisp List is empty.

### NULL_CAR                                                      **Evaluable Predicate**

`PBoolean NULL_CAR (Any Term)` is the Lisp List NULL function. It is defined for one term. It returns `TRUE` if the `Term` is NULL, as extracted with a `CAR` from an empty Lisp list.

### MEMQ                                                          **Evaluable Predicate**

`PBoolean MEMQ (Any Term, LISP_LIST)` returns `TRUE` if the `Any Term` is in the codeLISP_LIST.

### NULL_C                                                        **Evaluable Predicate**

`PBoolean NULL_C (OPRS_LIST)` is the C_List NULL function. It is defined for one term (subsequent terms in the list are ignored). It returns `TRUE` if the `OPRS_LIST` is empty.

### EQUAL                                                         **Evaluable Predicate**

`PBoolean EQUAL (TermList terms)` is the equal terms function. It is defined for two terms (subsequent terms in the list are ignored). It returns `TRUE` if both terms are equal.

### BOUNDP                                                        **Evaluable Predicate**

`PBoolean BOUNDP (Any Term)` It is defined for one term (subsequent terms in the list are ignored). It returns `FALSE` if the term is a `VARIABLE` and it is not bound. Return `TRUE` in all other cases.

### NUMBERP                                                       **Evaluable Predicate**

`PBoolean NUMBERP (Any Term)` It is defined for one term (subsequent terms in the list are ignored). It returns `TRUE` if the term is a `FLOAT` or an `INTEGER`, and returns `FALSE` otherwise.

### STRINGP                                                       **Evaluable Predicate**

PBoolean STRINGP (Any Term) It is defined for one term (subsequent terms in the list are ignored). It returns TRUE if the term is a STRING, and returns FALSE otherwise.

### CONSP                                            **Evaluable Predicate**

PBoolean CONSP (Any Term)  It is defined for one term (subsequent terms in the list are ignored).  It returns TRUE if the term is a LISP_LIST, and returns FALSE otherwise.

### ATOMP                                            **Evaluable Predicate**

PBoolean ATOMP (Any Term) It is defined for one term (subsequent terms in the list are ignored). It returns TRUE if the term is a ATOM, and returns FALSE otherwise.

## 5.8.2   How to Define your Own Evaluable Predicates

It is fairly easy to define your own evaluable predicates. To do so, you have to write a C function which takes a list of terms (TermList) as arguments and returns a PBoolean (TRUE or FALSE). Using the list library functions, you can then access the element of the list and compute the Boolean value.

You will find various examples of user-defined evaluable predicates in the file: *'user-ev-predicate.c'*.

```
PBoolean my_predicate(TermList tl)
{
    Term *t1, *t2;

    t1 = (Term *)get_list_pos(tl, 1);
    t2 = (Term *)get_list_pos(tl, 2);

    if (my_fancy_condition(t1, t2)) return TRUE;
    else return FALSE;
}
```

You can define as many evaluable predicates as you want.  You need to declare them in the kernel, as well as their external names (as it will appear in the OPs), the number of arguments they take and whether they are closed world predicates or not.  This declaration is made in the body of the declare_user_eval_pred function which is called upon start up of the kernel.

```
void declare_user_eval_pred(void)
{
    make_and_declare_eval_pred("EXTERNAL_NAME", my_predicate, 2, TRUE);
    return;
}
```

### make_and_declare_eval_pred        **Kernel User Function**

void make_and_declare_eval_pred (Predicat name, PFB pred, int ar, PBoolean cwp) is used to declare the evaluable predicate. You have to specify the predicate name, the C function which implements it, the arity of this predicate and a boolean to indicate if the predicate is a closed world predicate.

## declare_user_eval_pred                    **Kernel User Function**

void declare_user_eval_pred (void)  is the function in which you must put all the calls to make_and_declare_eval_pred. It is called upon start-up by the kernel and builds the appropriate table to map the evaluable predicate names and the corresponding C functions. Note that the user can redefine the predefined evaluable predicates by using their names. This can be useful for example if you want to overload their definitions.

## 5.9   OP Predicates

OP predicates cannot be satisfied in the database, they can only be satisfied by OP execution.  Keep in mind that whenever the system posts a goal, it will check if this goal is already satisfied in the database.  There exists a number of predicates that can only be satisfied by OP execution (example print, send-message, execute-command, read-inside, etc), for them, there is no need to check if they are satisfied or not in the database, so they should be declared as OP predicate.

As a consequence, consulting expressions in which the predicate is a OP predicate in the database result as a non operation.  Similarly, if you attempt to conclude a OP predicate in the database, the system will print a warning.

# Chapter 6

# Evaluable Functions

OPRS provides mechanisms to define, incorporate and use evaluable functions. By default there are a number of predefined evaluable functions. However, you can add a definition of evaluable functions, and even redefine the ones which are defined by default in the kernel. Actions (which have similitude with evaluable functions) are presented in [Using Action OPs], §7.7, page 114.

Evaluable functions are used whenever you want to evaluate some expression, to compute some results, or to have some "side effects" on external modules (by sending a message, or by calling a function which will have some effects on these external modules).

Evaluable functions can be used in a Composed Term embedded in an Expression. Whenever an evaluable function is called, each argument must be defined and bound (unless the function is able to handle unbound variable, but this is unlikely).

## 6.1 Predefined Evaluable Functions

All the evaluable functions return a `Term *`, i.e. a pointer to a `Term` structure (see [Data Structures and Types Used], §G.1.1, page 337). However, this term can contain different types of objects (see [Terms], §3.2, page 45), and this is left to the user to decide which type he needs to return. In the following description, we will indicate which is the type of the object contained in the `Term *` returned by the evaluable function. Besides, all the evaluable functions take a `TermList` as an argument. Whenever it is possible, we will specify the number of arguments and the type of the `Term` for each element.

To help the reader understand the descriptions for the evaluable functions in the following section, consider the `+` (plus) evaluable function:

**+**                                                        **Evaluable Function**

> `INTEGER or FLOAT + (INTEGER or FLOAT+)` is the plus function. It is defined for n terms. It adds any numbers to the first argument (`INTEGER` or `FLOAT`), and returns a term containing the result casted

95

according to the passed arguments (if all `INTEGER`s then `INTEGER`, `FLOAT` otherwise).

The `INTEGER or FLOAT` before the function name `+` is the type of the object contained in the `Term *` object that is returned by this `+` function. The `(INTEGER or FLOAT)+` after the function name specifies the type of the object contained in the TermList which is the argument to the `plus` function. In this case, it means at least one argument (this is represented with the trailing `+`), and all the arguments must be `FLOAT` or `INTEGER` `Term *`. See [liblist.a library], §G.3, page 350, for examples of how to access different types of objects contained in TermList, as well as [How to Define your Own Evaluable Functions], §6.2, page 107, for examples of how to store different types of objects in a `Term *` that will be returned by evaluable functions.

Evaluable functions can be classified in different sections according to their type, or to the type of objects they manipulate/return.

### 6.1.1  Arithmetic Evaluable Functions

These evaluable functions deal with numbers and perform the common arithmetic operation.

**+**                                                      **Evaluable Function**

> `INTEGER or FLOAT + (INTEGER or FLOAT+)` is the plus function. It is defined for n terms. It adds any numbers to the first argument (`INTEGER` or `FLOAT`), and returns a term containing the result casted according to the passed arguments (if all `INTEGER`s then `INTEGER`, `FLOAT` otherwise).

**−**                                                      **Evaluable Function**

> `INTEGER or FLOAT - (INTEGER or FLOAT+)` is the difference function. It is defined for n terms. It subtracts any number to the first argument (`INTEGER` or `FLOAT`), and returns a term containing the result casted according to the passed arguments (if all `INTEGER`s then `INTEGER`, `FLOAT` otherwise). With one argument, it returns the minus.

**\***                                                     **Evaluable Function**

> `INTEGER or FLOAT * (INTEGER or FLOAT+)` is the time function. It is defined for n terms. It multiplies any numbers (`INTEGER` or `FLOAT`), and returns a term containing the result casted according to the passed arguments (if all `INTEGER`s then `INTEGER`, `FLOAT` otherwise).

**/**                                                      **Evaluable Function**

> `INTEGER or FLOAT / (INTEGER or FLOAT+)` is the divide function. It is defined for n terms. It divides the first number (`INTEGER`

or `FLOAT`) by the subsequent numbers, and returns a term containing the result casted according to the passed arguments (if all `INTEGER` then `INTEGER`, `FLOAT` otherwise). With one argument, it returns the inverse.

### abs                                            Evaluable Function

`INTEGER or FLOAT abs (INTEGER or FLOAT)` is the abs function. It is defined for 1 term. It returns the abs value of its argument (`INTEGER` or `FLOAT`).

### mod                                            Evaluable Function

`INTEGER mod (INTEGER INTEGER)` is the modulo function. It is defined for two `INTEGER` terms. It returns an `INTEGER`, the modulo of the two integers (it is equivalent to the C `%` operation).

### rand                                           Evaluable Function

`INTEGER rand (INTEGER)` is defined for one `INTEGER` terms. It returns a random `INTEGER` between 0 and the int given as argument (not included).

### float-to-int                                   Evaluable Function

`INTEGER float-to-int (FLOAT)` is defined for 1 term. It returns the `FLOAT` passed as argument casted in an `INTEGER`.

### int-to-float                                   Evaluable Function

`FLOAT int-to-float (INTEGER)` is defined for 1 term. It returns the `INTEGER` passed as argument casted in a `FLOAT`.

## 6.1.2   Array Manipulation Evaluable Functions

These evaluable functions deal with array, `INT_ARRAY` and `FLOAT_ARRAY`. See [Array of Floats as a Term], §3.2.13, page 48 and [Array of Integers as a Term], §3.2.12, page 48, for more information on arrays.

### make-float-array                               Evaluable Function

`FLOAT_ARRAY make-float-array (INTEGER size)` returns a `Term *` containing a `FLOAT_ARRAY` of size `size`. This function should be used in a Special Action.

### get-float-array                                Evaluable Function

`FLOAT get-float-array (FLOAT_ARRAY float_array, INTEGER index)` returns a `Term *` containing the `FLOAT` (in fact it is a double) stored in the array `float_array` at `INTEGER index`.

### get-float-array-size                           Evaluable Function

> INTEGER get-float-array-size (FLOAT_ARRAY float_array) re-
> turns a `Term *` containing the `INTEGER` value of the size of the
> float_array.

### make-int-array                                   **Evaluable Function**

> INT_ARRAY make-int-array (INTEGER size) returns a `Term *` con-
> taining a `INT_ARRAY` of size `size`. This function should be used in a
> Special Action.

### get-int-array                                      **Evaluable Function**

> INTEGER get-int-array (INT_ARRAY int_array, INTEGER index)
> returns a `Term *` containing the `INTEGER` stored in the array int_array
> at `INTEGER` index.

### get-int-array-size                                **Evaluable Function**

> INTEGER get-int-array-size (INT_ARRAY int_array) returns a
> `Term *` containing the `INTEGER` value of the size of the int_array.

## 6.1.3   OP Instance Related Evaluable Functions

These evaluable functions are used to access various information about a OP
Instance.

### property-of                                        **Evaluable Function**

> Term * property-of (ATOM property, TT_OP_INSTANCE) returns
> the Term which is bound to the property `property` in the op-instance.

### fact-invoked-ops-of                               **Evaluable Function**

> LISP_LIST of op-instance fact-invoked-ops-of (LISP_LIST) is
> the fact-invoked-ops-of function.  It is defined for 1 term of type
> `LISP_LIST`. It returns a `LISP_LIST` containing the OP Instances
> which are invoked by a fact.

### get-the-decision-procedures-of   **Evaluable Function**

> LISP_LIST of op-instance get-the-decision-procedures-of (LISP_LIST
> of op-instance) returns a `LISP_LIST` containing all the OP In-
> stances of the `LISP_LIST` which have the property `decision-procedure`
> `TRUE`.

### op-instance-goal                                   **Evaluable Function**

> TT_GOAL or term op-instance-goal (TT_OP_INSTANCE) is the op-
> instance-goal function. It is defined for 1 term. It returns a `TT_GOAL`
> containing the goal which leads to the application of this OP In-
> stance, or returns the symbol NIL if it was invoked by a fact.

### safety-handlers-of                              **Evaluable Function**

LISP␣LIST of op-instance `safety-handlers-of` (LISP␣LIST of
`op-instance`) is the safety-handlers-of function. It is defined for 1
term LISP␣LIST. It returns the list of OP Instances which have the
property `SAFETY-HANDLER` set.

### preferred-of                                    **Evaluable Function**

LISP␣LIST of op-instance `preferred-of` (LISP␣LIST of TT␣OP␣INSTANCE)
is the preferred-of function. It is defined for 1 term LISP␣LIST. It
returns the list of OP Instances which have the property `PREFERRED`
set.

## 6.1.4  Fact and Goal Related Evaluable Functions

These evaluable functions are used to access various information about facts
and goals.

### GOAL-STATEMENT                                  **Evaluable Function**

GTEXPRESSION `GOAL-STATEMENT` GOAL will return the GTEXPRES-
SION of the GOAL.

### FACT-STATEMENT                                  **Evaluable Function**

GEXPRESSION `FACT-STATEMENT` FACT will return the GEXPRES-
SION of the FACT.

### GSTATEMENT-PREDICAT                             **Evaluable Function**

ATOM `GSTATEMENT-PREDICAT` GTEXPRESSION return the predicat of
the GTEXPRESSION as an ATOM.

### FSTATEMENT-PREDICAT                             **Evaluable Function**

ATOM `FSTATEMENT-PREDICAT` GEXPRESSION return the predicat of
the GEXPRESSION as an ATOM.

### GSTATEMENT-ARG                                  **Evaluable Function**

Any Term `GSTATEMENT-ARG` GTEXPRESSION, INTEGER pos returns
the pos'th argument of the GTEXPRESSION.

### FSTATEMENT-ARG                                  **Evaluable Function**

Any Term `FSTATEMENT-ARG` GEXPRESSION, INTEGER pos returns the
pos'th argument of the GEXPRESSION.

### 6.1.5   Intention Related Evaluable Functions

These evaluable functions are used to access various information about Intentions.

**get-intended-decision-procedures**                    **Evaluable Function**

> LISP_LIST of intentions get-intended-decision-procedures ()
> returns a LISP_LIST containing all the intentions of the intention
> graph which top level op-instance has the property `decision-procedure`
> true.

**number-of-intentions**                    **Evaluable Function**

> INTEGER number-of-intentions ()  returns an INTEGER which is
> the number of intentions in the intention-graph.

**get-current-intention**                    **Evaluable Function**

> TT_INTENTION get-current-intention () returns a TT_INTENTION
> containing the current intention.

**get-all-intentions**                    **Evaluable Function**

> LISP_LIST of Intentions get-all-intentions () returns a LISP_LIST
> containing all the intentions of the intention graph.

**get-other-intentions**                    **Evaluable Function**

> LISP_LIST of Intentions get-other-intentions ()  returns a
> LISP_LIST containing all the intentions of the intention graph, ex-
> cept the current intention.

**get-sleeping-intentions**                    **Evaluable Function**

> LISP_LIST of Intentions get-sleeping-intentions () returns
> a LISP_LIST containing all the intentions of the intention graph,
> which are sleeping.

**get-root-intentions**                    **Evaluable Function**

> LISP_LIST of Intentions get-root-intentions () returns a LISP_LIST
> containing all the intentions which are root of the intention graph.

**find-intention-id**                    **Evaluable Function**

> TT_INTENTION find-intention-id (ATOM tag) returns a TT_INTENTION
> containing an intention if there is at least one intention of the inten-
> tion graph, which have been tagged with the ATOM tag, and returns
> NULL otherwise.

**find-intentions-id**                    **Evaluable Function**

LISP_LIST of Intentions find-intentions-id (ATOM tag) returns a `LISP_LIST` containing all the intentions of the intention graph, which have been tagged with the `ATOM tag`.

### get-intention-priority     **Evaluable Function**

INTEGER get-intention-priority (TT_INTENTION) returns the priority of the intention.

### get-intention-time     **Evaluable Function**

INTEGER get-intention-time (TT_INTENTION) returns the time (date of creation) of the intention.

## 6.1.6   Time Related Evaluable Functions

### time     **Evaluable Function**

INTEGER time () returns an `INTEGER` which is the number of seconds since some defined time. In fact, the time origin is of no importance as it is the difference between two calls which is important.

### mtime     **Evaluable Function**

INTEGER mtime () returns an `INTEGER` which is the number of milliseconds since sometime... (in any case, it is the difference with another call which is important).

### USER-CLOCK-TICK     **Evaluable Function**

INTEGER USER-CLOCK-TICK () returns an `INTEGER` which is the number of machine TICK spent in user code used by the OPRS process since it started (`CLK_TCK`, defined in '¡time.h¿', is the number of TICK per seconds).

### SYS-CLOCK-TICK     **Evaluable Function**

INTEGER SYS-CLOCK-TICK () returns an `INTEGER` which is the number of machine TICK spent in system code (i.e. system calls) used by the OPRS process since it started (`CLK_TCK`, defined in '¡time.h¿', is the number of TICK per seconds).

### USER-SYS-CLOCK-TICK     **Evaluable Function**

INTEGER USER-SYS-CLOCK-TICK () returns an `INTEGER` which is the number of machine TICK spent in system AND in user code used by the OPRS process since it started (`CLK_TCK`, defined in '¡time.h¿', is the number of TICK per seconds).

### 6.1.7   Lisp Evaluable Functions

The following functions are defined for LISP like objects (for more information see [Lisp and Lisp-like Functions], §H, page 361). The functions `car` and `cdr` are the basic access functions in LISP. The `car` returns the first element of a list, and the `cdr` returns the rest of the list (i.e. the list without the first element).

For example in lisp, if the list `l` is equal to (`a b c`), then (`car l`) return `a`, and (`cdr l`) returns (`b c`). There are then some convenience functions which are provided. They are built following the following scheme: `c[a|d]+r`. For example (`caddr l`), is equivalent to: (`car (cdr (cdr l)))` which in our case returns `c`. OPRS provide all `c[a|d]+r` with at most three `a` or `d`.

**cons**                                        **Evaluable Function**

  LISP_LIST cons (Any Term LISP_LIST) is the cons function. It is
  defined for 2 terms, a `Any Term` and a `LISP_LIST`. It returns the new
  `LISP_LIST`.

**cons-tail**                                   **Evaluable Function**

  LISP_LIST cons-tail (Any Term LISP_LIST) is defined for 2 terms,
  a `Any Term` and a `LISP_LIST`. It adds the `Any Term` at the end of
  the `LISP_LIST`. It returns the new `LISP_LIST`.

**first**                                       **Evaluable Function**

  Any Term first (LISP_LIST)  is the first function.  It is defined
  for 1 term `LISP_LIST`. It returns the first (or car) `Any Term` of the
  `LISP_LIST`.

**car**                                         **Evaluable Function**

  Any Term car (LISP_LIST)  is defined  for 1 term `LISP_LIST`. It
  returns the car (or first) `Any Term` of the `LISP_LIST`.

**cdr**                                         **Evaluable Function**

  LISP_LIST cdr (LISP_LIST)  is defined  for 1 term `LISP_LIST`. It
  returns the cdr (or rest) `LISP_LIST` of the `LISP_LIST`.

**caar**                                        **Evaluable Function**

  Any Term caar (LISP_LIST)  is defined for 1 term `LISP_LIST`. It
  returns the caar `Any Term` of the `LISP_LIST`.

**cadr**                                        **Evaluable Function**

  Any Term cadr (LISP_LIST)  is defined for 1 term `LISP_LIST`. It
  returns the cadr (or second) `Any Term` of the `LISP_LIST`.

**cdar**                                        **Evaluable Function**

  LISP_LIST cdar (LISP_LIST)  is defined for 1 term `LISP_LIST`. It
  returns the cdar `LISP_LIST` of the `LISP_LIST`.

## cddr                                        **Evaluable Function**

LISP_LIST cddr (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the cddr LISP_LIST of the LISP_LIST.

## caaar                                       **Evaluable Function**

Any Term caaar (LISP_LIST)  defined for 1 term LISP_LIST. It re-
turns the caaar Any Term of the LISP_LIST.

## cadar                                       **Evaluable Function**

Any Term cadar (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the cadar Any Term of the LISP_LIST.

## cdaar                                       **Evaluable Function**

LISP_LIST cdaar (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the cdaar LISP_LIST of the LISP_LIST.

## cddar                                       **Evaluable Function**

LISP_LIST cddar (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the cddar LISP_LIST of the LISP_LIST.

## caadr                                       **Evaluable Function**

Any Term caadr (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the caaar Any Term of the LISP_LIST.

## caddr                                       **Evaluable Function**

Any Term caddr (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the caddr Any Term of the LISP_LIST.

## cdadr                                       **Evaluable Function**

LISP_LIST cdadr (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the cdadr LISP_LIST of the LISP_LIST.

## cdddr                                       **Evaluable Function**

LISP_LIST cdddr (LISP_LIST)  is defined for 1 term LISP_LIST. It
returns the cdddr LISP_LIST of the LISP_LIST.

## second                                      **Evaluable Function**

Any Term second (LISP_LIST) is the second function. It is defined
for 1 term LISP_LIST. It returns the second (or cadr) Any Term of
the LISP_LIST.

## last                                        **Evaluable Function**

`Any Term last (LISP_LIST)` is defined for 1 term `LISP_LIST`. It returns the last `Any Term` of the `LISP_LIST`. An error will occur if the list is empty.

## delete-from-list                    Evaluable Function

`LISP_LIST delete-from-list (Any Term, LISP_LIST)` returns a new `LISP_LIST` which is the one passed in argument in which all instance of `Any Term` have been removed. The order in the list is not preserved.

## list-difference                    Evaluable Function

`LISP_LIST list-difference (LISP_LIST LISP_LIST)` is the list-difference function. It is defined for 2 terms each of them being a `LISP_LIST`. It returns a new `LISP_LIST` which is the difference between the first one and the second one.

## list-intersection                    Evaluable Function

`LISP_LIST list-intersection (LISP_LIST LISP_LIST)` is defined for two `LISP_LIST` terms. It returns a `LISP_LIST` which is the intersection of the two `LISP_LIST`.

## list-union                    Evaluable Function

`LISP_LIST list-union (LISP_LIST LISP_LIST)` is defined for two `LISP_LIST` terms. It returns a `LISP_LIST` which is the union of the two `LISP_LIST`.

## list-difference-order                    Evaluable Function

`LISP_LIST list-difference-order (LISP_LIST LISP_LIST)` is the list-difference function. It is defined for 2 terms each of them being a `LISP_LIST`. It returns a new `LISP_LIST` which is the difference between the first one and the second one, with the element in the same order than in the first one.

## length                    Evaluable Function

`INTEGER length (LISP_LIST)` is the length function. It is defined for 1 term. It returns an `INTEGER`, the length of the `LISP_LIST`.

## select-randomly                    Evaluable Function

`Any Term select-randomly (LISP_LIST)` is the select-randomly function. It is defined for 1 term `LISP_LIST`. It returns one of its elements (a `Any Term`) chosen randomly.

## reverse                    Evaluable Function

`LISP_LIST reverse (LISP_LIST)` is the reverse function. It is defined for 1 term `LISP_LIST`. It returns the reverse of the `LISP_LIST`.

## sort-alpha              **Evaluable Function**

`LISP_LIST sort-alpha (LISP_LIST of terms)` is a sorting function. It is defined for 1 term `LISP_LIST`. It returns the same list with its element sorted alphanumericaly.

## l-list              **Evaluable Function**

`LISP_LIST l-list (TermList terms)` is the l-list function. It is defined for n terms. It returns the `LISP_LIST` containing all the TermList terms. The difference with the (`.` and `.`) reader (which can also be used to build `LISP_LIST`) is that the elements will be evaluated.

## 6.1.8    Miscellaneous Evaluable Functions

## gensym              **Evaluable Function**

`ATOM gensym ()` is the traditional gensym function (which create a new unique symbol in lisp) . It is defined for no argument. It returns a new unique `ATOM`.

## string-cat              **Evaluable Function**

`STRING string-cat (STRING STRING)` will return a `STRING` which is the concatenation of the two `STRING`s passed as argument.

## val              **Evaluable Function**

`Any Term val (Any Term)` is defined for 1 argument. It return this argument as is, without any modification. This function is very useful to force retrieving the value of a program variable so it is not bound again by the database/OP execution.

## ff-val              **Evaluable Function**

`Term * ff-val (VARIABLE GEXPRESSION)` is defined for 2 terms, a `VARIABLE` and a `GEXPRESSION`. It is used to retrieve the value of a functional fact. For example, if `POSITION` has been declared functional fact 1 (with `declare ff position 1`). Then calling (`FF-VAL $X (POSITION VALVE $X)`) will return the current `Term *` position of the `VALVE`. It returns the `ATOM NIL` if the predicate has not been declared as functional fact, or if no value were found. This makes it indistinguishable from a `NIL` real value.

## all              **Evaluable Function**

`LISP_LIST all (VARIABLE GEXPRESSION)` is the all function. It is defined for 2 terms, a `VARIABLE` and a GEXPRESSION. It returns a `LISP_LIST` containing all the possible and unique bindings of `VARIABLE` for which the GEXPRESSION is true in the database. see [Universal Quantification of Variables], §10.9, page 142 for more information on this subject.

### n-all                                                    **Evaluable Function**

`LISP_LIST of LISP_LIST n-all (LENV GEXPRESSION)` is the n-all function. It is defined for 2 terms, a `LENV` (a list of `VARIABLE`s) and a GEXPRESSION. It returns a `LISP_LIST` of `LISP_LIST` (in the same order as they are defined in LENV) containing all the possible bindings of `LENV` for which the GEXPRESSION is true in the database. N-ALL is used in OPs such as in `(!  (...  (n-all ($x $y) (foo $y $x)) ...  ))`, which returns the `LISP_LIST` of `LISP_LIST` containing the bindings of the `LENV` (example: if we have `(foo 1 2)` and `(foo 3 4)` in the database, returns `(.(.  1 2 .)  (.  3 4 .).))` which satisfies `(foo $y $x)`. See [Universal Quantification of Variables], §10.9, page 142 for more information on this subject.

### n-all-list                                               **Evaluable Function**

`LISP_LIST of LISP_LIST n-all-list (LENV GEXPRESSION)` is the n-all-list function. It is defined for 2 terms, a `LENV` (a list of `VARIABLE`) and a GEXPRESSION. It returns a `LISP_LIST` of `LISP_LIST` (in the same order as they are defined in LENV) each containing the bindings of the variable in `LENV` for which the GEXPRESSION is true in the database. N-ALL-LIST is used in OPs such as in `(!  (... (n-all-list ($x $y) (foo $y $x)) ...  ))`, which, if we have `(foo 1 2)` and `(foo 3 4)` in the database, returns: `(.(.  1 3 .) (.  2 4 .).))`. See [Universal Quantification of Variables], §10.9, page 142 for more information on this subject.

### all-pos                                                  **Evaluable Function**

`LISP_LIST all-pos (INTEGER EXPRESSION)` is defined for 2 terms, an `INTEGER` and a GEXPRESSION. It returns a `LISP_LIST` containing all the Terms in `INTEGER`'th position in all the Expression matching `EXPRESSION` passed as argument.

### mention                                                  **Evaluable Function**

`LISP_LIST of GEXPRESSION mention (AnyTerm)` is defined for 1 terms. It returns a `LISP_LIST` containing all the expressions in the database which mention the AnyTerm (even as a predicat or a function name).

### 6.1.9   Goal Building Evaluable Functions

These functions can be used to create goals which can then be intended with
the appropriate actions (see [Intending Goal Actions], §7.7.1, page 118, and see
[Intending Goals Directly], §10.6, page 140).

#### build-goal                                          Evaluable Function

> TT_GOAL build-goal (GTEXPRESSION)  is defined for 1 term con-
> taining a Gtexpression. It returns a **TT_GOAL** containing a goal which
> can then be intended directly with the appropriate goal intending
> actions (see [Intending Goal Actions], §7.7.1, page 118).

#### apply-subst-in-gtexpr                               Evaluable Function

> GTEXPRESSION apply-subst-in-gtexpr (VARIABLE AnyTerm GTEXPRESSION)
> is defined for 3 terms: a variable, a Term, and a gtexpression. It
> returns a term containing a Gtexpression, in which all occurrences
> of the variable is replaced by the term.

#### apply-subst-in-goal                                 Evaluable Function

> TT_GOAL apply-subst-in-goal (VARIABLE AnyTerm GTEXPRESSION)
> is defined for 3 terms: a variable, a Term, and a gtexpression. It
> returns a **TT_GOAL** containing a goal (created from the GTEXPRES-
> SION passed in the arguments list), and in which all occurrences of
> the variable is replaced by the term. The goal obtained can then be
> intended directly with the appropriate goal intending actions (see
> [Intending Goal Actions], §7.7.1, page 118).

## 6.2   How to Define your Own Evaluable Functions

It is fairly easy to define your own evaluable functions. To do so, you have to
write a C (or in any language you can link the object code with) function which
takes a list of terms (TermList) as arguments and returns a pointer to a new
Term. It is in fact critical that the `Term *` returned be a pointer to a NEW
Term. Using the list library functions you can then access the element of the
TermList and compute the returned value (see [Library and Kernel Functions],
§G, page 337).

   You will find various examples of user-defined evaluable functions and actions
in the file: *'user-ev-function.c'*. Here is a simple example of such function.

```
Term *toto_eval_func(TermList terms)
{
    Term *t1, *res;

    res = MAKE_OBJECT(Term);
```

```
    t1 = (Term *)get_list_pos(terms, 1);
    res->type = INTEGER;
    res->u.intval = my_function(t1);

    return res;
}
```

You can define as many evaluable functions as you want. You need to declare them in the kernel, as well as their external name (as it will appear in the OPs), and the number of arguments they take. This declaration is made in the body of the `declare_user_eval_funct` function which is called upon start up of the kernel.

```
void declare_user_eval_funct(void)
{
    make_and_declare_eval_funct("TOTO", toto_eval_func, 1);
    return;
}
```

### make_and_declare_eval_funct          Kernel User Function

> void `make_and_declare_eval_funct` (Function name, PFPT funct, int ar) is used to declare the evaluable function. You have to specify the function name, the C function which implements it, and the arity of this function.

### declare_user_eval_funct          Kernel User Function

> void `declare_user_eval_funct` (void) is the function in which you must put your call to `make_and_declare_eval_funct`. It is called upon start-up by the kernel and builds the appropriate table to map the actions and evaluable functions names and the corresponding C function. Note that the user can redefine the predefined evaluable functions or actions by using their name.

# Chapter 7

# Procedure Execution and Run Time

## 7.1   Run Time

As shown on figure 7.1, the OPRS Kernel runs the entire system. From a conceptual standpoint, it operates in a relatively simple way. At any particular time, certain goals are established and certain events occur that alter the beliefs held in the system database (1). These changes in the system goals and beliefs trigger (invoke) various OPs (2). One or more of these applicable OPs are then chosen and placed on the intention structure (3). Finally, OPRS selects a task (intention) from the root of the intention structure (4) and executes *one step* of that task (5). This results either in the performance of a primitive action (6), the establishment of a new subgoal, or the conclusion of some new belief (7).

   At this point the interpreter cycle starts again: the newly established goals and facts (if any) trigger new OPs, one or more of these are selected and placed on the intention structure, and again an intention is selected from that structure and partially executed.

## 7.2   Intention Graph

The Intention Graph is one of the most important component of the OPRS Kernel. It holds all the intentions/tasks which are currently "active". Think of it as the graph of all the tasks upon which the OPRS Kernel works at one point.

   These tasks are supposedly more or less independent. In other words, they should be working on their own problem, satisfying their own goal, responding to some events.

   Each task or intention can be visualized under the X-OPRS Kernel, if you have selected the intention graphic trace. It is represented as a box, containing
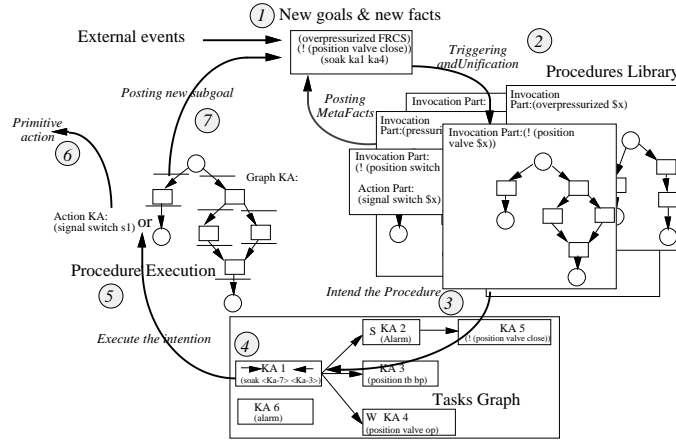
Figure 7.1: C Procedural Reasoning System main loop
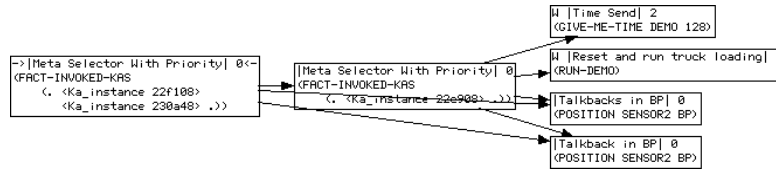


Figure 7.2: Intention Graph Development
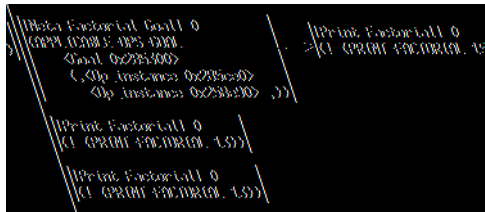


Figure 7.3: Intention Graph Development

the name of the top OP in this task (i.e. the one which was intended in this new task or intention) and the goal or the fact which is "responsible", or which triggered this OP, and lead to this intention.

This set of tasks is represented as a graph (which can be displayed in the X-OPRS as shown on figure 7.2 and figure 7.3). Only the roots of this graph can be executed. The other tasks, or intentions, must wait until they become root themselves before they can be executed. In other words, the precedence relation in this graph can be interpreted as a blocking relation. A task or intention cannot be executed until all other tasks before it disappear in the graph. This, for example, can be used by the user to create a new task which, if placed in front of the other tasks, will be executed before all the other ones. Moreover, the other tasks will resume their activity only when the new root has finished. For example, Figure 7.2 shows an example of an intention graph. Each intention is represented with a box which specifies the name of the procedure, as well as the fact or the goal responsible for its activity (i.e. the goal or the fact which lead to its execution). In this particular example, there are six tasks represented. The root of the graph is executing a meta level OP and is placed in front of another meta level OP which execution has been interrupted. These 2 meta level OPs are in front of four tasks which will resume/start their execution as soon as the two meta level OP are done.

Note that you can have more than one root in the intention graph. In this case, the system can utilize a user-defined mechanism to decide which intention should be the current intention, i.e. the one to be executed (see see [Intention Graph Sorting Predicate], §10.5, page 139). You can use priority (there is in fact a priority slot in each task), or date of creation, or whatever you think is the best heuristic to execute this intention.

The current intentions are the one which has been chosen to execute. If you have selected the intention graphic trace in X-OPRS Kernel, these intentions are recognizable because of the two small arrows surrounding their name.

Other intentions can be in three different states.

- They can be executable, in which case, they can be selected as the current intention if they are among the roots of the graph.

- They can be sleeping, which means that the most recent goal they posted was a wait goal and it has not been achieved yet (a S is visible on the graphic trace for task in this state).

- They can be awakening, which means that they were sleeping and the condition they were waiting to become true has just became true. As a result, they will receive some processing cycle time in the main loop and will become the current intention, if they are among roots of the graph.

It is important to note that each intention on the intention graph (appearing as a task box within the Intention Structure shown in Figure 7.1) represents an entire stack of invoked OPs (procedures). In particular, as each OP is executed, it establishes certain subgoals. These subgoals, in turn, invoke other OPs, and
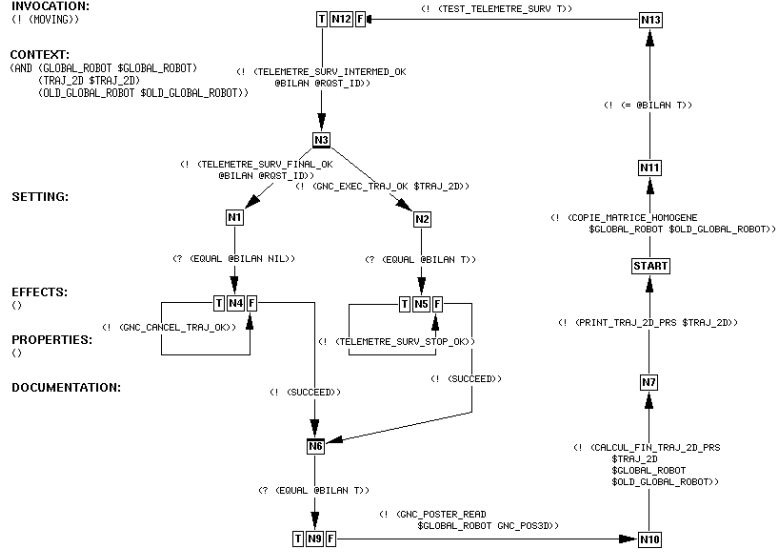
Figure 7.4: A OP with multiple threads.

so on. All the OPs so invoked form a runtime procedure stack, much like the runtime stack of so called subroutines in conventional programming languages. Where the system has only one task to perform, there is only one such stack, and consequently one task box. But where the system needs to perform multiple tasks, it spawns multiple run time stacks, executing, suspending, and resuming these in much the same manner as processes are handled in an operating system.

## 7.3   Multi Threads Execution

Multi-threads execution is linked to parallel execution in OPs. As described earlier, it is a very versatile mechanism which can be used to parallelized operations in a procedure. However, it is usually used for operations and actions performed in the same task, intention, i.e. working on a particular goal, or responding to a particular event. For example, Figure 7.4 shows a OP with two threads which are both needed to perform the goal specified in the invocation part. There are a number of OPRS runtime options linked to this mechanism, to enable/disable parallel posting of goals and parallel intending of OPs.

## 7.4 OPRS Kernel Main Loop

The OPRS main loop consists of one inner meta level reasoning loop inside the main loop. The inner loop determines the successive Sets Of Applicable OPs (SOAK), within the context of concluded beliefs on the previous SOAK. The inner loop stops when no applicable OPs are found, i.e., when the SOAK is empty.

The code of the OPRS main loop is provided below to show how meta level OPs are executed.

```
while (TRUE) {           /* Loop for ever. */
    check_stdin();       /* Check the input buffer. */
    shift_facts_goals(); /* Get new facts and new goals. */

    soak = find_soak();  /* Look for new applicable OPs. */

    while (!(list_empty(soak))) { /* While we have Applicable OPs. */
        post_soak_meta_fact(soak, oprs); /* Post the Meta Facts. */
        previous_soak = soak; /* Save the previous soak value. */

        shift_facts_goals(oprs);
        soak = find_soak(oprs);
    }

    if (!(list_empty(previous_soak))) { /* soak empty but previous soak non empty */
        post_soak_meta_fact(soak);
        if (parallel_intend) /* If parallel intending */
            intend_all(previous_soak); /* Intend all of them. */
        else
            intend(select_randomly(previous_soak));/* Intend one randomly. */
    }
    current_intention = choose_intention(); /* Choose an intention to execute. */
    execute_intention(current_intention); /* Execute one step of the intention. */

    previous_soak = soak;
}
```

When Ops are not being executed, the main loop is idle. However, the kernel monitors new events (coming from the Message Passer or from the OPRS-Server). In addition, one or more OPs might be sleeping and waiting for particular conditions to become true. The OPRS-Server kernel will wake up every main_loop_pool_sec + main_loop_pool_usec (seconds and milliseconds) to check if conditions have become true, and awake the appropriate OP.

Every main_loop_pool_sec + main_loop_pool_usec, the conditions which may have change "by themselves", like evaluable predicates depending on external conditions are checked. For example, if you have written an evaluable

predicate which tells if yes or no a particlar tank is full or not, a waiting condition on this predicate will be at least evaluated every `main_loop_pool_sec` `+main_loop_pool_usec`. As a consequence, putting a two small value (such as 0 seconds + 10 milliseconds) will put a higher burden on the kernel execution.

## 7.5   OP Applicability

The triggering mechanism, i.e., the mechanism which finds the currently applicable OPs, has been optimized for dynamic environments. The syntax and semantics of OPs require the Invocation Part to specify a goal or fact condition that will trigger the execution of the OP. In other words, only the occurrence of one of these goals or facts may render this OP applicable. When the procedures are loaded and compiled in the system, hashtables are built and used by the kernel to quickly retrieve (in constant time) the procedures which are triggered by a particular fact or a particular goal. Therefore, the kernel does not have to scan the whole library of procedures for applicable procedures, but only a subset of those procedures which are "relevant" to a new fact or a new goal. Of course, this does not prevent the system from using a full unification to check the applicability of the relevant procedures afterwards, but this is then done on a very small subset of the set of OPs.

## 7.6   Intending OP

The action of intending a OP instance, i.e. deciding to execute it is a very important step in OPRS Kernel and the X-OPRS Kernel main loop. No OP is executed before it is intended. Moreover, as seen above, there is only one way to intend a OP: when the current set of applicable OPs is empty, it intends all the OPs in the previous set of OPs or chooses one randomly in the previous set of applicable OPs, depending on the value of the `set parallel intend` (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33). Note, however, that applicable OPs can be intended by meta level OPs that are currently executing.

## 7.7   Using Action OPs

Action OPs are the basic or primitive actions of the system. Their activity range from actions such as printing a value on the screen, sending messages to another OPRS, through opening the valve of a system under OPRS control. Action OPs produce some type of activity which is implemented using a C function (or any external code linked to the OPRS Kernel).

There are two types of action OP (see [Using Action OPs], §7.7, page 114). Standard action (see [Standard Action], §4.3.3, page 66) and Special action (see [Special Action], §4.3.3, page 67).

When you define an action OP, you need to specify which C function needs to be called whenever this action OP is executed. This is very similar to defining evaluable functions.

## 7.7.1 Predefined Actions

All evaluable functions return a `Term *`. However, this term can contain different types of objects (see [Terms], §3.2, page 45). In the following description, we will indicate which type of object is contained in the `Term *` returned by the evaluable function. In addition, all the evaluable functions take a `TermList` as an argument. Whenever it is possible, we will specify the number of arguments and the type of the `Term` for each element.

To help the reader understand the descriptions for the evaluable functions in the following section, consider the `send-message` action:

**`send-message`** **Action**

> `ATOM send-message (ATOM GEXPRESSION)` is the send-message function. It is defined for two terms, an `ATOM`, the name of the recipient, and a GEXPRESSION (which must be an EXPRESSION).

The `ATOM` before the function name `send-message` is the type of the object contained in the `Term *` object that is returned by this `send-message` function. The `(ATOM GEXPRESSION)` after the function name specifies the type of the object contained in the TermList which is the argument to the `send-message` function. In this case, it means at two argument an `ATOM` (the name of the recipient) and a `GEXPRESSION` (the message itself).

Most predefined actions have a corresponding OP in *'new-default.opf'*, *'meta-intended-goal.opf'* or *'new-meta.opf'* (see [Default OPs], §F, page 315).

Predefined actions can be sorted in various categories:

### Printing Actions

These actions are used to print objects in various format.

**`print`** **Action**

> `ATOM print (ANY_TERM term)` is the print function. It is defined for 1 term. It prints the object and a cariage return.

**`print-inside`** **Action**

> `ATOM print-inside (COMPOSED_TERM term)` is defined for 1 term. The format of the term passed as argument is somewhat awkward and is here for upward compatibility with the old Lisp version of OPRS. The Term should look like this: `(FORMAT NIL "The Factorial of a is a." $X $N)))` i.e. a list containing the word `FORMAT`, then the symbol `NIL`, then a string, and finally a number of variables or terms. The a in the string will be replaced at print time with the value of the corresponding terms or variables.

## printf                                                          **Action**

`ATOM printf (COMPOSED_TERM term)` is defined for 1 term. The format of the term passed as argument is somewhat awkward but similar to the C format directive. The Term should look like this:
`(FORMAT "The Factorial of %d is %d." $X $N)))` i.e. a list containing the word `FORMAT`, then a string, and finally a number of variables or terms. The `%d` in the string will be replaced at print time with the value of the corresponding terms or variables. The following directives are supported: `%g`, `%d`, `%f` and `%s`. It does not print a cariage return at the end of the string.

**Input Actions**

## read-inside                                                     **Action**

`undefined read-inside ()` is the read-inside function. It is defined for no term. It returns the symbol read on the input. In fact, to make the read asynchronous, this function is currently implemented using a fact which is posted by the user. This fact is the basic events fact (see [Basic Events], §5.7, page 88), (`READ-RESPONSE <response>`), it will wake up the read action which will return the term `<response>`.

## read-inside-id                                                  **Action**

`undefined read-inside-id (ATOM id)` is the read-inside-id function. It is defined for one term, an `ATOM`. This function will wait for the fact : (`READ-RESPONSE-ID id <response>`), and returns the term `<response>`.

## read-inside-id-var                                              **Action**

`ATOM read-inside-id-var (ATOM id undefined response)` is the read-inside-id-var function. It is defined for two terms an `ATOM id` and an undefined Term `response`. This function will wait for the fact : (`READ-RESPONSE-ID id <response>`), if the term `response` given as second argument is an unbound variable it will bind it to the ¡response¿, else it will wait until a fact unifies it.

**Array Manipulation Actions**

## set-float-array                                                 **Action**

`ATOM set-float-array (FLOAT_ARRAY float_array, INTEGER index, FLOAT value)` will store the `FLOAT value` at index `index` in the array `float_array`.

## set-int-array                                                   **Action**

`ATOM set-int-array (INT_ARRAY int_array, INTEGER index, INTEGER value)` will store the `INTEGER value` at index `index` in the array `int_array`.

**Intending OP Instance Actions**

## intend-op                                                    Action

`ATOM intend-op (TT_OP_INSTANCE)` is the intend-op function. It is defined for one term, a `TT_OP_INSTANCE`. It will intend it, after the current intention.

### intend-op-with-priority                                     Action

`ATOM intend-op-with-priority (TT_OP_INSTANCE and INTEGER)` is defined for two terms, a `TT_OP_INSTANCE` and an `INTEGER`. The OP Instance will be intended, after the current intention, with the priority specified in the `INTEGER`.

### intend-op-after                                             Action

`ATOM intend-op-after (TT_OP_INSTANCE opi, LISP_LIST of intentions after)` is defined for two terms, a `TT_OP_INSTANCE` and a `LISP_LIST` of intentions. It will intend `opi` after all the intentions in `after`.

### intend-op-with-priority-after                               Action

`ATOM intend-op-with-priority-after (TT_OP_INSTANCE opi, INTEGER priority, LISP_LIST of intentions after)` is defined for three terms; `opi` a `TT_OP_INSTANCE`, `priority` an `INTEGER` and `after` a `LISP_LIST` of intentions. The `opi` will be intended with the priority `priority` after all the intentions in `after`.

### intend-op-before-after                                      Action

`ATOM intend-op-before-after (TT_OP_INSTANCE opi,LISP_LIST of intentions before, LISP_LIST of intentions after)` is defined for three terms, a `TT_OP_INSTANCE opi` which will be intended, before all the intentions in `before` and after all the intentions in `after`.

### intend-op-after-before                                      Action

`ATOM intend-op-after-before (TT_OP_INSTANCE opi, LISP_LIST of intentions after, LISP_LIST of intentions before)` is defined for three terms, a `TT_OP_INSTANCE opi` which will be intended, after all the intentions in `after` and before all the intentions in `before`.

### intend-op-with-priority-after-before                        Action

ATOM intend-op-with-priority-after-before (TT_OP_INSTANCE
opi, INTEGER priority, LISP_LIST of intentions after, LISP_LIST
of intentions before) is defined for four terms, a TT_OP_INSTANCE
opi which will be intended with the priority priority, after all the
intentions in after and before all the intentions in before.

## intend-all-ops-as-root                              Action

ATOM intend-all-ops-as-root (LISP_LIST of op-instance) is
defined for one term, a LISP_LIST of op-instance. Each OP Instance
will be intended as a root of the Intention Graph.

## intend-all-ops                                      Action

ATOM intend-all-ops (LISP_LIST of op-instance) is defined for
one term, a LISP_LIST of op-instance. Each OP Instance will be in-
tended in the Intention Graph, after the current intention.

## intend-all-ops-after                                Action

ATOM intend-all-ops-after (LISP_LIST of op-instance), LISP_LIST
of intentions after) is defined for two terms, a LISP_LIST of op-
instance and after a LISP_LIST of intentions.  Each OP Instance
will be intended in the Intention Graph, after all the intentions in
after.

**Intending Goal Actions**

## intend-all-goals-//                                 Action

ATOM intend-all-goals-// (LISP_LIST of goal)  is defined for
one term, a LISP_LIST of goals. Each goal will be intended in parallel
after the current intention.

## intend-all-goals-//-as-roots                        Action

ATOM intend-all-goals-//-as-roots (LISP_LIST of goal) is de-
fined for one term, a LISP_LIST of goals. Each goal will be intended
as a root of the intention graph.

## intend-all-goals-//-after                           Action

ATOM intend-all-goals-//-after (LISP_LIST of goal), LISP_LIST
of intentions after)  is defined for two terms, a LISP_LIST of
goals and after a LISP_LIST of intentions.  Each goal will be in-
tended in the Intention Graph, after all the intentions in after.

## intend-all-goals-//-as-roots-with-priority **Action**

ATOM `intend-all-goals-//-as-roots-with-priority` (`LISP_LIST`
`of op-instance, LISP_LIST of Term INTEGER`) is defined for two
terms, a `LISP_LIST` of goals and a `LISP_LIST` of Term `INTEGER`. Each
goal will be intended as a root with the priority specified in the
`INTEGER` list.

## `intend-all-goals-//-after-roots` <span style="float:right">Action</span>

ATOM `intend-all-goals-//-after-roots` (`LISP_LIST of goal`) is
defined for one term, a `LISP_LIST` of goals. Each goal will be in-
tended in parallel after the root(s) of the intention graph.

## `intend-goal` <span style="float:right">Action</span>

ATOM `intend-goal` (`TT_GOAL goal`) is defined for one term, a `TT_GOAL`
`goal` which will be intended after the current intention.

## `intend-goal-with-priority` <span style="float:right">Action</span>

ATOM `intend-goal-with-priority` (`TT_GOAL goal, INTEGER priority`)
is defined for two terms, a `TT_GOAL goal` which will be intended with
the priority `priority`, after the current intention.

## `intend-goal-after-before` <span style="float:right">Action</span>

ATOM `intend-goal-after-before` (`TT_GOAL goal, LISP_LIST of`
`intentions after, LISP_LIST of intentions before`) is defined
for three terms, a `TT_GOAL goal` which will be intended after all the
intentions in `after` and before all the intentions in `before`.

## `intend-goal-with-priority-after-before` <span style="float:right">Action</span>

ATOM `intend-goal-with-priority-after-before` (`TT_GOAL goal,`
`INTEGER priority, LISP_LIST of intentions after, LISP_LIST`
`of intentions before`) is defined for four terms, a `TT_GOAL goal`
which will be intended with the priority `priority`, after all the in-
tentions in `after` and before all the intentions in `before`.

**Intentions Manipulation Actions**

## `tag-current-intention` <span style="float:right">Action</span>

ATOM `tag-current-intention ATOM tag` will tag the current in-
tention, i.e. the intention in which it is executed, with the `ATOM`
passed in argument.

## `kill-other-intentions` <span style="float:right">Action</span>

ATOM `kill-other-intentions ()` is defined for no argument. It
will kill all the other intentions in the intention graph, except the
current one.

### kill-intentions                                    Action

`ATOM kill-intentions (LISP_LIST of Intentions)` will kill all
the intentions in the `LISP_LIST`, except itself.

### kill-intention                                     Action

`ATOM kill-intention (TT_INTENTION)` will kill the intention in
the `TT_INTENTION`, except itself.

### asleep-intentions                                  Action

`ATOM asleep-intentions (LISP_LIST of Intentions and ATOM wake-up-tag)`
will asleep all the intentions in the `LISP_LIST`, but cannot asleep it-
self. Theses intentions will be waked up by the basic event facts (see
[Basic Events], §5.7, page 88), `(INTENTION-WAKE-UP wake-up-tag)`.

### asleep-intention                                   Action

`ATOM asleep-intention (TT_INTENTION and ATOM wake-up-tag)`
will asleep the intention in the `TT_INTENTION`, but cannot asleep it-
self. This intention will be waked up by the basic event facts (see
[Basic Events], §5.7, page 88), `(INTENTION-WAKE-UP wake-up-tag)`.

### wake-up-intention                                  Action

`ATOM wake-up-intention (ATOM wake-up-tag)` will just post the
basic event facts (see [Basic Events], §5.7, page 88), `(INTENTION-WAKE-UP`
`wake-up-tag)`, to wake up intentions asleep by asleep-intention or
asleep-intentions.

### asleep-intentions-cond                             Action

`ATOM asleep-intentions-cond (LISP_LIST of Intentions and GEXPRESSION`
`condition)` will asleep all the intentions in the `LISP_LIST`, but
cannot asleep itself. Theses intentions will be waked up when the
`condition` will become true.

### asleep-intention-cond                              Action

`ATOM asleep-intention-cond (TT_INTENTION and GEXPRESSION condition)`
will asleep the intention in the `TT_INTENTION`, but cannot asleep
itself. This intention will be waked up when the `condition` will
become true.

### set-intention-priority                            Action

`ATOM set-intention-priority (TT_INTENTION and INTEGER new-priority)`
set the priority of the intention to the value `new-priority`.

### apply-sort-predicate-to-all                        Action

ATOM `apply-sort-predicate-to-all` `()` is defined for no Term, It will apply the current sorting predicate (see [Intention Graph Sorting Predicate], §10.5, page 139), to all the intentions of the graph.

### sort-intention-priority                          **Action**

ATOM `sort-intention-priority` `()` is defined for no Term, It will set the sorting predicate (see [Intention Graph Sorting Predicate], §10.5, page 139), to sort by priority.

### sort-intention-time                              **Action**

ATOM `sort-intention-time` `()` is defined for no Term, It will set the sorting predicate (see [Intention Graph Sorting Predicate], §10.5, page 139), to sort by time (date of creation).

### sort-intention-priority-time                     **Action**

ATOM `sort-intention-priority-time` `()` is defined for no Term, it will set the sorting predicate (see [Intention Graph Sorting Predicate], §10.5, page 139), to sort by priority then if two intentions have the same priority, by creation time.

### sort-intention-none                              **Action**

ATOM `sort-intention-none` `()` is defined for no Term, It will unset the sorting predicate (see [Intention Graph Sorting Predicate], §10.5, page 139).

## Miscellaneous Actions

### send-message                                     **Action**

ATOM `send-message` (ATOM GEXPRESSION) is the send-message function. It is defined for two terms, an `ATOM`, the name of the recipient, and a GEXPRESSION (which must be an EXPRESSION).

### broadcast-message                                **Action**

ATOM `broadcast-message` (GEXPRESSION) is the broadcast-message function. It is defined for one term, the GEXPRESSION (which must be an EXPRESSION) to send.

### multicast-message                                **Action**

ATOM `multicast-message` (LISP_LIST of ATOM GEXPRESSION) is defined for two term, the LISP_LIST of the recipients name (as ATOM) and the GEXPRESSION (which must be an EXPRESSION) to send.

### send-string                                      **Action**

`ATOM send-string (ATOM STRING)` is defined for two terms, an `ATOM`, the name of the recipient, and a `STRING`, which will be sent to the recipient.

### execute-command                                        Action

`ATOM execute-command (STRING)` is the execute-command action. It will execute the command (see [OPRS Kernel Commands], §2, page 27), as if it was typed by the user.

### start-critical-section                                 Action

`ATOM start-critical-section ()` is defined for no Term, it will start a critical section (see [Critical Section], §10.8, page 142).

### end-critical-section                                   Action

`ATOM end-critical-section ()` is defined for no Term, it will end the current critical section (see [Critical Section], §10.8, page 142).

### fail                                                    Action

`ATOM fail ()` Does nothing, just fail, i.e. return the `nil` symbol. A OP (called |`Fail`|) is defined in *'new-default.opf'* and call this action. This action can be useful when you need to explicitly fail a branch of execution.

### succeed                                                 Action

`ATOM succeed ()` Does nothing, just succeed, i.e. return the `T` symbol. A OP (called |`Succeed`|) is defined in *'new-default.opf'* and call this action. This action can be useful when you need an extra edge which does nothing between two nodes.

### test-and-set                                            Action

`ATOM test-and-set (GTEXPRESSION)` is the test-and-set function. It is defined for one `GTEXPRESSION`. It returns the result of posting the `GTEXPRESSION` as a goal. Therefore it can return `T`, `:wait` or `NIL`. This function should only be used by the —Test and Set— OP. This function has no reason to be ever since the "IF-THEN-ELSE" node has been introduced.

## 7.7.2   How to Define your Own Actions

It is fairly easy to define your own actions. To do so, you have to write a C function which takes a list of terms (TermList) as arguments and returns a pointer to a new Term. It is in fact critical that the `Term *` returned be a pointer to a NEW Term. Using the list library functions you can then access the element of the TermList and compute the returned value.

The value returned by the evaluation of this function is meaningful. It must be a pointer to term, and this term will be freed by the caller. If it returns the term symbol `:wait`, the function has not completed its execution and it should be called again later. If it returns the term symbol `nil`, then the action is considered as failed and the OP failed the goal it was working on. Any other term value returned is considered as a success, and the action OP is successful.

Keep in mind that you can define Special Actions which call C functions defined for evaluable functions (see [Predefined Evaluable Functions], §6.1, page 95). However, the compiler will warn you of such practice.

You will find various examples of user-defined evaluable functions and actions in the file: *'user-action.c'*.

```
Term *action_bar_foo(TermList terms)
{
    Term *t1, *t2, *res;

    res = MAKE_OBJECT(Term); /* This will make a Term. */

    t1 = (Term *)get_list_pos(terms, 1);
    t2 = (Term *)get_list_pos(terms, 2);
    if ((t1->type != ATOM) || (t2->type != TERM_COMP))
        fprintf(stderr, "Expecting an ATOM and a TERM_COMP in action_bar_foo.");
    my_action(t2->u.term, t1->u.id);
    res->type = ATOM;
    res->u.id = lisp_t_sym;    /* Return T */

    return res;
}
```

You can define as many evaluable functions and actions as you want. You need to declare them in the kernel, as well as their external name (as it will appear in the OPs), and the number of arguments they take. This declaration is made in the body of the **declare_user_eval_funct** function which is called upon start up of the kernel.

```
void declare_user_action(void)
{
    make_and_declare_action("BAR-FOO", action_bar_foo, 2);
    return;
}
```

### make_and_declare_action                    **Kernel User Function**

> void **make_and_declare_action** (Function name, PFPT funct, int ar) is used to declare an Action. You have to specify the function name, the C function which implements it, and the arity of this function.

**declare_user_action**                    **Kernel User Function**

> void declare_user_action (void)  is the function in which you
> must put your call to make_and_declare_action. It is called upon
> start-up by the kernel and builds the appropriate table to map the
> actions and the corresponding C function. Note that the user can
> redefine the predefined actions by using their name.

## 7.8   Graph OP Traversal

When a OP is executed, the execution starts at the START node. Then it proceeds from one node to an adjacent one if the goal labelling the edge connecting the two nodes can be achieved. This goes until the control reach an end node, i.e. a node without outgoing edges. If there are more than one edge outgoing from the current node, the system will try them one after the other until it findd one it can achieve. Note however, that there are no a priori predefined path... The system will try them in a random order. Note also that upon success, untested path are not kept as possible backtrack point; OPRS Kernel does not backtrack.

Since version 1.1, there are split and join nodes (see [Split and Join Node], §4.3.2, page 65). From the graph traversal point of view, these new nodes introduce parallel execution. A split mode indicates that all the outgoing branches have to be traversed/executed in their own thread, and a join node indicates that as many execution threads as there are ingoing branches must reach this node before execution can proceed from it.

## 7.9   Goal Commitment

There is a real goal commitment in OPRS. When a goal is posted, it will be reposted automatically until the system decides that it has been failed.

A goal is failed, when the main loop cannot find any applicable OPs for this goal. In fact, this is a little bit more tricky. It is when the main loop cannot find any applicable OP which have not already been tried with exactly the same binding environment.

If for example you are reaching an edge in the execution of a OP where the goal G1 has to be achieved. This goal will first be posted, and the system will look for all the applicable OP for this goal (and possibly other goals or facts). We shall assume that three OPs are applicable to satisfy this goal. At this point, either one of these applicable OPs is intended, or none. If none are intended (presumably because a more important one has been intended), when the system will resume the execution of the intention in which G1 appeared, then it will realize that this goal has not been failed (nor has it been achieved in fact, or by chance by a side effect of other OP execution). Therefore, it will be reposted, and the applicable OP for this goal will be recomputed (thus taking into account new changes in the world). Now, we shall assume then again 3

OPs are applicable and `OP1` one of them is chosen. The system will then try to execute `OP1`. If it succeeds, then `G1` is achieved, and the execution of the original OP can resume. If it fails, then again, when the system will resume the execution of the intention in which `G1` appeared, it will realize that this goal has not been failed. It will repost the goal G1, however, this time, upon looking for applicable OPs, and if `OP1` is still applicable with exactly the same binding then previously found, then `OP1` will not be put in the applicable OP. In other words, you do not try the same thing twice, and you consider that a goal is failed when everything has been tried.

## 7.10  Message Passing

Message passing is the basic communication mechanism in OPRS. It is very easy to use and provides a simple and powerful mean to communicate with other OPRS modules or external modules.

# Chapter 8

# Parallel Execution of OPs in OPRS

New in version 1.1, one can write OPs with parallel execution in various branches of the OPs. This new feature has a number of consequences on the way OPs can be executed, and on the performance of the system.

## 8.1   Changes in the OP Representation

Parallel execution, or conjunctive execution, is represented using split and join nodes.

We illustrate this new construction with concrete examples. Figure 8.1 shows a OP which computes Fibonacci. In this particular case, the two recursive calls can be done in parallel.

Figure 8.2 shows an example of such construct. In this particular example we mixed the "IF-THEN-ELSE" construction with the split node. The F node of the NO "IF-THEN-ELSE" node is a split node (this is represented with the thick bottom of the node). Similarly, the S4 node is a join node. Basically, a split node splits as many threads as it has outgoing edges, and a join node merges as many thread as it has ingoing edges.

In this particular example, for each recursive call, we will get two execution threads. As a consequence, the number of threads active in the system can raise dramatically.

Note that if one of the two parallel threads fail, the whole OP fails. This is the reason why this construct is also called conjunctive execution.

Last, Figure 8.3 presents a OP withdrawn from a mobile robot execution control application. It shows an example of a procedure implementing a surveillance. Out from node N3, two threads are started, one to execute a trajectory, another one to set a monitoring. If the trajectory executes properly, it then stops the monitoring task which will return without modifying the @BILAN variable. Otherwise, if the monitoring detects an obstacle, it returns and the @BILAN

Figure 8.1: A OP to compute Fibonacci (without parallelism).



Figure 8.2: A OP to compute Fibonacci (with parallelism).

Figure 8.3: A OP with two threads, one monitoring, the other one executing.

variable is set to nil, which leads this thread to cancel the trajectory execution.

## 8.2 New Traces and New Options

The introduction of parallel execution in OPRS has introduced a number of new options and trace.

As for the new options, there are two of them (see [OPRS Kernel Run Option Commands], §2.6, page 32).

- `set parallel post on|off` Turn on or off the parallel posting of goals. When this option is `ON`, one goal for each thread active in the current intention will be posted. In the Fibonacci example presented above, it means that both goals `(! (FIBONACCI (- $N 1) $RES2))` and `(! (FIBONACCI (- $N 2) $RES3))`, will be posted in parallel.

- `set parallel intend on|off` Turn on or off the parallel intending of OP instance. When this option is `ON`, all the OP Instances found in the `PREVIOUS_SOAK` (see [OPRS Kernel Main Loop], §7.4, page 113) are intended. **'This option has some very important consequences on the standard behavior of the kernel'**. In any case the kernel always checks that a particular OP Instance has not been already intended before intending it. This is to make sure, for example, that you do not intend again from a meta level OP, a OP instance already intended by the main

loop or another meta level OP. Moreover, the kernel always check that a OP Instance intended for a particular goal is intended in the proper place, i.e. it is not intended if there is already another OP instance which has been intended for the same goal. Note however, that it may be intended later if it is still applicable and if the other one has failed.

- `set parallel intention on|off` Turn on or off the parallel intention execution.

There is also a new trace associated to the new forking/joining mechanism:

- `trace thread on|off` Turn on or off traces on thread creation and merging.

## 8.3    Performance Considerations

The parallel execution of OPs has a number of consequences on the performance of the system. The positive consequence, is that parallel posting and parallel intending (see [New Traces and New Options], §8.2, page 129) can increase the performance as the main loop does less "loop" and therefore some constant overhead of the main loop is called less time. The drawback, is that the main loop becomes longer... and the reaction time of the system becomes bigger. The Fibonacci example is a good example of this behaviors. If you post a goal requesting a large Fibonacci number, the system will fork a large number of threads, and will post and intend for each of them.... At some time, you can have hundreds of threads active, and this can lead to a large reaction time of the system.

# Chapter 9

# Meta Level Reasoning

What is meta level reasoning? By meta level reasoning, we mean all the mechanisms which enable the user to control various parts or mechanisms of the OPRS Kernel main loop. This definition is very large and we will see in this chapter that it covers many aspects of the OPRS system. The most used meta level mechanism is probably the one deciding which OP to intend when more than one OP are applicable. Besides, the meta level reasoning can be used to achieve other goals while developing a real world application. For example, by controlling the way OPs are intended, one can use meta level OPs to implement priority mechanism, or decision theory mechanism, or evidential reasoning, etc. The field is open, and OPRS provides very powerful mechanisms to implement advanced reasoning.

## 9.1   SOAK and other Meta Facts

SOAK stands for `Set Of Applicable OPs`. This fact as well as `APPLICABLE-OPS-FACT`, `APPLICABLE-OPS-GOAL`, `FACT-INVOKED-OPS` and `GOAL-INVOKED-OPS`, are automatically concluded by the kernel while it computes the current set of applicable OPs. They are basic event facts (see [Basic Events], §5.7, page 88), therefore, they are not "remembered" in the database, but can trigger OPs.

This mechanism is under the control of a flag, that the user can turn on or off by using the command `set meta on|off` (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33). If it is turned `off`, none of theses Meta Facts will be concluded. If it is turned `on`, the individually selected meta facts will be concluded.

- (SOAK list-of-op-instances). `list-of-op-instances` contains the `LISP_LIST` of all the op-instances applicable in this loop. Note that, as shown in the main loop presented in [OPRS Kernel Main Loop], §7.4, page 113, the soak is updated at each loop and is not incremented with the new OP applicable in the current loop. This fact can be used to trigger Meta OPs which want to decide and sort which OP should be intended, and

131

which one can be forgotten, or postponed, or whatever you decide to do with them. Keep in mind that, with meta level reasoning, the limit of the application is your mind...

The posting of this Meta Fact is under the control of the `soak` option (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

- `(APPLICABLE-OPS-FACT fact list-of-op-instances)`. `APPLICABLE-OPS-FACT` contains all the OPs applicable (in `list-of-op-instances`) because of the `fact`. `fact` is a `TT_FACT` Term. `list-of-op-instances` is a `LISP_LIST` Term containing OP-Instances. This fact is not concluded for empty lists of applicable OPs. In other words, if no OP is applicable because of a fact, then we do not conclude `(APPLICABLE-OPS-FACT fact (.  .))`. The reason is left as an exercise to the reader (hint: if we were to conclude such a fact, we would blow up the memory space of the system in a few minutes...).

  The posting of this Meta Fact is under the control of the `app_ops_fact` option (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

- `(APPLICABLE-OPS-GOAL goal list-of-op-instances)`. `APPLICABLE-OPS-GOAL` contains all the OPs applicable (in `list-of-op-instances`) because of the `goal`. `goal` is a `TT_GOAL` Term. `list-of-op-instances` is a `LISP_LIST` Term containing the OP-Instances. This fact can be used to sort out which OP is best suited to fulfill the goal. Due to the goal commitment (see [Goal Commitment], §7.9, page 124), it is of little, if no, interest to keep OPs you have not intended in this list for future use.

  The posting of this Meta Fact is under the control of the `app_ops_goal` option (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

- `(FACT-INVOKED-OPS fact-invoked-ops)`. This meta fact points at the list of all the OPs applicable because of a fact (any fact) in the previous loop.

  The posting of this Meta Fact is under the control of the `fact_inv` option (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

- `(GOAL-INVOKED-OPS goal-invoked-ops)`. This meta fact points at the list of all the OPs applicable because of a goal (any goal) in the previous loop.

  The posting of this Meta Fact is under the control of the `goal_inv` option (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

## 9.2   Writing Meta Level OPs

There is nothing really special about writing Meta Level OPs. They are standard OPs and nothing particular can distinguish them from other OPs.

However, one often wants the Meta Level OP to be executed "before" any other already executing intention. To do so, the OPRS Kernel provides the appropriate mechanism. If you put the property `DECISION-PROCEDURE` to `T` in the properties list of a OP, this OP, if intended in a new intention, will be intended as a root of the intention graph before all the current roots of the intention graph. This will indeed ensure that this OP gets executed before all the others. Keep in mind, however, that this does not prevent the execution of this OP from being interrupted... In other words, if the OPRS Kernel decides (because of new events) to still intend new OPs while this OP is executed, it may interrupt the execution of this OP.

At last, let us stress the fact that writing Meta Level OPs is an interesting and powerful, but dangerous exercise. For example, one can easily see the consequence of a Meta Level OP which would be applicable to itself... It would lead the OPRS Kernel in an infinite loop from which it would never come back, trying to figure out for ever which OP is applicable, never reaching the fix point of the recursion (which is to have no OP applicable in one loop, see [OPRS Kernel Main Loop], §7.4, page 113).

Another more subtle but not less dangerous syndrome is the Meta Level OP which execution leads to its own applicability... For instance, assume you have a Meta Level OP which is able to decide what to do when you have two OPs applicable for the same goal. Now, assume that this very OP posts a goal for which there are two applicable OPs... Well... You end up with an ever growing intention graph. At least, in this case, you see the intention graph growing indefinitely if you graphic trace it.

There are many ways to avoid these pitfalls: most of them consist in using appropriate properties to guard against oneself, or to only apply to a set of OPs which have these properties.

## 9.3   Other Aspects of the Meta Level

As explained earlier, there are other ways to use Meta Level OPs, such as insuring mutual exclusion on non sharable resources, for example.

# Chapter 10

# Advanced Features

There are a number of features of OPRS that are considered advanced, in the sense that they are not required by the "standard application" but can be used by in some particular situations. This chapter describes these features and explain when they can be used and for which purpose.

## 10.1   OP Properties

Properties in OPs are basically information which is linked to the OPs and their execution environment. Keep in mind that the value of a property can be any term with evaluable functions and variables. As a result, the value of a property is a run time value (i.e. the value of the property is the value resulting of its evaluation at the time it is retrieved), and can heavily depend on the moment at which it is evaluated.

Properties are very much linked to the use of Meta Level OPs. When one writes Meta Level OPs, one often makes reference to some user-defined properties. Such properties can be priority, Bayesian information, utility information or even resources they consume. As for Meta Level reasoning, properties form a general powerful mechanism which is given to the user to implement any desired control algorithm or heuristic.

## 10.2   User Hooks

There are two user hooks provided to the user to change global variables or to initialize its own data structure (they are defined in *'default-user-external_f.h'*).

**start_kernel_user_hook**                    **Kernel User Function**

> `void start_kernel_user_hook (void)` is the function which is called by the kernel upon start-up to eventually give a chance to the user to set his own global value predicate. You can put in it the affectation of intention_list_sort_predicate to the address of your function.

You can also change the value of global such as `main_loop_pool_sec`
or `main_loop_pool_usec` (see [Important Variables], §G.1.2, page
338). Last, you can make any call required by your application...
This hook is executed before any loading command specified with
the `-x` argument.

Example:

```
void start_kernel_user_hook()
{
    intention_scheduler = &intention_scheduler_time_sharing;
    main_loop_pool_sec = 0L;
    main_loop_pool_usec = 10000L; /* 10 milliseconds */
    create_my_widget_tree(x_oprs_top_level_widget);
    create_my_own_objects();
    start_my_own_server();
}
```

### end_kernel_user_hook                Kernel User Function

`void end_kernel_user_hook (void)` is the function which is called
by the kernel upon exiting to eventually give a chance to the user to
properly terminate services, close connections, free global structure
or memory, etc.

Example:

```
void end_kernel_user_hook()
{
    free_my_own_objects();
    shutdown_my_own_server();
}
```

## 10.3   User Code Error Handler

There are two functions provided to the user to report fatal or recoverable error
(their prototype is defined in *'oprs-error_f-pub.h'*). It is strongly advised to use
these functions.

### report_fatal_external_error        Kernel User Function

`void report_fatal_external_error (char *error_message)` is
the function used to report fatal errors. This function does not
return (because the error is considered to be fatal, and terefore the
execution cannot proceed from this point). In other words, it will
"jump" to a safe place and try to recover from the error. Accoring
to the current state in which the error occurs the OPRS Kernel will
take some action to protect the operations still valid and running in

the kernel. Nevertheless, most of the time, such an error will lead to the failure/cancel of the intention in which it occured.

Many informations are displayed by the kernel on the error itself and the various operations which resulted of this error.

The consequence of such fatal error may be so dramatic that you should not attempt to continue execution. The problem which lead to error should be fixed first.

Example:

```
Term *float_to_int_ef(TermList terms)
{
     Term *res, *t1;

     res = MAKE_OBJECT(Term);

     t1 = (Term *)get_list_pos(terms, 1);

     if (t1->type != FLOAT) {
          report_fatal_external_error("Expecting a FLOAT in float_to_int_ef.");
     }

     res->type = INTEGER;
     res->u.intval = (int)t1->u.doubleval;

     return res;
}
```

## report_recoverable_external_error        Kernel User Function

void **report_recoverable_external_error** (char *error_message) is the function used to report recoverable errors. This function returns, and the user can then decide what to do, which reasonable value to return, etc.

Here also, many informations are displayed on the screen to provide the user with valuable hints as where the error occured and in which condition.

Example:

```
Term *action_execute_command(TermList terms)
{
     Term *term, *res;

     res = MAKE_OBJECT(Term);
     res->type = ATOM;
```

```
        term = (Term *)get_list_pos(terms,1);
        if (term->type != STRING) {
            report_recoverable_external_error("Expecting a STRING in action_execute_
            res->u.id = nil_sym;
        } else {
            PString command;

            command = (char *)OPRS_MALLOC((strlen(term->u.string) + 2) * sizeof(char
            sprintf(command, "%s\n", term->u.string);
            send_command_to_parser(command);
            OPRS_FREE(command);
            res->u.id = lisp_t_sym;
        }
        return res;
    }
```

## 10.4   Intention Graph Scheduling

The OPRS Kernel provides a mechanism to schedule the root of the intention
graph, in parallel intention execution mode or not. To do so, the user needs to
define his own scheduler functions with the following prototypes:

### my_intention_list_scheduler          Kernel User Function

Intention_List my_intention_list_scheduler (Intention_List
il)  is the prototype of an intention_list scheduler. It takes an Inten-
tion_List as argument and should return one (probably the same). In
any case, the first runnable intention in the returned list will become
the current intention.

### intention_scheduler                          Kernel Variable

extern PFPL intention_scheduler  is the global variable which
points at the appropriate function used when the system is not in
parallel intention execution. If it is set to NULL, then no scheduling
is used, and the sorting predicate may be used (if set).

### intention_par_scheduler                      Kernel Variable

extern PFPL intention_par_scheduler is the global variable which
points at the appropriate function used when the kernel is in parallel
execution mode. If it is set to NULL, then no scheduling is used,
and all the root of the intention graph are executed.

Here is an example of how to set the scheduler (withdrawn from *'default-
user-external.c'*):

```
void start_kernel_user_hook()
{
     intention_scheduler = &intention_scheduler_time_sharing;
}
```

Here is an example of a scheduler function, which will schedule a new intention every 6 times it is called:

```
Intention_List intention_scheduler_time_sharing(Intention_List l)
{
     static int loop = 0;

     if (loop++ == 6) {
          Intention *i =  (Intention *)get_from_head(l);
          add_to_tail(l,i);
          loop = 0;
     }
     return l;
}
```

All these functions and examples are defined in *'user-external.c'*.

## 10.5   Intention Graph Sorting Predicate

The OPRS Kernel provides a mechanism to "sort" the root of the intention graph under some user specified criteria.

This mechanism is used if and only if no scheduler is defined, and the kernel is not in parallel intention execution mode. In other words, `intention_scheduler` has to be set to `NULL`, and the `Parallel Intention Execution` flag must be off..

To define a sorting predicate, the user needs to define his own sorting functions with the following prototype:

**my_intention_list_sort**                         **Kernel User Function**

> `PBoolean my_intention_list_sort (Intention *i1, Intention *i2)` is the prototype of an intention_list sorting predicates. Note that the intention root of the graph, which "maximizes" this predicate is the one which will become the current intention.

Of course, the name can be changed, but it takes two pointers to an Intention and returns a PBoolean (`TRUE` or `FALSE`).

**intention_list_sort_predicate**                 **Kernel Variable**

> `extern PFB intention_list_sort_predicate`  is the global variable which points at the appropriate function. If it is set to NULL, then the list is not sorted at all.

Here is an example of how to modify the default sorting predicate (withdrawn from *'default-user-external.c'*):

```
void start_kernel_user_hook()
{
    intention_scheduler = NULL; /* Set it to NULL, or the sorting predicate
                                 * will not be used at all. */
    intention_list_sort_predicate = &my_intention_list_sort;
}
```

Here is an example of a sorting function, which will sort the intentions by priority:

```
PBoolean my_intention_list_sort_example(Intention *i1, Intention *i2)
{
    return (intention_priority(i1) > intention_priority(i2));
}
```

Note that, if you want to keep the intention list undisturbed by the sorting algorithm, when the list is already sorted, then your sorting predicate must return `FALSE` when two intentions are equivalent (in the previous example we use `>` rather than `>=`).

All these functions and examples are defined in *'user-external.c'*.


## 10.6   Intending Goals Directly

There are more than one way to achieve a goal. The standard mechanism consists in posting a goal, finding one or more applicable OPs for this goal.

However, there is at least one other way to intend a more imperative goal. It is called "goal intending" as opposed to the previous method we called "applicable OP intending". In the "goal intending" method, you directly intend a new intention which has to achieve a particular goal. The kernel may discover later that there are no applicable OPs to achieve this goal, but that will be seen below.

The main difference is that you intend before even knowing if some OPs will be applicable or not. The "goal intending" method is seldom used, but can be very useful. For instance, one can post parallel goals, each of them in its own intention by using this technique. An example of such program is given in the *'fact-meta.opf'* file.

The meta level OP required to perform this goal intending is in the file *'meta-intended-goal.opf'*. The actions performed are defined in [Intending Goal Actions], §7.7.1, page 118, and the functions to build these goals are defined in [Goal Building Evaluable Functions], §6.1.9, page 107.

## 10.7   Current and Quote

The `current` and `quote` mechanism is quite simple. Basically, one may want to postpone or immediately run the evaluation of some expressions (presumably in an evaluable function term) in a posted goal. For example, if one posts the goal `(!  (foo (+ 3 4)))`, then the `+` operation could be carried on only when required, presumably when the system checks if it is true in the database. As long as it is possible, the system could try to achieve `(!  (foo (+ 3 4)))`. In this particular case, it does not really make much difference, because `(+ 3 4)` is always 7, independently of when you are doing it. But let's assume now that we want to achieve a goal such as `(?  (> (pressure-of tk1) 245))`, and that the fluent `pressure-of` can return a different value for `tk1` according to the moment at which you ask for the value. Then, obviously the goal `(?  (> (pressure-of tk1) 245))` can have a different interpretation according to the moment at which you post and interpret it. . . As said earlier, by default, now the kernel always evaluates evaluable functions (and their arguments) at posting time (note that this is a new behavior, and it differs from SRI's OPRS). But the user can prevent a fluent to be evaluated at the time the goal is posted. To do so, one just needs to use the `quote` "function" (actually, it is not really a function but for this purpose, it can be considered as one). This `quote` function defines a context in which none of the evaluable functions will be evaluated at posting time, unless they are embedded in a `current` "function".

To illustrate this mechanism, we shall consider the following example. If you want to wait until 6 seconds have elapsed, in the old scheme (or if the `eval_on_post` option is `off`), you post the goal `(^ (>= (time) (+ (current (time)) 6)))`. Because you really want to distinguish between the call to `time` which will be done at "goal posting" time, and the call to `time` which is done at "goal satisfaction" time. But now (by default or if the `eval_on_post` option is `on`) you need to post: `(^ (>= (quote (time)) (+ (time) 6)))`.

This mechanism is put under a flag control, that the user can set on or off using the command `set eval post on|off` (see [OPRS Kernel Run Option Commands], §2.6, page 32). As said earlier, the new default behavior is to always evaluate the evaluable functions in a posted goal.

If this flag is set to `off`, the evaluation is done when required by the database, or when a `current` is used. One drawback of this approach (and this is the reason why the default mechanism is to evaluate every evaluable functions) is that it forces the user to use `current` whenever he wants to force an evaluation. For example, if you program factorial without using `current`, the real computation (the n - 1 multiplication) may be delayed until "printing" or until you affect the value to some variable (the `=` OP does an explicit `current` on its second argument). Another drawback, is that evaluable functions can then be evaluated more than once... If this evaluable function runs with considerable overhead, this my lead to very poor performance.

If it is set to `on`, which is its default value, any evaluable function would by default be evaluated at goal posting time, unless it is in the quote function. To keep the same semantic, the goal `(^ (>= (time) (+ (current (time)) 6)))`

would then become: `(^ (>= (quote (time)) (+ (time) 6)))`.

## 10.8    Critical Section

A critical section mechanism is provided in OPRS. Its use is very simple but should be reserved to very short sequences of goal execution, for which mutual exclusion is required (to allocate resources, for example).

During the critical section, the current intention and the current thread cannot be changed, and remains the same. Moreover, external events are not parsed (they are kept in the input buffer though), and meta level facts are not concluded. However, facts and goals posted by the current thread are taken into account.

A number of situations will break the critical section, i.e. they will force the kernel to exit the critical section state. This will happen if the thread in critical section joins (and it is not the last thread to join). Similarly, if you perform active maintenance in a critical section, the system will break it. You may also get warning if you are doing suspicious things such as spliting in a critical section...

Critical sections are not re-entrant, thus, it is forbidden to open a new critical section while you are already in one (there is little interest in doing so). Keep in mind that while the kernel is in a critical section, external events are not parsed, and the reaction time of the system is therefore increased.

To start and end a critical section, use the `start-critical-section` and the `end-critical-section` action. Corresponding OPs are provided in *'new-default.opf'*.

## 10.9    Universal Quantification of Variables

Universal quantification of variables can be obtained by using the `all`, `n-all` and `n-all-list` functions (see [Lisp Evaluable Functions], §6.1.7, page 102).

These functions return lists of binding which universally quantify some variables.

## 10.10    User Pointers

It is possible for a user to define its own data structure to manipulate and to be manipulated by the kernel. For example, one can define a robot path as a particular C data structure, which can then be manipulated by its pointer. Appropriate actions, evaluable functions and evaluable predicates can then respectively be used to create this objects, to access slots or test properties of this object. One issue arising of this facility is to determine which objects should be manipulated as user-defined objects (which are thus opaque to the kernel) and which objects should be represented explicitly in the database. This trade off is

a readability/efficiency/accessibility issue. For example, the information hidden in user-defined objects cannot be easily used to trigger procedure execution.

## 10.11 Action Slicing

Long user-defined actions can be time sliced. By returning a special token `:wait`, they are not considered by the kernel as completed and will be called again to finish their duties. An action can be time sliced in as many parts as the user decides when it programs it. For example, if it writes an action which perform some long computation such as writing a collection of data in a file, it may decides to write one object at a time and to call it as many time as there are objects. Keep in mind that the reactivity of the application depends of the longest action/evaluable functions of your application. Therefore, to increase the reactivity of your kernel, you may have to time slice the execution of actions.

See `action_first_call` and `action_number` (see [Intention Manipulation Functions], §G.1.8, page 344).

# Part IV

# OPRS-Server

# Overview of the OPRS-Server

The OPRS-Server is an important tool in the OPRS development environment. The philosophy behind OPRS prevents the user from interacting synchronously with a OPRS Kernel. However, one should be able to issue commands to a running kernel without disturbing the OPRS Kernel main loop. This is where the OPRS-Server comes in. The OPRS-Server is a program which enables the user to interact with a OPRS Kernel as much as possible. Moreover, the server allows the user to create new kernels, kill them, and so on... As most OPRS Development Environment programs, the OPRS-Server will start the Message Passer if necessary.

The OPRS Server enables the user to communicate directly with OPRS Kernels. Indeed, OPRS Kernels must be able to execute their procedures without being disturbed synchronously by the user. That is the reason why, the user can communicate with OPRS Kernels through the OPRS Server. This does not apply to X-OPRS Kernel with which the user can communicate using the X11/Motif interface.

# Chapter 11

# How to Use the OPRS-Server

## 11.1 Arguments of the OPRS-Server

Usage:

```
oprs-server [-X] [-l upper|lower|none] [-i server-port-number]
            [-m message-passer-hostname]
            [-j message-passer-port-number]
            [-l upper|lower|none ]
```

All the arguments are optional.

**-X** to specify that all the OPRS Kernels created by using the `make` command in the OPRS-Server, will be X-OPRS Kernels.

**-i** to specify the port on which the OPRS-Server is listening to connections from OPRS Kernels.

**-m** to specify the hostname on which the Message Passer is running (or will be started). If the OPRS-Server cannot connect (even after starting it) to this hostname on the specified port, then the program exits with an error message.

**-l upper|lower|none** can be used to print and parse all the symbol and id in upper case, lower case or in no particular case.All the kernels created by this OPRS-Server will inherit this property.

**-j** to specify the port on which the Message Passer is listening.

## 11.2   OPRS-Server Environment Variables

There are a number of environment variables which can be used to customize the OPRS-Server or to define default arguments. Arguments passed using the command line have precedence on those acquired from environment variables.

OPRS_MP_PORT is used to specify the port on which the Message Passer will listen to connection. It is used by the OPRS Kernel, the X-OPRS Kernel, the OPRS-Server and the Message Passer. It is equivalent to the `-j` command line argument. Example:

```
setenv OPRS_MP_PORT 3456
```

OPRS_MP_HOST is used to specify the host on which the Message Passer will listen to connection. It is used by the OPRS Kernel, the X-OPRS Kernel, the Message Passer and the OPRS-Server. It is equivalent to the `-m` command line argument. Example:

```
setenv OPRS_MP_HOST machine.site.domain
```

OPRS_SERVER_PORT is used to specify the port on which the OPRS-Server will listen to connection. It is used by the OPRS Kernel, the X-OPRS Kernel and the OPRS-Server. It is equivalent to the `-i` command line argument. Example:

```
setenv OPRS_SERVER_PORT 3457
```

OPRS_ID_CASE is used to specify if the program should upper case, lower case or should not change the case of the parsed Id. This is equivalent to the `-l` option. The possible values are `lower`, `upper` or `none`: Example:

```
setenv OPRS_ID_CASE none
```

## 11.3   Commands of the OPRS-Server

Here is the list of the commands the OPRS-Server recognizes.

### 11.3.1   OPRS-Server Commands to Handle OPRS Kernel

- `make name`. To create a OPRS Kernel named `name` (in a separate Unix process).

- `make-x name`. To create a X-OPRS Kernel named `name` (in a separate Unix process).

- `kill name`. To kill the OPRS Kernel named `name`. This can only work if the OPRS Kernel has been started with the `make` command.

- `accept`. To accept the connection of a new OPRS Kernel (or a X-OPRS Kernel). This is used whenever a kernel has been started from a Unix shell (from a remote host for example) and is waiting for the OPRS-Server to accept its connection. There is a common mistake when one uses the `accept` command. If you have started a OPRS Kernel and the connection to the OPRS-Server does not work after an `accept`, you probably started this server with a port number for it or the Message Passer which is not the default one. Remember to specify these numbers in the command line of the OPRS Kernel you start.

- `connect name`. To connect the standard input to the OPRS agent named `name`. This puts the OPRS agent in *command* mode.

- `disconnect`. To instruct the connected OPRS to leave the `stdin` and give it back to the OPRS-Server. The OPRS client will return in *run* mode. In fact, this command is not a OPRS-Server command but a OPRS Kernel command.

- `reset kernel name`. To send a `reset kernel` command to the OPRS Kernel named `name` (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).

- `reset parser name`. To reset the parser of the OPRS Kernel named `name`.

## 11.3.2   OPRS-Server Communication Commands

- `send name message`. To send the `message` to the OPRS named `name`. Example: `send foo (bar boo 3)`.

- `add name goal|fact`. To send a `goal` or a `fact` to the OPRS named `name`. This is how you can post new facts or new goals in a OPRS client. Example for a fact: `add foo (bar boo 3)` or for a goal: `add foo (! (print-factorial 3))`.

- `transmit name string`. To send a `string` command to a OPRS client named `name`. This is how you can send commands to a OPRS client without connecting to it. If the command you want to send contains double quotes (such as `include "data/foo.inc"`), then you must backslash double quote like in: `transmit foo "include`
  `"data/foo.inc`
  `" "`.

- `transmit_all string`. To send a `string` command to all the OPRS Kernel clients named connected to this OPRS-Server. The syntax of the string is similar to the one used in the `transmit` command.

- `broadcast message`.  To send the `message` to all the OPRS connected to this OPRS-Server's Message Passer.  Example: `broadcast (bar boo 3)`.

### 11.3.3   OPRS-Server Miscellaneous Commands

- `q|quit|exit|EOF`. To quit the OPRS-Server. This also kills all the OPRS clients started by this OPRS-Server.

- `include file_name`.  To execute all the commands in `file_name`.  The recommended extension for these files is '.inc'.  Include file can contain other include directives.  Only two commands are forbidden in include file: `connect` and `disconnect` (see [Include File Format], §2.14, page 40).

- `show version|copyright`. To print the version or the copyright notice.

- `help|h|?`. To print some on-line help.

# Part V

# Message Passer

# Overview of the Message Passer

The Message Passer is the program which allows an application to communicate with OPRS Kernels and X-OPRS Kernels. The Message Passer has one and only one function: it passes messages between various programs which have been registered. Most of the time, it is started by the OPRS-Server, a X-OPRS Kernel or a OPRS Kernel, so you do not have to call it directly. However, you can if you want, start it on any host, with any port on which to listen. In this case, you have to call it with the port number you want it to listen. See [Argument of the Message Passer], §12.1, page 157, for details. There is a companion program to the Message Passer: `kill-mp` which can be used to kill the Message Passer. Moreover, if the Message Passer has no client registered for more than five hours, it will exit (to make sure your machine is not loaded by unused Message Passer processes).

The main characteristic of the Message Passer are:

- Communication using TCP/IP, the most popular communication media and protocol on the Unix operating system.

- Communication on heterogeneous network. One can have a X-OPRS Kernel on a Sparc Station communicating with a OPRS Kernel on a DEC Station, while the application run on a VAX.

- Various protocols available between the simulators, the applications and the Message Passer.

- Easy to use from your application, using registration and communication functions provided in a library .

The Message Passer is identified by a host name and a port number. In other words, one can potentially run as many Message Passer as desired on a network of heterogeneous machines. Usually, one will run one Message Passer for one OPRS application. This Message Passer serves as the central server for messages and information passing between the different programs involved in a specific application.

By default the Message Passer listens on the port 3300, however, there are means to get it to listen onto another port. This new port can currently be specified when you start the OPRS-Server using the appropriate argument (see [Arguments of the OPRS-Server], §11.1, page 149, for details), or when you start the Message Passer on its own.

# Chapter 12

# How to Use the Message Passer

The various programs of the OPRS development environment connect to the Message Passer without any intervention of the user. However, if you write your own module and want it to communicate with the Message Passer, then you need to register it and follow a particular protocol.

The registration mainly consists in connecting to a public TCP/IP Internet socket, and in sending the desired protocol and its name. There are two possible protocols for connection to the message passer. After the registration, the client (which in most cases is a OPRS Kernel) is supposed to check from time to time its `mp-socket` by selecting it (in C) or listening to it (in Lisp). If something is present, a message is available and should be read promptly. Similarly, when a message has to be sent to another OPRS kernel, or to an external module which has registered to the Message Passer, it is just a matter of writing two strings on the socket `mp-socket`: one for the name of the recipient and one for the message itself. These strings are sent in a particular format, and the user should not attempt to send them directly but instead use the library functions provided for this very purpose.

Note that each module needs to connect to the Message Passer with a unique name.

The Message Passer can be killed or shutdown using the `kill-mp` program.

## 12.1   Argument of the Message Passer

Although it is seldom started from the Unix shell, one can start the Message Passer on its own. In this case, you have to specify the proper arguments in the OPRS-Server command line (see [Arguments of the OPRS-Server], §11.1, page 149) and the OPRS command line (see [Arguments to the oprs Command], §1.2, page 22) .

Usage:

```
mp-oprs [-j message-passer-port-number]
        [-l filename] [-x] [-v]
```

All the arguments are optional.

-j to specify the port on which the Message Passer is listening.

-l filename can be used to log in the file *'filename'* all the messages passed
     by the Message Passer.

-v to specify a verbose mode for the Message Passer.  In verbose mode, all
     messages passed by the Message Passer are traced.

-x to specify that any new registration made to the Message Passer with an
     already registered name lead to the deconnection of the old client.  The
     default behavior is to refuse the connection to client with name already
     used.

## 12.2   Message Passer Environment Variables

There is one environment variable which can be used to customize the Message
Passer. However, the argument passed using the command line has precedence
on the one acquired from the environment variables.

OPRS_MP_PORT is used to specify the port on which the Message Passer will listen
     to connection.  It is used by the OPRS Kernel, the X-OPRS Kernel, the
     OPRS-Server and the Message Passer. Example:

```
setenv OPRS_MP_PORT 3456
```

## 12.3   Argument of the Message Passer Killer

One can use the kill-mp program to kill a Message Passer.  Note that this
program can kill any Message Passer you started on any host. So this command
should be used with extreme caution. The Message Passer will check that the
Message Passer Killer program has been started by the same user than the one
who started it (the Message Passer). Moreover, the super user (the user root)
can kill any Message Passer.
    Usage:

```
kill-mp [-m message-passer-hostname]
        [-j message-passer-port-number]
```

All the arguments are optional.

-m to specify the hostname on which the Message Passer is running (and will
     be killed).

`-j` to specify the port on which the Message Passer is listening (and will connected to to be killed).

## 12.4 Message Passer Killer Environment Variables

There are two environment variable which can be used to customize the Message Passer Killer. However, the argument passed using the command line have precedence on the one acquired from the environment variables.

OPRS_MP_PORT is used to specify the port on which the Message Passer Killer connect to kill the Message Passer. It is equivalent to the `-j` command line argument. Example:

    setenv OPRS_MP_PORT 3456

OPRS_MP_HOST is used to specify the host on which the Message Passer Killer will connect to kill the Message Passer. It is equivalent to the `-m` command line argument. Example:

    setenv OPRS_MP_HOST machine.site.domain

## 12.5 How to Connect to the Message Passer from OPRS-Server and OPRS Kernel

There is nothing particular to do for the OPRS-Server or a OPRS Kernel to use the Message Passer. For all these programs, the registration to the Message Passer is mandatory and automatic. Upon starting, they register to the Message Passer, and they regularly check if something is present for them on the appropriate socket.

## 12.6 How to Connect to the Message Passer from an External Module

Any external module or program can register to the Message Passer. It is even the only way for an arbitrary program to communicate with OPRS Kernels. By default, the Message Passer always reports on `stderr` the registration of a new client. There are two different protocols: the `MESSAGES_PT` protocol and the `STRINGS_PT` protocol. When you establish a connection, you need to specify the protocol connection. Note that if you use the registration function without protocol specification, the rule to define the protocol is the following. The `MESSAGES_PT` protocol is the default one, to use the `STRINGS_PT` protocol, you need to suffix the name of your module with a `/`. So if your module is called `foo`,

you should send the string `foo/` on the socket for registration. However, the module will be known under the `foo` name. This mechanism remains for upward compatibility. In any case, you should try to use the registration function with the protocol specification.

`mp_port`, `external_register_to_the_mp_host`, `external_register_to_the_mp` are not supported anymore.

The registration functions and variables are provided in the *'libmp.a'* library for this purpose. Their prototype is defined in *'mp-pub.h'*.

The following protocol types are currently supported:

```
typedef enum {MESSAGES_PT, STRINGS_PT} Protocol_Type;
```

### mp_socket                                          MP Library Variable

> `int mp_socket`  is the Message Passer socket on which a program can send messages to and receive messages from the Message Passer.

### mp_name                                            MP Library Variable

> `char * mp_name`  is the name of the client as set by the Message Passer. It is usually the name you gave as argument.

### external_register_to_the_mp_host_prot  MP Library Function

> `int external_register_to_the_mp_host_prot (char *name, char *host_name, int port, Protocol_Type prot)`  registers the calling program to the Message Passer under the name `name`, on the host specified in `host_name`, on the port `port`, with the protocol specified in `port`.

### external_register_to_the_mp_prot                  MP Library Function

> `int external_register_to_the_mp_prot (char *name, int port, Protocol_Type prot)`  is similar to the previous function except that the host on which the Message Passer is expected to run is the same as the one on which the program runs.

These two functions set and return the value of `mp_socket`. If this value is -1, then the connection attempt failed.

## 12.7   Messages Format

The format of data between sender and recipient using the Message Passer depends on the established protocol. It is up to each module to parse and interpret them as they arrive. From the sender to the Message Passer, the format is always two strings (one for the recipient and one for the message). However, from the Message Passer to a module, it depends. In `MESSAGES_PT` mode, the string

"receive <name-sender> <message>" is sent as is. In STRINGS_PT mode, the name and the message are sent as two separate strings, each string being sent as an int (representing its size) and a byte stream for the string itself.

For example, in MESSAGES_PT mode, if the OPRS Kernel FOO wants to send the message (position valve1 closed) to the OPRS Kernel BAR, then it calls the function send_message. The Message Passer sees two strings coming on FOO's socket: "BAR" the recipient, and "(POSITION VALVE1 CLOSED)" the printed representation of the message. After processing it, the Message Passer sends one string to BAR: "receive FOO (POSITION VALVE1 CLOSED)" which is in fact interpreted as a command by the BAR kernel that the message (POSITION VALVE1 CLOSED) has been received from the kernel FOO.

In STRINGS_PT mode, the name and the message are sent as two separate strings, each string being sent as an int for its size and a byte stream for the string.

Here are the functions provided to read, write, and send message. Their prototype is defined in *'mp-pub.h'*.

## read_string_from_socket                MP Library Function

PString read_string_from_socket (int socket, int *size) is the function, in the *'libmp.a'* library, which can be used to read a string from the mp-socket in STRINGS_PT protocol. This function needs to be called twice, one time for the name of the sender, and one time for the text of the message. The size parameter is a return parameter to tell you how big the string is. Do not forget to free the result with the free function.

## send_message_string                MP Library Function

void send_message_string (PString message, PString rec) is the function, in the *'libmp.a'* library, which can also be used to send a message message to the recipient rec. This function send_message_string is preferred to the former function write_string_to_socket.

## multicast_message_string                MP Library Function

void multicast_message_string (PString message, unsigned int nb_recs, PString *recs) is the function, in the *'libmp.a'* library, which can be used to send a message message to a list of nb_recs recipient which names are in the array of PString recs.

## broadcast_message_string                MP Library Function

void broadcast_message_string (PString message) is the function, in the *'libmp.a'* library, which is used to send a message message to all the connected agents, except the sender.

## 12.8   Example of C Code to Connect to the Message Passer

This code comes from the Truck Loading Demo (See [Truck Loading Example], §23.1, page 273), and can be found in the file *'demo/truck-demo/src/oprs-interface.c'*.

```c
void demo_init_arg(int argc, char **argv)
{
    int c, getoptflg = 0;
    int mpname_flg = 0, mpnumber_flg = 0, demoname_flg = 0;

    struct hostent *check_hostname;
    int mp_port;
    extern int optind;
    extern char *optarg;
    int maxlength = MAX_HOST_NAME * sizeof(char);

    while ((c = getopt(argc, argv, "m:j:n:h")) != EOF) {
        switch (c)
        {
        case 'm':
            mpname_flg++;
            mp_host_name = optarg;
            break;
        case 'j':
            mpnumber_flg++;
            if (!sscanf (optarg, "%d", &mp_port ))
                getoptflg++;
            break;
        case 'n':
            demoname_flg++;
            demo_name = optarg;
            break;
        case 'h':
        default:
            getoptflg++;

        }
    }
    if (getoptflg) {
        fprintf(stderr, DEMO_ARG_MESSAGE );
        exit(1);
    }

    if (mpname_flg){
```

```
            if ((check_hostname = gethostbyname (mp_host_name)) == NULL){
                fprintf(stderr, "Invalid mp host name \n");
                exit (1);
            }
    } else {
            mp_host_name = (char *)malloc (maxlength);
            if (gethostname(mp_host_name, MAX_HOST_NAME) != 0) {
                fprintf(stderr, "Error in gethostname \n");
                exit(1);
            }
    }
    if (!mpnumber_flg)
            mp_port = MP_PORT;

    if ( !demoname_flg ){
            demo_name = default_demo_name;
            connect_name = default_connect_name;
    } else {
            int i, length = strlen (demo_name);

            if (demo_name[length-1] != '/') { /* The name doesn't end with a
                                                 '\', add it to get the
                                                 right message format . */

                connect_name = (char *)malloc (length +2) ;
                for (i = 0; i< length ; i++){
                    if (islower(demo_name [i]))
                        connect_name[i] = toupper (demo_name[i]);
                    else
                        connect_name[i] = demo_name[i];
                }
                connect_name[length] = '/';
                connect_name[length + 1] = '\0';

            }
    }
}

void send_message_to_oprs (char *message)
{
    char trace_message[BUF_SIZE];

    if (!demo->connected) {
            demo_error ("send_message: You are not connected ");
            return;
    }
```

```
    send_message_string(message, OPRS_NAME);

    sprintf (trace_message, "Send: %s\n", message);
    oprs_message(trace_message);

}

void get_oprs_message (XtPointer client_data, int *fid, XtInputId id)
{
    char trace_message[BUF_SIZE];
    int length;
    char *sender;
    char *message;

    sender = read_string_from_socket(*fid, &length);
    message = read_string_from_socket(*fid, &length);

    if ( decode_message (message, sender) != 0){
        sprintf (trace_message, "Received %s from %s \n", message, sender);
        demo_error (trace_message);
    }

    free(sender);
    free(message);
}

void connect_to_mp ()
{
    if ((mp_socket = external_register_to_the_mp_host_pfrot(connect_name,
mp_host_name, mp_port, STRINGS_PT)) == -1) {
        demo->connected = FALSE;
        demo_warning ("Unable to register to the Message Passer");
    } else {
        demo_message ("You are connected to the Message Passer\n");
        demo->connected = TRUE;
        XtAppAddInput (app_context,
                        mp_socket,
                        XtInputReadMask,
                        get_oprs_message,                     /* the read function */
                        NULL);
    }
}
```

## 12.9   Example of Lisp Code to Connect to the Message Passer

Here are the foreign function definitions to register a lisp program to the Message Passer, and to send messages to the Message Passer:

```
(in-pacopge "OPRS" :use '("LISP" "LUCID-COMMON-LISP"))

;;; Loading

(defun i-oprs-load ()
  (load-foreign-libraries
   nil
   (list "/usr/local/oprs/lib/libmp.a"
         "-lm"
         "-lc")))

;;; functions  base

(def-foreign-function
  (i-external-register-to-the-mp
       (:name "_external_register_to_the_mp")
       (:language :c)
       (:return-type :signed-32bit))
  (name :string))

(def-foreign-function
  (i-external-register-to-the-mp-host
       (:name "_external_register_to_the_mp_host")
       (:language :c)
       (:return-type :signed-32bit))
  (name :string)
  (host :string))

(def-foreign-function
  (i-send-message-string
       (:name "_send_message_string")
       (:language :c)
       (:return-type :null))
  (msg :string)
  (target :string))

(i-oprs-load)
```

Here are some lisp functions to be used to communicate with the Message Passer. Note that as OPRS uses a lisp like syntax, we can use read to read the

message coming from the OPRS Kernels.

```lisp
(in-pacopge "OPRS" :use '("LISP"))

(defvar *mp-stream* nil)

(defun valid-sd (sd)
  (and (integerp sd) (> sd 0)))

(defun already-registered ()
  (streamp *mp-stream*))

(defun oprs-register-to-mp (name &optional (machine (machine-instance)))
  (check-type name string)
  (check-type machine string)
  (when (already-registered)
    (warn "Connection to the Message Passer already established.")
    (return-from oprs-register-to-mp nil))
  (let ((sd (i-external-register-to-the-mp-host name machine)))
    (cond ((valid-sd sd)
           (setf *mp-stream* (make-lisp-stream :input-handle sd
                                               :output-handle sd
                                               :auto-force t))
           t)
          (t (warn "Failed to connect to the Message Passer."))
          nil)))

(defun oprs-close ()
  (break-if-not-registered)
  (close *mp-stream*)
  (setf *mp-stream* nil))

;;;---
;;; writing
;;;---

(defun oprs-write (&key target msg)
  (check-type target string)
  (break-if-not-registered)
  (i-send-message-string (format nil "~S" msg) target))

;;;---
;;; reading
;;;---

(defun oprs-listen ()
```

```lisp
  (break-if-not-registered)
  (loop
   (if (and (listen *mp-stream*)
            (member (peek-char nil *mp-stream*)
                    '(#\space
                      #\newline
                      #\linefeed
                      #\return
                      #\page
                      #\backspace
                      #\tab
                      #\)
                      )))
       (read-char *mp-stream*)
       (return)))
  (listen *mp-stream*))

(defstruct oprs-msg
  sender
  contents
)

(defun oprs-read ()
  (break-if-not-registered)
  (if (oprs-listen)
      (progn
        (read *mp-stream*) ; This is to remove the "receive" keyword
        (make-oprs-msg :sender (read *mp-stream*)
                       :contents (read *mp-stream*)))
      :NO-MSG))

(defun break-if-not-registered ()
  (unless (already-registered)
    (break "No connection with the Message Passer.")))
```

Here are the entries you may want to export out of the `OPRS` pacopge.

```lisp
(in-pacopge "OPRS" :use '("LISP" "LUCID-COMMON-LISP"))

(defvar *oprs-exports*)

(eval-when (load eval compile)
  (setf *oprs-exports*
        '(
          oprs-register-to-mp
```

```
        oprs-close

        oprs-write
        oprs-listen
        oprs-read

        oprs-msg-sender
        oprs-msg-contents
        )))

(export *oprs-exports*)
```

## 12.10   Errors Reported by the Message Passer

A certain number of errors can be reported by the Message Passer. In general, the Message Passer is very verbose and notifies the user of any unexpected event or situation. Whenever it is possible, the reported message indicates the host and the port number on which this Message Passer is running. The most common problems are:

`"Disconnecting the client:  %s from the message passer."` This usually happens when a client died and the communicating socket has been closed. When the Message Passer realizes this, it prints this message.

`"Registering the client:  %s with protocol:  %s."` is printed upon successful connection to the Message Passer.

`"logging output in file '%s'."` is printed whenever the Message Passer log its output to a file.

`"nobody registered for more than %d seconds, mp-oprs (%d):  exit."` is printed when the Message Passer exits when there is no connection and no new connection have been made in a certain amount of time.

`"already has a client named:  %s.  Denying registration."` This problem is reported if a new client tries to register with a name already used by somebody else.

`"EOF in get_and_send_message (recipient) from %s."` This error is reported when the Message Passer gets an End of File while reading the name of the recipient on a socket. Most often it appears when a client dies.

`"EOF in get_and_send_message, (message) from %s."` This error is reported when the Message Passer gets an End of File while reading the message part of a pair recipient-message on a socket.

`"A message has been sent to %s, but no such agent exists."` This error is reported if a message is sent to a an unknown client.

`"unknown message type in get_and_send_message from %s."` This error occurs whenever a message with an unknown type is received by the message passer.

`"Disconnecting the client:  %s from the message passer."` This message is printed when a client is disconnected by the Message Passer.

`"kill request, checking identity."` A kill request has been received by the Message Passer which checks if it has been sent by an authorized client.

`"denying kill-mp, you are not the user who started this message passer"` is printed when a the client which sent the kill request is not authorized.

`"shutting down the message passer socket."` is printed when the Message Passer exit after a kill request has been sent.

`"A message could not be delivered to %s."` This error is reported if a message cannot be properly delivered to its recipient for some unknown reason.

# Part VI

# X-OPRS Kernel

# Overview of the X-OPRS Kernel

The X-OPRS Kernel is an important program of the OPRS Development Environment pacopge: it has all the characteristics of the OPRS Kernel, but better, it can execute OPs and procedures (i.e. graphically trace them) under the X11 Window/Motif interface.

X-OPRS Kernel is the X11/Motif version of the OPRS Kernel. It is functionally identical to the OPRS Kernel, but allow the user to graphically follow the execution of the procedures, as well as the evolution of the tasks graph. Using the graphical user interface, one have access to the underlying OPRS Kernel to perform the following operations:

- To graphically follow the execution of selected procedures

- To graphically follow the evolution of the current tasks of the system,

- To select the procedure to be traced,

- To consult the database,

- To stop and resume the execution of the kernel, or to execute step by step some selected procedures,

- To establish new goals or to conclude new facts in the database

- To select the various operations of the kernel to be traced,

- To select the run-time options of the kernel,

- To load new procedures or new databases,

- To access an on-line help and documentation.

You can call this program directly from an Unix shell or you can call it directly from the OPRS-Server when you execute the `make-x` command, or with the `make` command when the OPRS-Server has been started with the `-X` argument (see [Arguments of the OPRS-Server], §11.1, page 149).

X-OPRS Kernel contains the OPRS Kernel. In fact, the kernel part of X-OPRS is the OPRS Kernel. The OPRS main loop is running interleaved with the Xt Application Main Loop. Note that the OPRS Kernel has better performance than its X11 counterpart (because of the absence of the Xt Application Main Loop). Any performance study should be made with the OPRS Kernel alone (except, of course, if the goal is to evaluate the performance of the X interface).

One interesting feature of the X-OPRS Kernel is that you can interact with the kernel more easily than with the OPRS Kernel. With the OPRS Kernel you can only interact with the help of the OPRS-Server or by connecting to the OPRS itself. With the X-OPRS Kernel however, the Xt Main Loop, which runs interleaved with the OPRS main loop, allows you to do "asynchronous" operations with the running kernel. Therefore, it is not necessary (nor is it permitted) to `connect` (see [Commands of the OPRS-Server], §11.3, page 150) to a X-OPRS Kernel.

There is another program, `oprs-cat`, which runs when you run an X-OPRS Kernel. Its goal is to echo on its `stdout` whatever is sent on its `stdin`... This is used internally by X-OPRS to display text trace in the Text Trace Pane.

# Chapter 13

# How to Use the X-OPRS Kernel

The X-OPRS Kernel is used exactly as its tty version, the OPRS Kernel (see [How to Use the OPRS Kernel], §1, page 21), and the argument available are the same (see [Arguments to the oprs Command], §1.2, page 22). Note however that a number of Xt arguments are available and can be used by the user (see [Xt Command Line Arguments], §L.1, page 387). Moreover, there are a number of specific arguments to the X-OPRS Kernel which we now introduce.

**-log filename** can be used to log in the file *'filename'* all the outputs produced by the kernel and appearing in the text window.

**-pwt** can be used to print the X-OPRS Kernel widget tree (see [X-OPRS Motif Widgets Hierarchy], §L.2.3, page 388) . This can be useful if you do not have the documentation at hand and still want to now the name or type of a particular widget.

**-peo** can be used to print the temporal operator in English instead of the single letter. It will print `achieve` instead of `!`, and `wait` instead of `^` and so on. This is equivalent to the `-p` argument of the OPRS Kernel.

Various commands can be used to control the execution of the X-OPRS Kernel. These commands are grouped into two sets: the "Menubar" and the "Control and Status Panel".

## 13.1   X-OPRS Kernel Environment Variables

There are a number of environment variables which can be used to customize the X-OPRS Kernel or to define default arguments. Arguments passed using the command line have precedence on those acquired from environment variables.

OPRS_DATA_PATH is used to specify a data path, i.e. a colon separated list of
directories where the kernel will look for data files (*'.inc'*, *'.opf'* and *'.db'*).
It is used by the OPRS Kernel and the X-OPRS Kernel. It is equivalent
to the -d command line argument. Example:

```
setenv OPRS_DATA_PATH ./data:/usr/local/oprs/data:${HOME}/data
```

OPRS_DOC_DIR is used to specify the location of the online OPRS Development
Environment documentation. It is used by the X-OPRS Kernel and the
OP Editor. Example:

```
setenv OPRS_DOC_DIR /usr/local/oprs/doc
```

OPRS_MP_PORT is used to specify the port on which the Message Passer will listen
to connection. It is used by the OPRS Kernel, the X-OPRS Kernel, the
OPRS-Server and the Message Passer. It is equivalent to the -j command
line argument. Example:

```
setenv OPRS_MP_PORT 3456
```

OPRS_MP_HOST is used to specify the host on which the Message Passer will listen
to connection. It is used by the OPRS Kernel, the X-OPRS Kernel and
the OPRS-Server. It is equivalent to the -m command line argument.
Example:

```
setenv OPRS_MP_HOST machine.site.domain
```

OPRS_SERVER_PORT is used to specify the port on which the OPRS-Server will
listen to connection. It is used by the OPRS Kernel, the X-OPRS Kernel
and the OPRS-Server. It is equivalent to the -i command line argument.
Example:

```
setenv OPRS_SERVER_PORT 3457
```

OPRS_SERVER_HOST is used to specify the host on which the OPRS-Server will
listen to connection. It is used by the OPRS Kernel and the X-OPRS
Kernel. It is equivalent to the -s command line argument. Example:

```
setenv OPRS_SERVER_HOST machine.site.domain
```

OPRS_ID_CASE is used to specify if the program should upper case, lower case
or should not change the case of the parsed Id. This is equivalent to the
-l option. The possible values are lower, upper or none: Example:

```
setenv OPRS_ID_CASE none
```
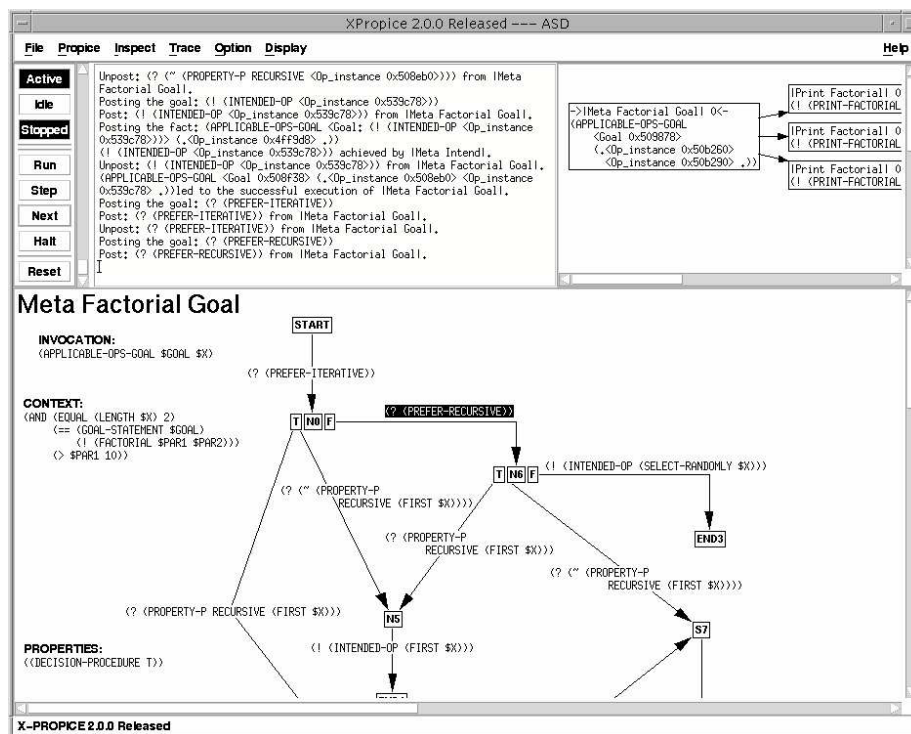
Figure 13.1: X-OPRS Window

## 13.2    Windows and Panes of the X-OPRS Kernel

As shown on Figure 13.1, panes and menus are present in the default X-OPRS configuration. The different menus and control panels are explained in the [Menubar], §13.3, page 180, and in the [Control and Status Panel], §13.4, page 201.

### 13.2.1    Text Pane

This pane is located on the upper left part of the X-OPRS frame. It is used for any Text output. The outputs can be generated either by selected trace, or by the OPRS Kernel itself. This pane does not accept any input. The text contained in this pane is scrollable and previous output can be re displayed using the scroll bars. However, for memory allocation reasons, the size of the buffer corresponding to this text is limited but can be changed by the user (see [Change Size Text Pane], §13.3.6, page 201).

### 13.2.2    Graphic OP Pane

This pane is used to display the tracing of executing OPs when the appropriate trace flags are on. It can also be used to display a particular OP (see [Display Menu], §13.3.6, page 200). The user can use the scroll bars to change the view port of this pane. He can also click on and drag the window himself to move the view port around.

In this pane, if you left click on a goal, or an invocation part, or an effect part, the X-OPRS Kernel will propose a list of relevant OP (among the OP currently loaded in the X-OPRS Kernel). This is very convenient to jump from one OP to the OPs which may achieve a similar goal or which may achieve a subgoal of this procedure.

### 13.2.3    Graphic Intention Pane

This pane is used to display the tracing of Intentions and Tasks when the appropriate trace flags are on. The user can use the scroll bars to change the view port of this pane. He can also click on the window himself to move the view port around.

If you right click on an intention in the intention graph a window displaying the traces (OP text traces, OP success and failures, etc) specific to this intention will appear (as shown on Figure 13.2).

If you middle click on an intention a Text Window Dialog Box is poped up and contains the status of the selected intention (see Figure 13.3). This dialog box is not modal and has an Update button which can be used to update the window to display the current intention graph.

Figure 13.2: Specific Intention Trace Window
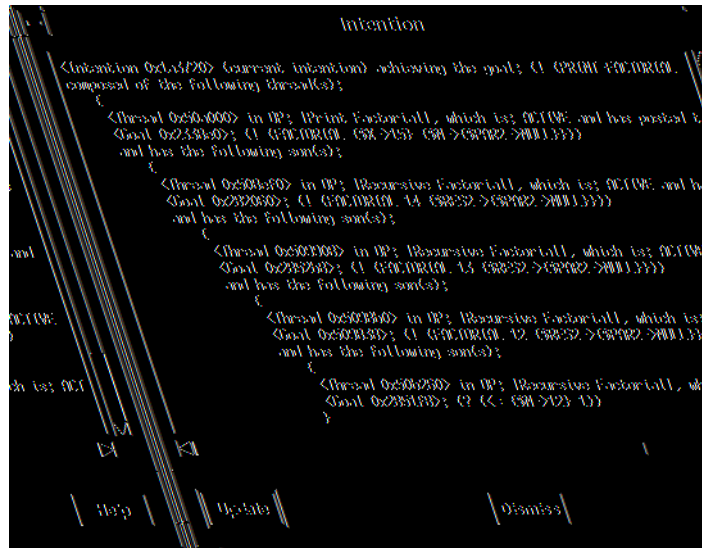


Figure 13.3: Show Intention Dialog Box



Figure 13.4: X-OPRS Menu Bar
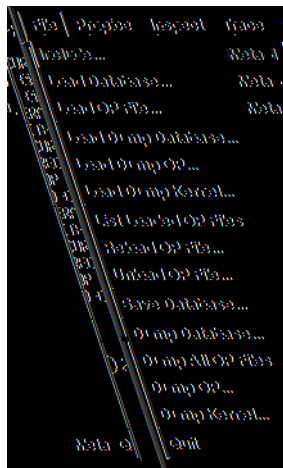
Figure 13.5: X-OPRS File Menu

## 13.3   Menubar

The Menu Bar (Figure 13.4) contains different buttons from which cascade menus pop when they are selected with the mouse.

### 13.3.1   File Menu

The File menu (Figure 13.5) contains all the commands dealing with files. Most of these commands have their counterpart in the OPRS Kernel (see [OPRS Kernel Commands], §2, page 27).

#### Include

This command is used to load an include file. The default (and recommended) extension for these files is '.inc' (see [Include File Format], §2.14, page 40). When this command is selected, a file selection dialog box appears to allow the user to select the include file to load. This command is equivalent to the `include` command of the OPRS Kernel (see [OPRS Kernel Loading Commands], §2.4, page 30).

#### Load Database

This command is used to load a database file. The default (and recommended) extension for these files is '.db' (see [Database File Format], §5.1, page 79). When this command is selected, a file selection dialog box appears to allow the user to select the database file to load. This command is equivalent to the `load db` command of the OPRS Kernel (see [OPRS Kernel Loading Commands], §2.4,

page 30). The kernel database can be emptied with the "Empty Fact Database" command (see [OPRS Menu], §13.3.2, page 184).

### X-OPRS Load OP File

This command is used to load a OP File, in OPF format (see [OPF Format], §17.1, page 241). The default (and recommended) extension for these files is *'.opf'* . When this command is selected, a file selection dialog box appears to enable the user to select the OP file to load. This command is equivalent to the `load opf` command of the OPRS Kernel (see [OPRS Kernel Loading Commands], §2.4, page 30). The kernel OP Library can be emptied with the "Empty OP Database" command (see [OPRS Menu], §13.3.2, page 184).
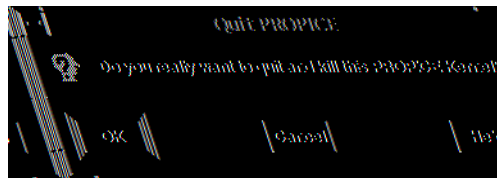
### Load Dump Database

This command is used to load dump database file. The default (and recommended) extension for these files is *'.ddb'*. When this command is selected, a file selection dialog box appears to allow the user to select the database file to load. This command is equivalent to the `load dump db` command of the OPRS Kernel (see [OPRS Kernel Dumping/Loading Commands], §2.11, page 36).

### Load Dump OP

This command is used to load a dump OP File. The default (and recommended) extension for these files is *'.dopf'* . When this command is selected, a file selection dialog box appears to enable the user to select the OP file to load. This command is equivalent to the `load dump op` command of the OPRS Kernel (see [OPRS Kernel Dumping/Loading Commands], §2.11, page 36).

### Load Dump Kernel

This command is used to load a dump kernel File. The default (and recommended) extension for these files is *'.dkrn'* . When this command is selected, a file selection dialog box appears to enable the user to select the kernel file to load. This command is equivalent to the `load dump kernel` command of the OPRS Kernel (see [OPRS Kernel Dumping/Loading Commands], §2.11, page 36).

### List Loaded OP Files

This command is used to list all the OP File loaded in the X-OPRS Kernel. This command is equivalent to the `list_opfs` command of the OPRS Kernel (see [OPRS Kernel OP Library Commands], §2.3, page 29).

### Reload OP File

This command is used to reload a OP File, in OPF format (see [OPF Format], §17.1, page 241). The default (and recommended) extension for these files is
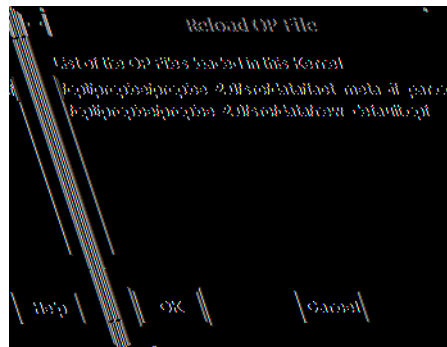
Figure 13.6: Reload OP File Dialog List
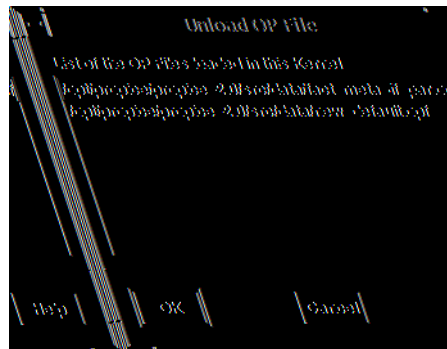


Figure 13.7: Unload OP File Dialog List

*'.opf'* . When this command is selected, a selection dialog with the list of load OP file appears to enable the user to select the OP file to reload (Figure 13.6). This command is equivalent to the `reload opf` command of the OPRS Kernel (see [OPRS Kernel Loading Commands], §2.4, page 30).

### X-OPRS Unload OP File

This command is used to unload a OP File. When this command is selected, a dialog box displaying all the OP Files currently loaded in the kernel appears to enable the user to select the OP file to unload (see Figure 13.7). This command is equivalent to the `unload_opf` command of the OPRS Kernel (see [OPRS Kernel OP Library Commands], §2.3, page 29). The kernel OP Library can be emptied with the "Empty OP Library" command (see [OPRS Menu], §13.3.2, page 184).

Figure 13.8: Quit Dialog Box

**Save Database**

This command is used to save the current state of the database in a file. The default (and recommended) extension for this file is *'.db'* . When this command is selected, a file selection dialog box appears to allow the user to select the database file to use. This command is equivalent to the `save db` command of the OPRS Kernel (see [OPRS Kernel Database Commands], §2.2, page 28).

**Dump Database**

This command is used to dump the database in a file. The default (and recommended) extension for these files is *'.ddb'*. When this command is selected, a file selection dialog box appears to allow the user to select the file in which to dump the database. This command is equivalent to the `dump db` command of the OPRS Kernel (see [OPRS Kernel Dumping/Loading Commands], §2.11, page 36).

**Dump OP**

This command is used to a dump the op library in a file. The default (and recommended) extension for these files is *'.dopf'* . When this command is selected, a file selection dialog box appears to enable the user to select the file in which to dump the OP library. This command is equivalent to the `dump op` command of the OPRS Kernel (see [OPRS Kernel Dumping/Loading Commands], §2.11, page 36).

**Dump Kernel**

This command is used to a dump the kernel in a file. The default (and recommended) extension for these files is *'.dkrn'* . When this command is selected, a file selection dialog box appears to enable the user to select the file in which to dump the kernel. This command is equivalent to the `dump kernel` command of the OPRS Kernel (see [OPRS Kernel Dumping/Loading Commands], §2.11, page 36).
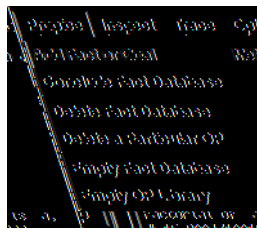
Figure 13.9: X-OPRS Oprs Menu
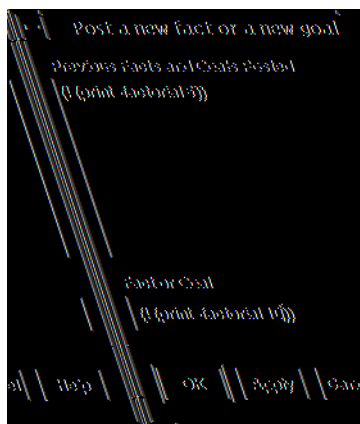


Figure 13.10: Add Fact or Goal Prompt Dialog

### X-OPRS Quit

This command is used when you want to quit the X-OPRS Kernel. You are asked to confirm that you want to quit (see Figure 13.8). This command is equivalent to the `q|quit|exit|EOF` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39). Quitting the X-OPRS Kernel will execute the `end_kernel_user_hook` function (see [Advanced Features], §10, page 135).

## 13.3.2   OPRS Menu

This menu (Figure 13.9) contains a number of commands to deal with the running X-OPRS Kernel.

### Add Fact or Goal

This command is used to add a fact or a goal to the running X-OPRS Kernel. When this command is selected, a prompt dialog box appears to enable the
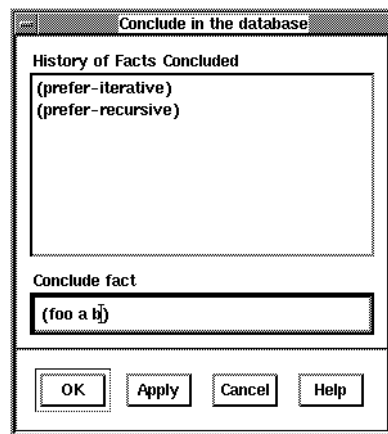
Figure 13.11: Conclude Database Dialog Box

user to enter the fact or the goal to add (see Figure 13.10). This command is equivalent to the `add goal|fact` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39). If you want to enter more than one fact or goal in the same OPRS Kernel loop, you can halt the kernel (with the "Halt" button), call this menu a number of time and then restart it.

Moreover, this command can be used to conclude a fact in the database. However, it is not purely equivalent to the `conclude expression` of the OPRS Kernel: the `add fact` leads to some OP executions if they are applicable, while the `conclude` command only concludes the fact in the database.

**Conclude Fact Database**

This command is used to conclude a expression in the running X-OPRS Kernel. When this command is selected, a Prompt dialog box appears to enable the user to enter the expression to conclude (see Figure 13.11). This command is equivalent to the `conclude expression` command of the OPRS Kernel (see [OPRS Kernel Database Commands], §2.2, page 28).

**Delete Fact Database**

This command is used to delete a gexpression in the running X-OPRS Kernel. When this command is selected, a Prompt dialog box appears to enable the user to enter the gexpression to delete (see Figure 13.12). This command is equivalent to the `delete gexpression` command of the OPRS Kernel (see [OPRS Kernel Database Commands], §2.2, page 28).
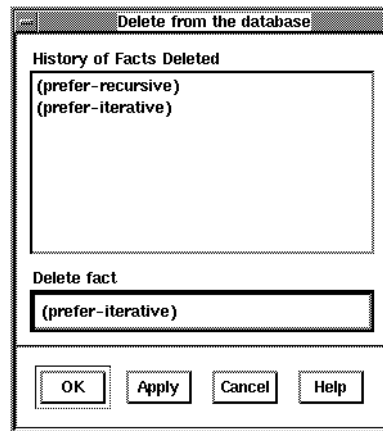
Figure 13.12: Delete Database Dialog Box
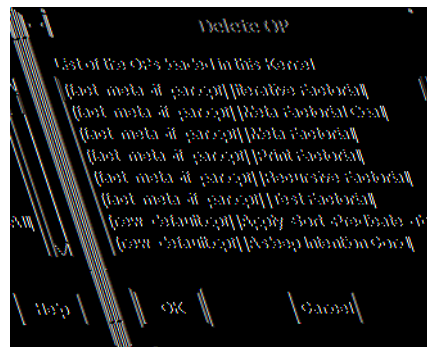


Figure 13.13: Delete OP Dialog Box
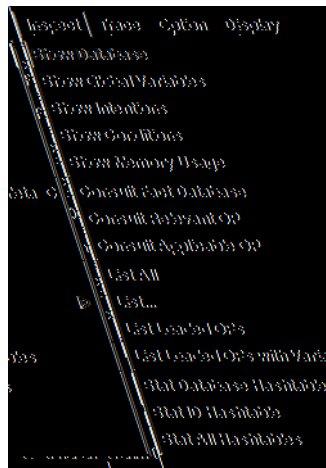
Figure 13.14: X-OPRS Inspect Menu

**Delete a OP**

This command can be used to delete a particular OP from the kernel. When selected, a dialog box (see Figure 13.13) appears on the screen with the list of the OP currently loaded. You can then select the one you want to delete. It is not recommended to do this while the X-OPRS Kernel is running, particularly if it is executing the OP you are deleting.

**Empty Fact Database**

This command is used to clear the entire fact database. It is equivalent to the `empty fact db` command of the OPRS Kernel (see [OPRS Kernel Loading Commands], §2.4, page 30).

**Empty OP Library**

This command is used to clear the OP Library. It is equivalent to the `empty_op` command of the OPRS Kernel (see [OPRS Kernel Loading Commands], §2.4, page 30). This command should not be used while the kernel is executing some OPs.

### 13.3.3 Inspect Menu

This menu (Figure 13.14) contains a number of commands to inspect the running X-OPRS Kernel.
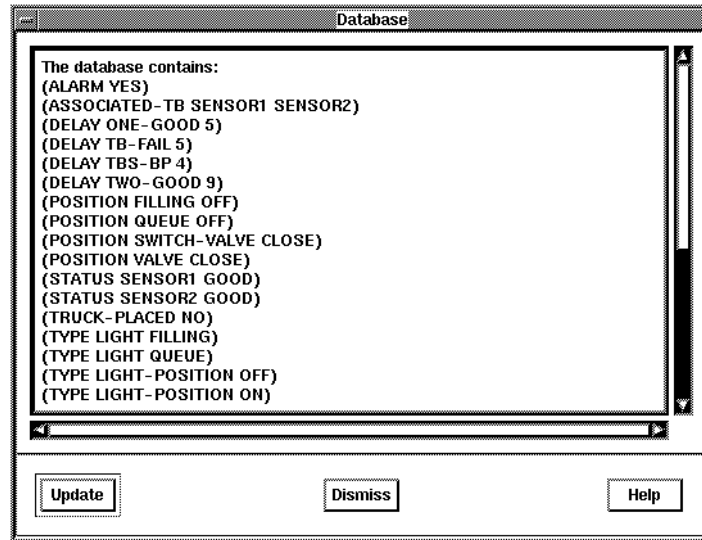
Figure 13.15: Show Database Dialog Box

### Show Database

This command is used to show the database content. It is equivalent to the `show db` command of the OPRS Kernel (see [OPRS Kernel Database Commands], §2.2, page 28). When this command is selected a Text Window Dialog Box is poped up and contains the current database content (sorted alphanumericaly) (see Figure 13.15). This dialog box is not modal and has an Update button which can be used to update the window to display the current content of the database.
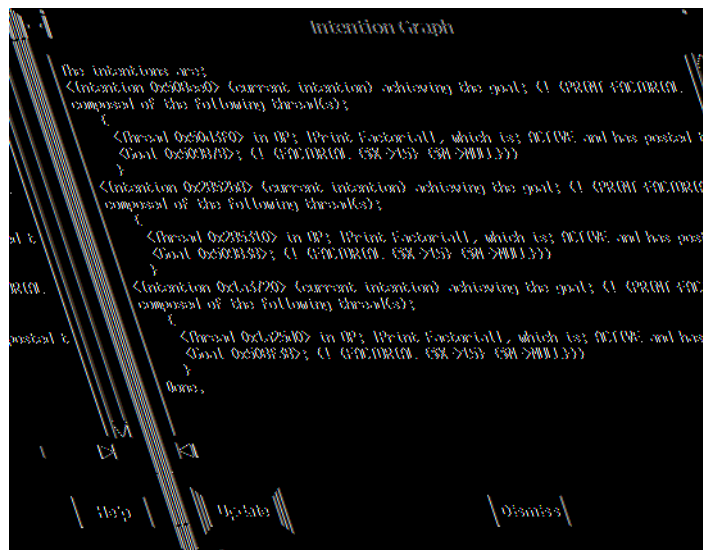
### Show Global Variables

This command is used to show the global variables. It is equivalent to the `show variable` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).

### Show Intentions

Display all the intentions, if any, with lot of information on the status of their thread, etc. This command is equivalent to the `show intention` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39). When this command is selected a Text Window Dialog Box is poped up and contains the current intention graph (see Figure 13.16). This dialog box is not modal and has an Update button which can be used to update the window to display the current intention graph..

Figure 13.16: Show Intentions Dialog Box

**Show Conditions**

Display all the conditions, if any, with lot of information on the status of their thread, etc. This command is equivalent to the `show condition` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39). When this command is selected a Text Window Dialog Box is poped up and contains the current condition. graph (see Figure 13.17). This dialog box is not modal and has an Update button which can be used to update the window to display the current intention graph..

**Show Memory Usage**

This command is used to show the memory usage. It is equivalent to the `show memory` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).

**Consult Fact Database**

This command is used to consult a gexpression in the running X-OPRS Kernel. When this command is selected, a Prompt dialog box appears to enable the user to enter the gexpression to consult (see Figure 13.18). The result appears in the text window. This command is equivalent to the `consult gexpression` command of the OPRS Kernel (see [OPRS Kernel Database Commands], §2.2, page 28).
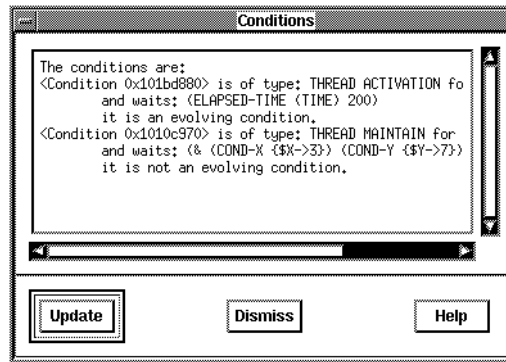
Figure 13.17: Show Conditions Dialog Box
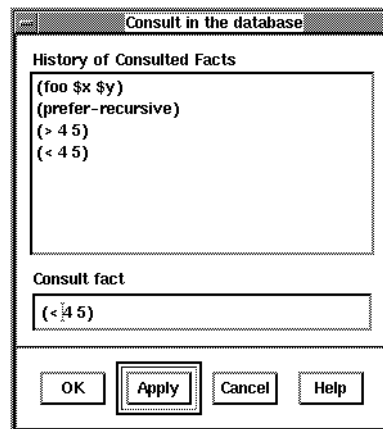


Figure 13.18: Consult Database Dialog Box

Figure 13.19: X-OPRS Inspect List Menu

**Consult Relevant OP**

This command is used to find out which OPs are relevant for a fact or a goal. When this command is selected, a prompt dialog box appears to enable the user to enter the fact or the goal to consult. The result appears in the text window. This command is equivalent to the `consult relevant op fact|goal` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).

**Consult Applicable OP**

This command is used to find out which OPs are applicable for a fact or a goal. When this command is selected, a prompt dialog box appears to enable the user to enter the fact or the goal to consult. The result appears in the text window. This command is equivalent to the `consult applicable op fact|goal` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).

**List All**

List all sort of information about this kernel (actions, evaluable functions, predicates, etc). It is equivalent to the `list all` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Submenu**

This command brings a submenu with the following commands:

**List Predicate**   This command will list the Predicate. It is equivalent to the `list predicate` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Evaluable Predicate**   This command will list the Evaluable Predicate. It is equivalent to the `list evaluable predicate` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Closed World Predicate**   This command will list the Closed World Predicate. It is equivalent to the `list cwp` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Functional Fact Predicate**   This command will list the Functional Fact Predicate. It is equivalent to the `list ff` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Basic Event Predicate**   This command will list the Basic Event Predicate. It is equivalent to the `list be` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List OP Predicate**   This command will list the OP Predicate. It is equivalent to the `list op predicate` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Function**   This command will list the Function. It is equivalent to the `list function` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Evaluable Function**   This command will list the Evaluable Function. It is equivalent to the `list evaluable function` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

**List Action**   This command will list the Actions. It is equivalent to the `list Action` command of the OPRS Kernel (see [OPRS Kernel Listing Commands], §2.10, page 35).

### List Loaded OPs

This command is used to list all the OPs loaded in the X-OPRS Kernel. This command is equivalent to the `list_ops` command of the OPRS Kernel (see [OPRS Kernel OP Library Commands], §2.3, page 29).

### Stat Database Hashtables

This command is used to find out the state of the database hashtables. This command is equivalent to the `stat db` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).
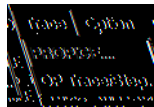
Figure 13.20: X-OPRS Trace Menu

**Stat Symbol Hashtable**

This command is used to find out the state of the Symbol hashtable. This command is equivalent to the `stat id` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).

**Stat All Hashtables**

This command is used to find out the state of all the hashtables of the kernel. This command is equivalent to the `stat all` command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39).

### 13.3.4   Trace Menu

The trace menu (Figure 13.20) contains all the commands which allow the user to set various traces in the X-OPRS Kernel.

**OPRS Trace**

When this menu is selected, a menu options dialog box appears for the user to set or unset various trace options (see Figure 13.21). Most of these trace options have their pending flag in the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Goal Posting** Turn on or off information on the goal posted in the kernel. This command is equivalent to the `trace goal on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Fact Posting** Turn on or off information on facts posted in the kernel. This command is equivalent to the `trace fact on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

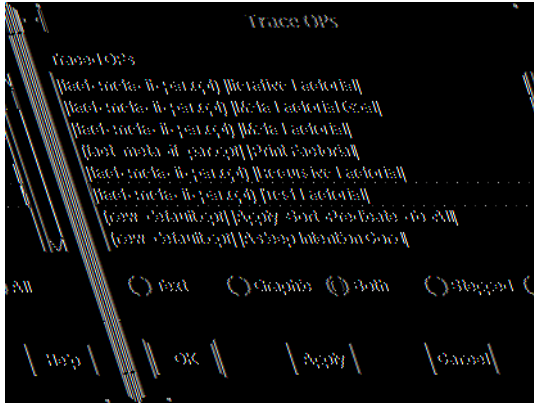**Conclude from Parser** Turn on or off information on expression concluded in the kernel. This command is equivalent to the `trace conclude on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Message Reception** Turn on or off information on messages received by the kernel. This command is equivalent to the `trace receive on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).
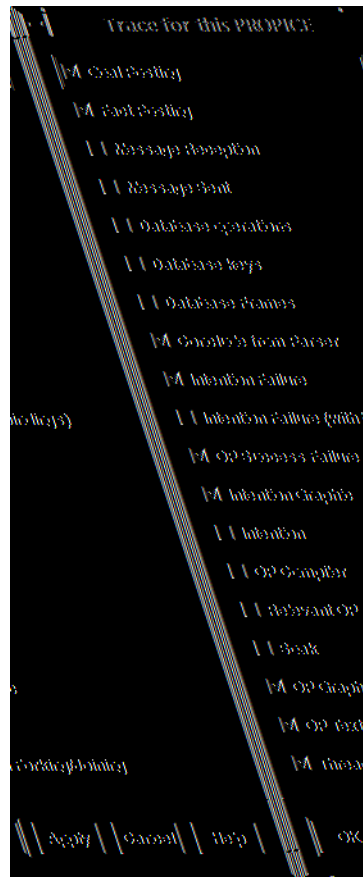
Figure 13.21: X-OPRS Trace Dialog Box

**Message Sent** Turn on or off information on messages sent by the kernel. This command is equivalent to the `trace send on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Database** Turn on or off trace on database operations. This command is equivalent to the `trace db on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Database Frame** Turn on or off the printing of the frames while printing the result of a consultation. This command is equivalent to the `trace db frame on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**OP Success Failure** Turn on or off information on the success or failure of OPs. This command is equivalent to the `trace suc_fail on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Intention Failure** Turn on or off information on the failure of Intentions. This command is equivalent to the `trace intention failure on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Intention Graphic** Turn on or off graphic trace on the intention operation in the kernel.

**Intention** Turn on or off information on the intention operation in the kernel. This command is equivalent to the `trace intend on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**OP Compiler** Turn on or off information on the compilation of OPs. This command is equivalent to the `trace load op on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31). (very verbose)

**Relevant OP** Turn on or off information on relevant OPs. This command is equivalent to the `trace relevant op on|off` command of the OPRS kernel (see [OPRS Kernel Trace Commands], §2.5, page 31). (quite verbose)

**Soak** Turn on or off information on the set of applicable OPs. This command is equivalent to the `trace applicable op on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**OP Graphic** Turn on or off graphic traces on executing OPs. This command is equivalent to the `trace graphic on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

Figure 13.22: OP Graphic List Dialog

**OP Text**  Turn on or off text traces on executing OPs.  This command is equivalent to the `trace text on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**Thread Forking/Joining**  Turn on or off information on thread creation and merging.  This command is equivalent to the `trace thread on|off` command of the OPRS Kernel (see [OPRS Kernel Trace Commands], §2.5, page 31).

**OP Trace/Step**

This menu pops up list dialog boxes of all the OPs loaded in the X-OPRS kernel (see Figure 13.22).  You can then select or unselect the OPs for which you want tracing (Text trace or Graphic trace) and stepping (with the Next command) enabled.  The radio buttons are used to select the status displayed, and the status which will be applied to the selected OPs if you select OK or Apply. The Both buttons select both graphic and text trace. The all button select the two trace (graphic and text) and enable the OP steping.

As for the trace, the corresponding `OP Graphic` and/or `Text Graphic` option of the OPRS Trace option menu has to be on for the trace to be displayed.

You can also use the `trace graphic op`, `trace graphic opf` and `trace step opf` command describe in [OPRS Kernel OP Library Commands], §2.3, page 29.

### 13.3.5  Option Menu

The option menu (Figure 13.23) contains all the commands which allow the user to set various options in the X-OPRS Kernel.

Figure 13.23: X-OPRS Option Menu
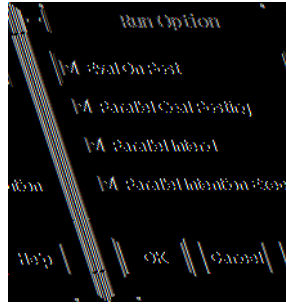


Figure 13.24: X-OPRS Run Option Dialog Box

**OPRS Run Option**

When this menu is selected, a run menu options dialog box appears for the user to set or unset various options (see Figure 13.24). Most of these options have their pending flag in the OPRS Kernel (see [OPRS Kernel Run Option Commands], §2.6, page 32).

**Eval On Post** Turn on or off the current-quote mechanism (see [Current and Quote], §10.7, page 141). This command is equivalent to the `set eval_on_post on|off` command of the OPRS Kernel (see [OPRS Kernel Run Option Commands], §2.6, page 32).

**Parallel Goal Posting** Turn on or off the parallel posting of goals. When this option is on, one goal for each thread active in the current intention will be posted (see [New Traces and New Options], §8.2, page 129). This command is equivalent to the `set parallel post on|off` command of the OPRS Kernel (see [OPRS Kernel Run Option Commands], §2.6, page 32).

**Parallel Intend** Turn on or off the parallel intending of OP instance. When this option is `ON`, all the OP Instances found in the `PREVIOUS_SOAK` (see [OPRS Kernel Main Loop], §7.4, page 113) are intended (see [New Traces and New Options], §8.2, page 129). This command is equivalent to the `set parallel intend on|off` command of the OPRS Kernel (see [OPRS Kernel Run Option Commands], §2.6, page 32).
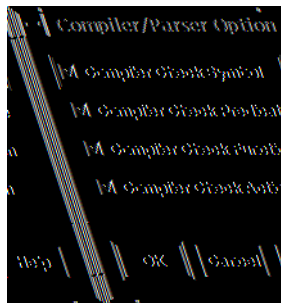
Figure 13.25: X-OPRS Compiler/Parser Option Dialog Box

**Parallel Intention Execution** Turn on or off the parallel execution of all the intention root of the intention graph. See [New Traces and New Options], §8.2, page 129. This command is equivalent to the `set parallel intention on|off` command of the OPRS Kernel (see [OPRS Kernel Run Option Commands], §2.6, page 32).

**Time Stamping** Turn on or off the time stamping in the kernel of various date (fact creation, goal creation, etc). This command is equivalent to the `set time_stamping on|off` command of the OPRS Kernel (see [OPRS Kernel Run Option Commands], §2.6, page 32).

**OPRS Compiler/Parser Option**

When this menu is selected, a menu options dialog box appears for the user to set or unset various options (see Figure 13.25). Most of these options have their pending flag in the OPRS Kernel (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

**Compiler Check Action** Turn on or off action checking in the OP Compiler. See [Action Checking], §4.4.1, page 76 for more information. This command is equivalent to the `set action on|off` command of the OPRS Kernel (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

**Compiler Check Function** Turn on or off function checking in the OP Compiler. See [Function Checking], §4.4.3, page 77 for more information. This command is equivalent to the `set function on|off` command of the OPRS Kernel (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

**Compiler Check Predicate** Turn on or off predicate checking in the OP Compiler. See [Predicate Checking], §4.4.2, page 76 for more information. This command is equivalent to the `set predicate on|off` com-
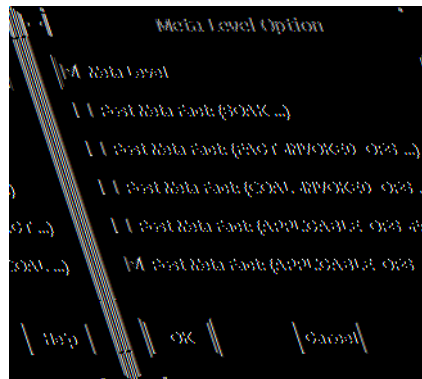
Figure 13.26: X-OPRS Meta Level Option Dialog Box

mand of the OPRS Kernel (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

**Compiler Check Symbol** Turn on or off symbol checking in the OP Compiler. See [Symbol Checking], §4.4.4, page 77 for more information. This command is equivalent to the `set symbol on|off` command of the OPRS Kernel (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

**OPRS Meta Level Option**

When this menu is selected, a menu options dialog box appears for the user to set or unset various options (see Figure 13.26). Most of these options have their pending flag in the OPRS Kernel (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

**Meta Level** Turn on or off the metalevel mechanism, which greatly increases the performance of the system. This command is equivalent to the `set meta on|off` command of the OPRS Kernel (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

**Post Meta Fact: (SOAK ...)** Turn on or off the posting of the (`SOAK`) meta fact. This command is equivalent to the `set soak on|off` command of the OPRS Kernel (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

**Post Meta Fact: (FACT-INVOKED-OPS ...)** Turn on or off the posting of the (`FACT-INVOKED-OPS`) meta fact. This command is equivalent to the `set meta fact on|off` command of the OPRS Kernel (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).
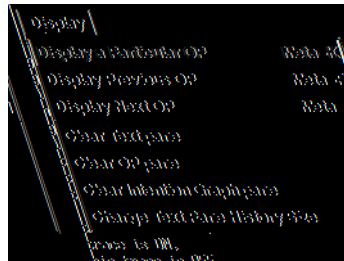
Figure 13.27: X-OPRS Display Menu

**Post Meta Fact: (GOAL-INVOKED-OPS ...)**  Turn on or off the posting of the `(GOAL-INVOKED-OPS)` meta fact.  This command is equivalent to the `set meta goal on|off` command of the OPRS Kernel (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

**Post Meta Fact: (APPLICABLE-OPS-FACT ...)**  Turn on or off the posting of the `(APPLICABLE-OPS-FACT)` meta fact.  This command is equivalent to the `set meta fact op on|off` command of the OPRS Kernel (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

**Post Meta Fact: (APPLICABLE-OPS-GOAL ...)**  Turn on or off the posting of the `(APPLICABLE-OPS-GOAL)` meta fact.  This command is equivalent to the `set meta goal op on|off` command of the OPRS Kernel (see [OPRS Kernel Meta Level Option Commands], §2.7, page 33).

### 13.3.6   Display Menu

This menu (Figure 13.27) contains the following items.

**Display a Particular OP**

This command can be used to display a particular OP on the screen.

**Display Previous OP**

This command can be used to display the previous OP on the screen.

**Display Next OP**

This command can be used to display the next OP on the screen.

**Clear Text Pane**

Clear the text pane.
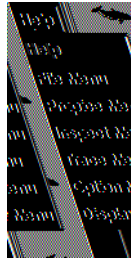
Figure 13.28: X-OPRS Help Menu

**Clear OP Pane**

Clear the OP pane.

**Clear Intention Graph Pane**

Clear the intention graph pane.

**Change Size Text Pane**

This command can be used to change the size of the scrollable area of the Text pane.

### 13.3.7   X-OPRS Help Menu

The help menu (Figure 13.28) contains a number of items. The first one, when selected, pops up the documentation. The other items have the name of the menu in the menu bar. They points to the respective documentation section.

## 13.4   Control and Status Panel

The Control and Status Panel (Figure 13.29) is divided in 2 parts, the Status Panel and the Control Button Menu. The first one indicates the current status of the kernel, and the second one allows the user to control its execution.

### 13.4.1   Status Panel

At all time, the status panel tells the user what is the status of the X-OPRS Kernel. The status is updated from time to time by the X-OPRS kernel (every tenth of a second or so). These buttons and the state they represent are not exclusive, therefore more than one of these buttons can be ON at the same time.

**Idle** When this button is ON, it means that there was no new goal nor new fact in the current loop of the kernel. This button sometimes blink off
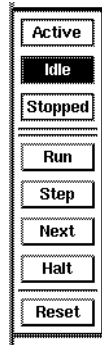
Figure 13.29: The Control and Status Panel

> while the kernel is running because indeed, you can have sometime, a loop without new goal or new fact.

**Stopped** When this button is ON, it means that the X-OPRS Kernel has been stopped by the user (using the step, next or halt command).

**Active** This button is ON when there is "something to do".

### 13.4.2   Control Button Menu

To some extend, the user can control the execution of the X-OPRS Kernel by using the button of the Control Panel. To activate a command is just a matter of clicking on the appropriate button. These commands are equivalent to the command presented in See [OPRS Kernel Status and Control Commands], §2.12, page 38.

**Halt** Click on this button to stop the execution of the X-OPRS kernel. While the kernel is halted, you can perform most command, such as consulting the fact or OP Library, adding new fact or new goal (in which case they will be taken into account whenever the kernel is restarted), displaying a particular OP, load new OPs, etc.

**Run** Click on this button to restart a stopped X-OPRS Kernel.

**Step** Click on this button to step through one loop of the X-OPRS kernel. Note that one loop execution does not always produce any noticeable or visible effect. . . .

**Next** Click on this button to run the kernel until the control hits an edge of a step trace OP. This is very useful when you graphic trace OPs. At each click, the execution goes from one traced edge to the next traced edge (whatever execution happens in between).

**Reset** Click on this button to reset the kernel. This command is equivalent
to the `reset kernel` command of the OPRS Kernel (see [OPRS Kernel
Status and Control Commands], §2.12, page 38).

# Part VII

# OP Compiler

# Overview of the OP Compiler

The OP Compiler is the program which compiles OP files (textual or graphical) in OPRS internal code. There are a number of good reasons to use such program:

- it is nuch faster to reload compiled OPs than non compiled one (as the kernel must compile them at loading time).

- the compiled code is portable, and can thus be reloaded on any compatible OPRS Kernel.

- one needs to use compiled OPs if one has a OPRS Application Environment. Indeed, OPRS Application Environment are runtime environment only and can only load OPs produced and compiled in a OPRS Development Environment.

The OP Compiler is in fact a part of the OPRS Kernel, and most of the code thus shared with the OPRS Kernel.

# Chapter 14

# How to Use the OP Compiler

The OP Compiler is only part of the OPRS Development Environment. Moreover, if one produce OPRS code for a OPRS Application Environment, one need to compile the OP before passing them to theOPRS Application Environment.

## 14.1  Argument of the OP Compiler

Usage:

```
opc [-v] [-t] [-X] [-e]* [-o output-dopf]*
    [-i command-file]* [-l input-opf]*
    [-d imputdopf]* [-p oprs-data-path]* op-file*
```

Most arguments are optional and can be used more than once. The order of the argument is very important as they get executed in the ordre they are given.

-v will produce verbose output.

-t will compile ops in text mode.

-X will compile ops in graphic mode, the default.

-e will empty the internal op database.

-o output-dopf will the compiled op present in the op db in the specified file.

-i command-file will include an include .inc or symbol .sym file.

-l input-opf will load a .opf file in the op db

-d imputdopf will load a .dopf file in the op db.

`-p oprs-data-path` will set the oprs-data-path (overiding any previous value).

`op-file` will empty the op db, load op-file and dump the compiled version.

## 14.2    OP Compiler Environment Variables

There is one environment variable which can be used to customize the OP Compiler. However, the argument passed using the command line has precedence on the one acquired from the environment variables.

`OPRS_DATA_PATH` is used to specify a data path, i.e. a colon separated list of directories where the kernel will look for data files (*'.inc'*, *'.opf'* and *'.db'*). It is used by the OPRS Kernel and the X-OPRS Kernel. It is equivalent to the `-p` command line argument. Note that the use of `-p` will overide any value previously set (with `OPRS_DATA_PATH` or wih a previous `-p`). Example:

```
setenv OPRS_DATA_PATH ./data:/usr/local/oprs/data:${HOME}/data
```

## 14.3    Using the OP Compiler

## 14.4    Errors Reported by the OP Compiler

A certain number of errors can be reported by the OP Compiler.

**Part VIII**

# OP Editor

# Overview of the OP Editor

The OP Editor is the graphical editing tool of the OPRS development environment. It is not included in the OPRS application environment, because this product does not provide any tool to create OPs and procedures. This tool uses X11/Motif as a graphical user interface.

The OP Editor is the graphical editor for procedures/OPs. It enables the user to create, edit and modify the procedures of a OPRS application.

- It runs under the X11 window system with the Motif widgets toolkit.

- Its embedded lexical and syntax checker ensure that the procedure the user write will be loaded in the OPRS Kernel or X-OPRS Kernel.

- It is upward compatible with the GRASPER II Graph format , and Sun Graph format.

- It allows the user to edit more than one procedure and more than one procedure file at the same time.

- It provides an on-line help and documentation using the your prefered HTML browser (Netscape).

- It has a friendly user interface.

The editor manipulates OPs and procedures which are stored in files. A file is called a OP File and has different formats (see [OP File Format], §17, page 241).

Each OP file can contain a certain number of OPs and procedures. When you edit a OP file, you can select the specific procedure you want to edit.

When selected, OPs and procedures are visible on the screen and can be modified as desired by the user.

The OP files produced by the OP Editor can then be loaded in a OPRS Kernel or an X-OPRS Kernel to be executed.

# Chapter 15

# How to Use the OP Editor

The OP Editor is invoked from the Unix shell with the command:
```
% op-editor
```

## 15.1   Arguments of the OP Editor

Usage:

```
op-editor [-D file-directory] [-F ACS-op-file] [-l upper|lower|none]
          [-m message-passer-hostname] [-j message-passer-port-number]
          [-c op-file*] [-pwt] [-peo] [-L en|fr] [op-file]*
```

The `op-editor` command also accept all the standard Xt arguments (see [Xt Command Line Arguments], §L.1, page 387).

All the arguments are optional.

-D to specify a directory from which you want the file to be looked for. If you say `-D data`, it tries to load subsequent files specified in the command line from the `data` sub directory.

-F to specify a file to load in OPF format. This is equivalent to the Load OP file command (see [Load OP File], §16.1.1, page 221). The '.opf' extension must be omitted as it will be added by the argument parser.

-m to specify the hostname on which the Message Passer runs or will be started. If the OP Editor cannot connect to this hostname on the specified port (even after trying to start the Message Passer), then the program exits with an error message. This option only applies if your license agreement require connection of the OP Editor to the Message Passer.

-j to specify the port on which the Message Passer is expecting a connection (or will be started if necessary). This option only applies if your license agreement require connection of the OP Editor to the Message Passer.
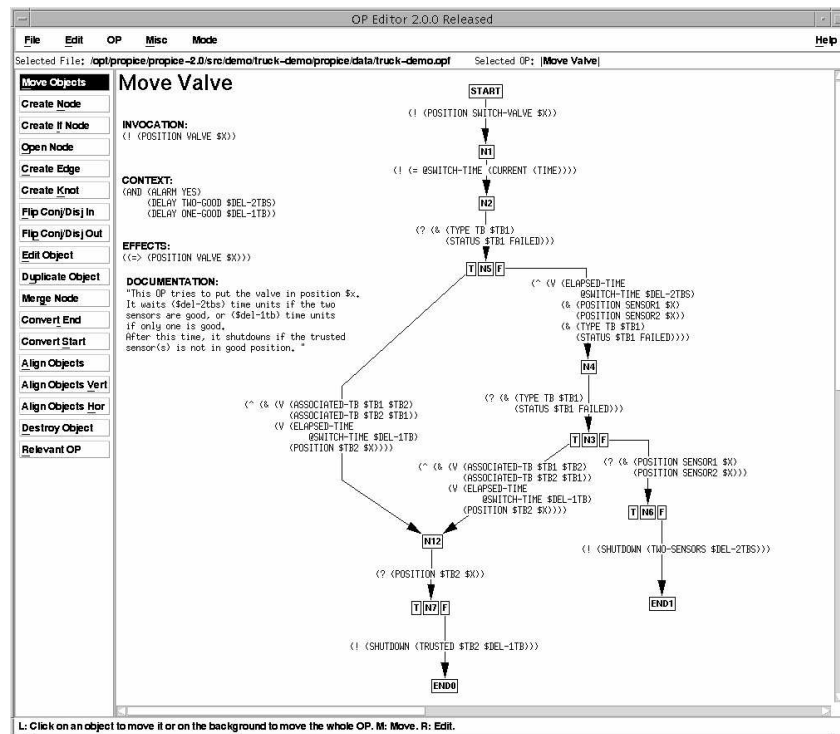
Figure 15.1: OP Editor Window

-c to convert OP file in the newest OP File format. A backup copy of the file is done with the *'.bak'* suffix.

-pwt is used to print the OP Editor widget tree (see [OP Editor Motif Widgets Hierarchy], §L.3.2, page 390) . This can be useful if you do not have the documentation at hand and still want to now the name or type of a particular widget.

-peo (which stands for Print English Operator) can be used to print the temporal operator in english instead of the single letter. It will print `achieve` instead of !, and `wait` instead of ^ and so on.

-l upper|lower|none can be used to print and parse all the symbol and id in upper case, lower case or in no particular case.

-L en|fr can be used to select the language of the interface (French or English). Note that by default your kernel is in english. Note also that for the applications with an X interface (i.e. X-OPRS Kernel and the OP Editor the choice of the *'app-defaults'* file will select the language (see [Xt/Motif Widgets Hierarchy and Resources], §L, page 385). In this case, selecting a different value with the option will lead to a warning and to a mix of language in the interface.

op-file to specify files to load in OPF format. This is equivalent to the Load OP file command (see [Load OP File], §16.1.1, page 221). The complete name must be given.

If you specify -G, -S or -F, the respective file extensions must be omitted.

## 15.2  OP Editor Environment Variables

There are a number of environment variables which can be used to customize the OP Editor or to define default arguments. Arguments passed using the command line have precedence on those acquired from environment variables.

OPRS_DOC_DIR is used to specify the location of the online OPRS Development Environment documentation. It is used by the X-OPRS Kernel and the OP Editor. Example:

```
setenv OPRS_DOC_DIR /usr/local/oprs/doc
```

OPRS_ID_CASE is used to specify if the program should upper case, lower case or should not change the case of the parsed Id. This is equivalent to the -l option. The possible values are `lower`, `upper` or `none`: Example:

```
setenv OPRS_ID_CASE none
```
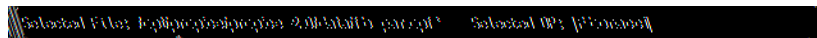
Figure 15.2: Selection Pane

## 15.3   Creating a OP

Creating a OP is a fairly easy task. In the OP Editor, select the Create OP
menu item (see [Create OP], §16.1.3, page 225) in the OP pull down menu. This
pops up a large dialog box (shown on figure 16.6) which can be filled with some
of the information needed to create a OP. Do not worry if you mistype or forget
something. The OP Editor checks the syntax, and you can change or edit the
different fields afterwards. Note that you can select the type of OP to create: a
Graph OP or an Action OP (this is not something you can change afterwards,
so you must choose now). According to your choice, you will get a OP with a
start node or (exclusive or) an action field. Click OK to create the OP. If any
syntax error is found, the OP Editor tells you in which field it was located.

As soon as the create dialog box disappears, you see your OP on the screen,
or more precisely its squeleton, as it only contains the fields you have filled up
and the start node (or the action field). By selecting the various operations of
the working menu (see [Working Menu Items], §16.2, page 235) and by following
the instructions in the Footer help window (see [Footer and Dialog Box Help],
§15.7, page 219), you can then create nodes or edges, edit them, and so on.

## 15.4   Editing an Existing OP

Editing an existing OP is also a fairly easy operation. In the OP Editor, select
the Select OP menu item (see [Select OP], §16.1.3, page 225) in the OP pull
down menu. This pops up a selection list dialog box containing all the OPs
present in this OP file. You can then select the one you want to see in the
editing area.

## 15.5   Scroll Bars

Scroll bars can be used at any time to change the view port of the drawing area.
The drawing area is virtually "as big as needed", so do not be afraid to draw
OPs as big as you want. When the OP Editor starts, it "creates" a window
big enough to contain the biggest OP you loaded. However, you can increase
the maximum size of the window by selecting the Change Drawing Size (see
[Change Drawing Size], §16.1.4, page 232) menu item in the Misc Menu.
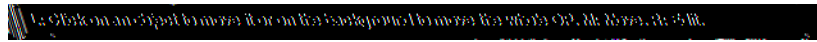
Figure 15.3: Footer Help Pane

## 15.6  Selection Pane

This pane (Figure 15.2) is located just below the menu bar and gives the following information to the user: the file name which is currently edited, the OP name which is currently selected, and a marker indicating whether the file has been modified or not.

## 15.7  Footer and Dialog Box Help

At any time, the user can get information and valuable help on what he is expected to do by looking at the footer window of the OP Editor (Figure 15.3). Moreover, most dialog boxes have "HELP" button pointing at the proper section of the on line manual (the present manual), which can be browsed with your HTML browser.

## 15.8  Pretty Printing

All text objects displayed by the OP Editor (but also the X-OPRS Kernel) are pretty printed. Pretty printing is usually appreciated when one has complex expressions to edit, it makes the editing process easier and faster. Moreover, the user can specify on which width the pretty printer should try to print the object, as well as, if it should (when possible) fill up the lines (see [Edit Object], §16.2.11, page 238). This information (the width and the line filling) is kept in the OPF format for each fields of each OP.

# Chapter 16

# OP Editor Commands

The commands of the OP-editor are grouped into two different sets of menu. The first set is a menu bar with cascade menus. The second set is a group of push buttons on the left side of the drawing pane. It is called the working menu, as it contains the commands which are most often used to edit OPs.

## 16.1   Menubar of the OP Editor

The Menu Bar (Figure 16.1) contains different buttons from which cascade menus pop when selected.

### 16.1.1   File Menu of the OP Editor

The file menu (Figure 16.2) contains all the operations dealing with files, i.e. operations to load OP files in different formats, to save or write them, etc.

**Load OP File**

This command is used to load a OP File, in OPF format (see [OPF Format], §17.1, page 241). The default (and recommended) extension for this file is *'.opf'* . When this command is selected, a file selection dialog box appears to enable the user to select the file to load (see Figure 16.3) .

If the load command is successful, the Select OP dialog box pops up, and the user is asked to select a OP to display.



Figure 16.1: OP Editor Menu Bar

221
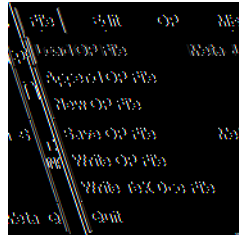
Figure 16.2: OP Editor File Menu
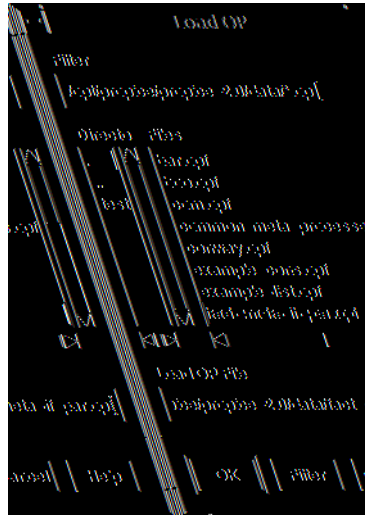


Figure 16.3: Load OP File Selection Box

**Append OP File**

This command is used to append a OP File, in OPF format (see [OPF Format], §17.1, page 241), to the currently selected OP file. When this command is selected, a file selection dialog box appears to enable the user to select the file to append.

If the append command is successful, the Select OP dialog box pops up, and the user is asked to select a OP to display.

**New OP File**

This command can be used to create a new OP file. By default, the name of this new file is *'Untitled'* with an increasing suffix number. The name of the file is chosen and defined the first time you save it with the save or write command.

**Load Grasper OP File**

This command is provided for upward compatibility with SRI Lisp OPRS.

This command is used to load a OP File in Grasper Graph format (see [GGRAPH Format], §17.2, page 241). The default (and recommended) extension for these files is *'.ggraph'* . When this command is selected, a File Selection dialog box appears to allow the user to select the file to load.

**Load Sun OP File**

This command is provided for upward compatibility with SRI Lisp OPRS.

This command is used to load a OP File in Sun Graph format (see [SGRAPH Format], §17.3, page 243). The default (and recommended) extension for this file is *'.sgraph'*. When this command is selected, a File Selection dialog box appears to allow the user to select the file to load. Note that nothing appears on the screen as the result of this command. You first need to save the file (in OPF format), and then reload the saved file.

**Save OP File**

The OP Editor can only save in OPF File format (see [OPF Format], §17.1, page 241). By default, the save command saves the currently selected OP File. If this OP file has no real filename (if it has been created with the New OP File command), then a Selection File dialog box pops up for you to choose the filename. The default and recommended extension is *'.opf'*.

If the currently selected OP file is not in the OPF File Format, an error dialog box pops up to advise you to use the write command instead.

**Write OP File**

The write command can be used to write the currently selected OP File in OPF format in the filename specified in the Selection File dialog box.
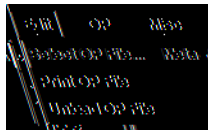
Figure 16.4: OP Editor Edit Menu

**Write TeX Doc File**

The write tex doc file command can be used to write a TEX/LATEX format documentation of the selected file in the filename specified in the Selection File dialog box. This documentation can the be included in TEX/LATEX document.

**Quit**

This command is used when you want to quit the OP Editor. If some files have not been saved, you are asked to confirm you want to quit without saving them.

## 16.1.2   Edit Menu of the OP Editor

This menu (Figure 16.4) contains the commands to deal with loaded OP Files.

**Select OP File**

This command allows you to change the currently selected file. A list of currently loaded or known OP Files is presented in a Selection Dialog Box. If the selection is successful, a Select OP dialog box pops up for you to choose the OP to display.

**Print OP File**

This command applies the "print" command (see [OP Editor Resources], §L.3.1, page 389, and see [Change Print Command], §16.1.4, page 232) to all the OPs of the current OP file. Beware, this can be a rather long process.

**Unload OP File**

This command allows the user to unload a OP file. As a result, the file will not appear in the Select OP file menu any longer.

## 16.1.3   OP Menu

This menu (Figure 16.5) contains all the commands which deal with OPs and, most often, with the Selected OP.
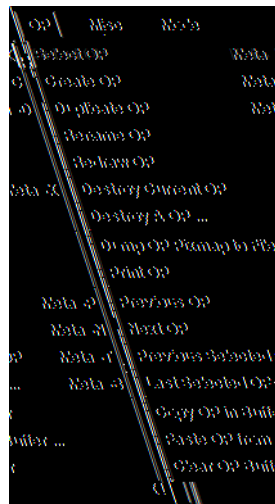
Figure 16.5: OP Editor Op Menu

**Select OP**

This command allows the user to change the Selected OP. Keep in mind that the name of the Selected OP is visible at any time in the information pane just below the Menu Bar.

**Create OP**

This command allows the user to create a new OP. A big dialog box pops up on the screen (see figure 16.6), and you are asked to fill up the various fields of the OP you want to create. Note that only two fields are really required: the name and the invocation part. The resulting OP is shown on Figure 16.7.

If you click on the Action toggle button, the Dialog Box will change as shown on Figure 16.8). An Action field is added, and you can enter the action part of the OP. The resulting OP is shown on Figure 16.9.

If you click on the Text toggle button, the Dialog Box will change as shown on Figure 16.10). A Body field is added, and you can enter the body part of the OP. The resulting OP is shown on Figure 16.11.

**Duplicate OP**

This command allows the user to duplicate the currently Selected OP. A prompt dialog box pops up for you to give the name of the duplicate OP.
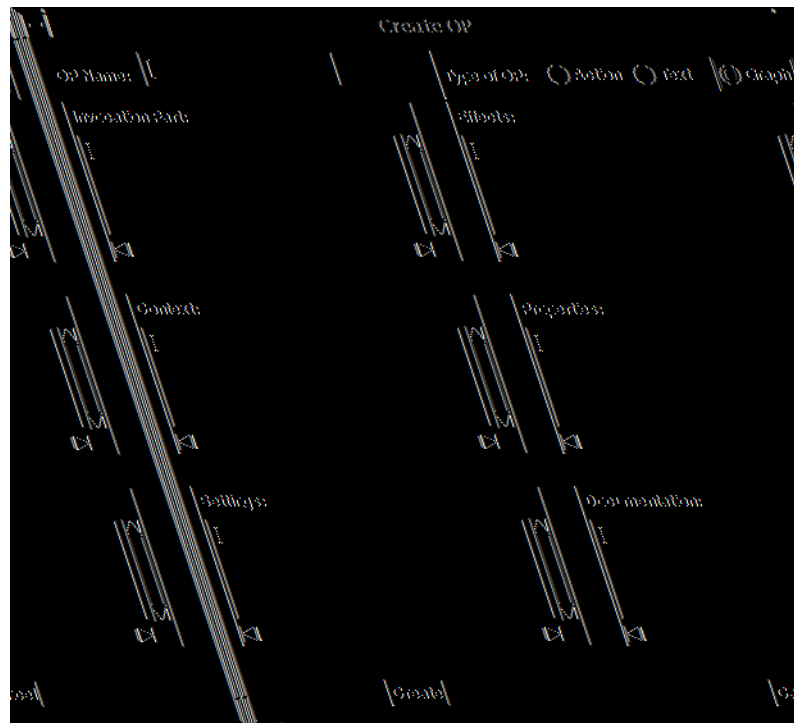
Figure 16.6: Create OP Dialog Box (Graph OP)

**Test Graph**　　　　　　　START

**INVOCATION:**
(ALARM $X)

**CONTEXT:**
(STATUS $X GOOD)

**SETTING:**
(CONNECTED $X $Y)

**EFFECTS:**
((~) (ALARM $X)))

**PROPERTIES:**
((EMERGENCY MAXIMUM)
 (PRIORITY 4))

**DOCUMENTATION:**
"This is a dummy procedure to
illustrate the Create KA dialog Box."
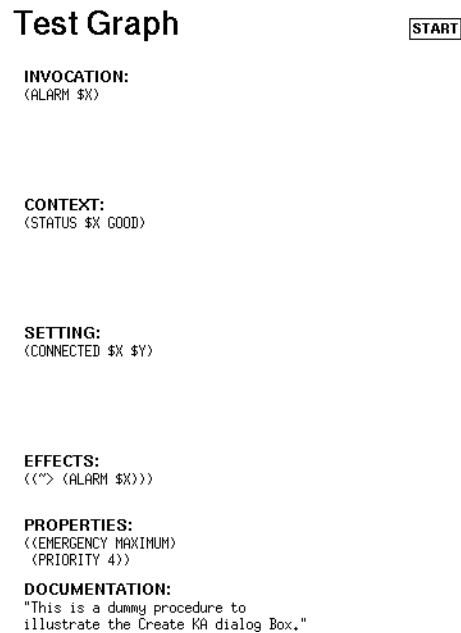
Figure 16.7: Resulting Graph OP

### Rename OP

This command allows the user to rename the currently Selected OP. A prompt dialog box pops up for you to give the new name of the selected OP. The text field is initialized with the previous name.

### Print OP

Will print the current OP using the print command (see [Change Print Command], §16.1.4, page 232). This is done by dumping a bitmap of the OP in a temporary file, and then by processing the resulting file with the print command. The temporary file is then deleted. This OP is dumped in `xwd` format. It used to be in `xpm` format, but the `xwd` format is much smaller on disk and much faster to produce.

### Dump OP Pixmap to File

This command allows the user to dump a bitmap or pixmap of the current OP. A file selection dialog box pops up to allow the user to choose in which file the pixmap should be dumped. Note that the resulting file is **'HUGE'**...
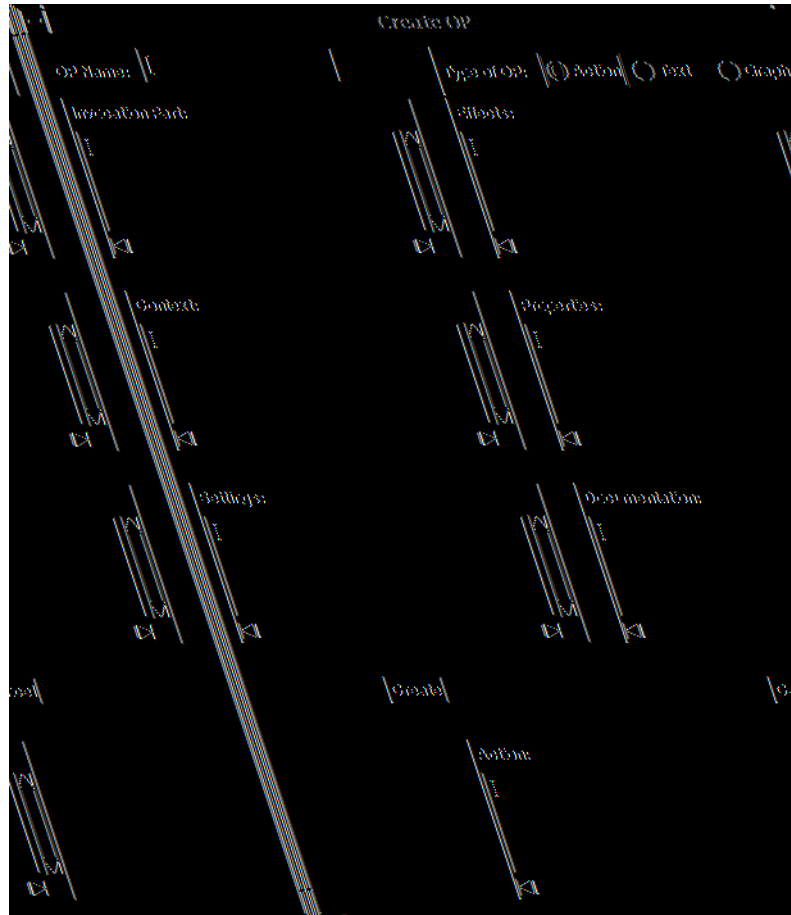
Figure 16.8: Create OP Dialog Box (Action OP)

## Test Action

**INVOCATION:**
(ALARM $X)

**ACTION:**
(SHUTDOWN $X $Y)

**CONTEXT:**
(STATUS $X GOOD)

**SETTING:**
(CONNECTED $X $Y)

**EFFECTS:**
((~> (ALARM $X)))

**PROPERTIES:**
((EMERGENCY MAXIMUM)
 (PRIORITY 4))

**DOCUMENTATION:**
"This is a dummy procedure to
illustrate the Create KA dialog Box."

Figure 16.9: Resulting Action OP

### Destroy Current OP

This command allows the user to destroy the current OP.

'**Caution:**' The current version of the OP Editor does not have an Undo facility... So extreme care should be exercised when using this command.

### Destroy A OP

This command allows the user to destroy a OP selected with a OP Selection dialog box.

'**Caution:**' The current version of the OP Editor does not have an Undo facility... So extreme care should be exercised when using this command.

### Previous OP

This command allows the user to quickly select the previous OP in the same OPFile.

### Next OP

This command allows the user to quickly select the next OP in the same OPFile.
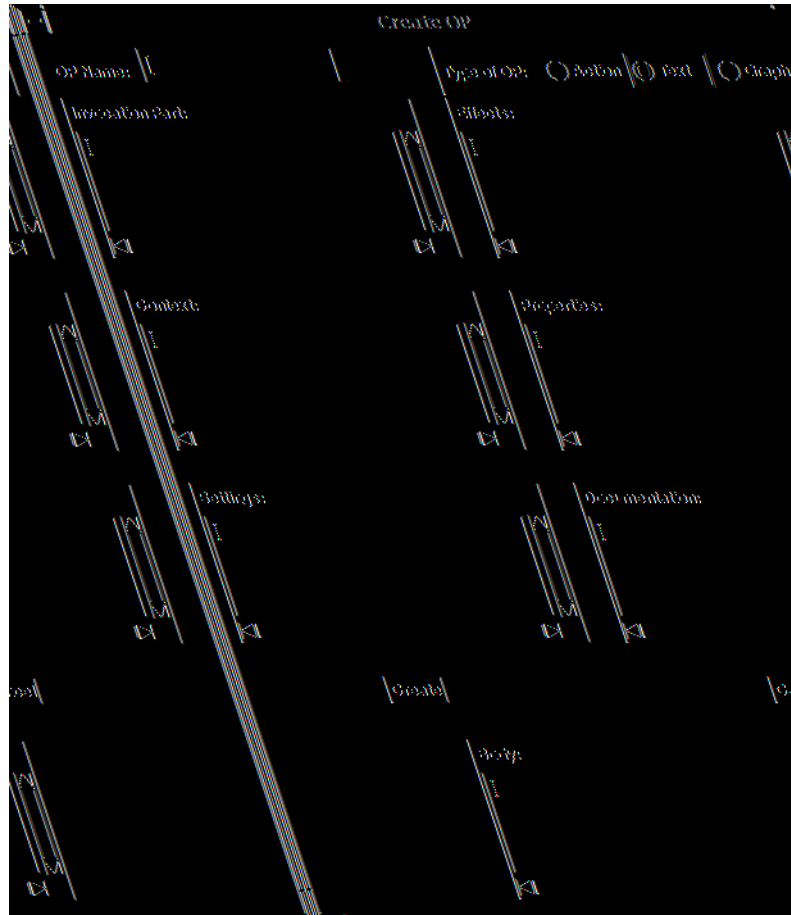
Figure 16.10:  Create OP Dialog Box (Text OP)

## Test Text

**INVOCATION:**
(ALARM $X)

**BODY:**
(;;; This is a Dummy comment.
 (! (SHUTDOWN $X $Y))
 ;;; Another comment.
 )

**CONTEXT:**
(STATUS $X GOOD)

**SETTING:**
(CONNECTED $X $Y)

**EFFECTS:**
((~> (ALARM $X)))

**PROPERTIES:**
((EMERGENCY MAXIMUM)
 (PRIORITY 4))

**DOCUMENTATION:**
"This is a dummy procedure to
illustrate the Create KA dialog Box."

Figure 16.11: Resulting Text OP

**Toggle selected OPs**

This command allows the user to quickly select the previously selected OP, even
if this OP is in another OPFile.

**Last Selected OPs**

This command allows the user to quickly select a OP in the list of the previously
selected OPs, even if this OP is in another OP File.

**Copy OP in Buffer**

This command allows you to copy a OP in an internal Copy/Paste buffer. This
OP can then be retrieved or pasted in another OP File, using the Paste OP
from Buffer command.

**Paste OP from Buffer**

This command enables the user to retrieve one or more OPs from the Copy/Paste
internal buffer and paste it in the current OP file.

Figure 16.12: OP Editor Misc Menu

**Clear OP Buffer**

Clear all the OPs in the Copy/Paste buffer.

## 16.1.4   Misc Menu

This menu (Figure 16.12) contains commands to customize the OP Editor or the appearance of the OP on the screen.

**Change Print Command**

Allow the user to change the print command. The default value is: `xwdtopnm < %s | pnmtops -r | lpr`. The default value is the one defined in the *'Op-editor.ad'* resources file (see [OP Editor Resources], §L.3.1, page 389). It can be modified by the user (by setting his own resource).

In any case, you must specify a command which contains a `%s` to specify the argument to which it will apply (the temporary file where the OP has been dumped). Moreover, this command must be able to send a `xwd` (it used to be in `xpm` format) to the printer. We strongly advise the user to get the pbmplus pacopge by Jef Posopnzer to print the result [Pos89].

The default command converts from xwd to pnm, then transforms it in encapsulated postscript with run length encoding and finally pipes it in the printer...

Note that you can use this command to dump OPs in Encapsulated Postscript format (to include them in user manuals for example) by redirecting the last command in a file instead of piping it in `lpr`.

**Change Drawing Size**

This command pops up a small window (see Figure 16.13) containing the current size of the drawing area in pixel. You can then change it. It is not recommended to reduce it as you could render some parts of a OP invisible or undisplayable (the invisible information is still present though, you just cannot access it).

**Symbols List**

This command pops up a small window (see Figure 16.14) containing the current list of symbols declared in this OP file. You can then change the content of this
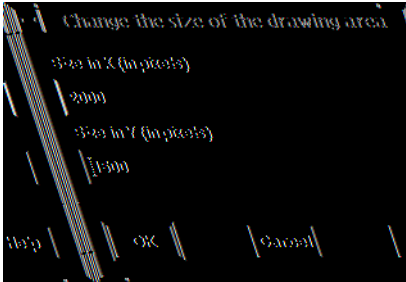
Figure 16.13: Change Drawing Size Dialog Box
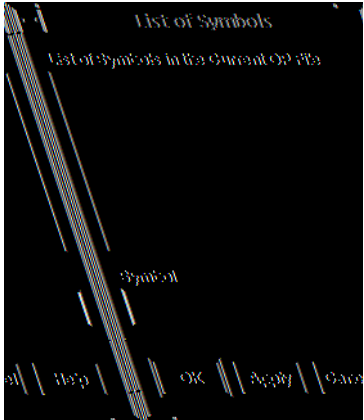


Figure 16.14: Symbol List Dialog Box
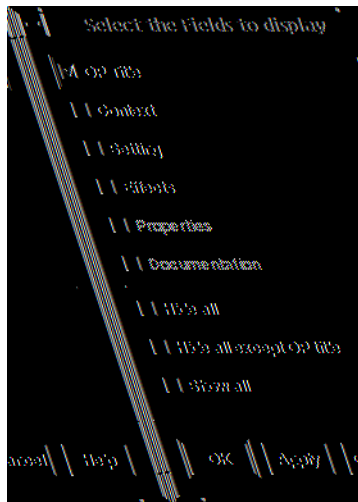
Figure 16.15: Selected Fields Dialog Box

list which is saved within the OP file.

**Change Visible Fields**

When selected, this menu item pops up a dialog box (see Figure 16.15) which enables you to tune the way a OP is displayed on the screen, dealing with the visibility of unused or unnecessary fields. The dialog box offers you a list of push buttons corresponding to the different text fields of a OP (documentation part, context, and so on). These push buttons are `on` when the corresponding field is visible. They are `off` when it is invisible, and they are insensitive when the corresponding field is not empty (in which case they cannot be made invisible). You also have a `all` button and a `none` button which allow you to make all the fields (to which the operation applies) visible or invisible.

Note that you cannot hide a non empty field, and you cannot hide the Invocation Part (which is always required). Similarly, to edit a field, you need to make it visible first.

Note that you can hide the name of the OP if you want.

## 16.1.5   Mode Menu

This Mode menu (Figure 16.16) contains commands equivalent to the one in the working menu (see [Working Menu Items], §16.2, page 235). Its purpose is mainly to enable the use of accelerators on the working menu (as Motif only allowes the use of accelerator on button in menu).
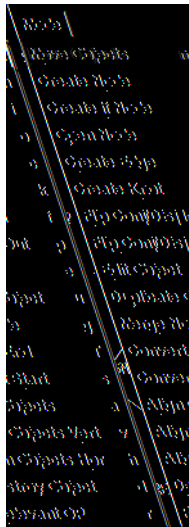
Figure 16.16: OP Editor Mode Menu



Figure 16.17: OP Editor Help Menu

### 16.1.6   OP Editor Help Menu

The help menu (Figure 16.17) contains a number of items which, when selected, pop your HTML browser with the proper documentation. The first item is Help and shows the top level directory of the documentation. The other item pops up the browser but at the selected documentation section.

## 16.2   Working Menu Items

The Working Menu (Figure 16.18) contains the commands which are most often used to edit a OP. Most of them deal with the components of a OP. They remain selected until you select another mode. For example, if you have selected `Destroy Object`, you stay in this mode until you select another mode (this can be rather dangerous). By default, the OP Editor starts in the `Move Objects`
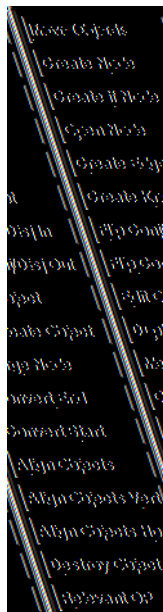
Figure 16.18: OP Editor Working Menu

mode. Independently of any working selected menu, middle click is always equivalent to Move Objects and right click (whenever it is available) to Edit Object.

### 16.2.1    Move Objects

This command is used to move objects on the screen. The movable objects are: nodes, knots, text fields, edge texts. Just click on them and a bounding rectangle appears. You can then drag the object to its new position.

Note that if you click on the background window (i.e. not on an object), the whole window moves. This can be faster than using the scrolling bar.

### 16.2.2    Create Node

Create a new node wherever you click.

### 16.2.3    Open Node

A new node will be created just under the node on which you click. The outgoing edge from the first node will be transferred on the second one. In other word, you just opened the node to insert an edge on it.
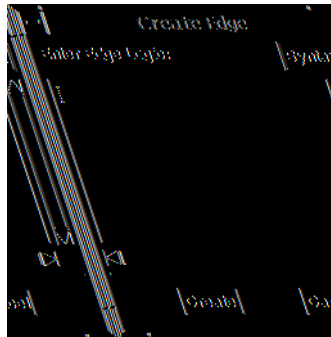
Figure 16.19: Create Edge Dialog Box

## 16.2.4   Create If Node

This command can be used to create IF-THEN-ELSE nodes.

## 16.2.5   Flip Conj/Disj Out

When selected, this command allow the user to flip the status (disjunctive/conjunctive) of the outgoing edges of a node. Conjunctive corresponds to creation of new threads.

## 16.2.6   Flip Conj/Disj In

When selected, this command allow the user to flip the status (disjunctive/conjunctive) of the ingoing edges of a node. Conjunctive correspond to merge of multiple threads.

## 16.2.7   Create Edge

Create an edge between the first selected node and the second one. You can create intermediate knots by clicking on their desired position. If, after selecting the first node, you change your mind and want to unselect it, just click on the right mouse button. This will unselect the first selected node.

If you have selected a second node, a prompt dialog box pops up (see Figure 16.19) and asks you to enter the goal to put on the edge.

## 16.2.8   Create Knot

You first need to select an edge to which you want to add a knot by clicking on the text part of the edge. Then, you can add as many knots as you want. Click right to unselect the edge.
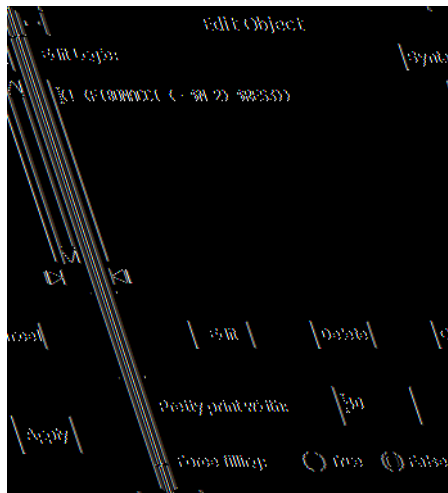
Figure 16.20: Edit Object Dialog Box

### 16.2.9   Duplicate Objects

Duplicate a node or an edge. For the node, it will duplicate all its ingoing and outgoing edges.

### 16.2.10   Merge Node

Merge two nodes and all their edges.

### 16.2.11   Edit Object

When this command is selected, the user can edit any editable object (i.e. text fields, edge texts and node names). A modal dialog box appear (Figure 16.20) and the user can change the object edited. This dialog box has two particularities, there is a text field where the user can enter a pretty print width, and a toggle button to specify if the pretty printer should fill up lines or not (see [Pretty Printing], §15.8, page 219). The OP Editor is quite rigorous for the syntax allowed in the text field. If anything wrong is entered, you get an Error Dialog Box.

### 16.2.12   Convert End

This enables the user to convert (or unconvert) node to end node. The system checks that the node you are converting is a leaf of the graph or that the node is not the Start node.

### 16.2.13 Convert Start

This enables the user to convert a node to a start node. Considering that only one start node is allowed, converting a node to start node unconverts the current start node which will therefore automatically become a standard node. The OP Editor checks that the nodes you are converting do not have ingoing edges or are not the start node.

### 16.2.14 Align Object

This command enables the user to align nodes and text fields. You first select an object (the anchor) on which you want to align other objects. Then you click on the objects to align. It will align the other objects on the closest of the vertical or horizontal line going through the anchor. If you want to unselect the anchor, you need to press the right mouse button.

### 16.2.15 Align Object Vert

This command enables the user to align nodes and text fields. You first select an object (the anchor) on which you want to align vertically other objects. Then you click on the objects to align. If you want to unselect the anchor, you need to press the right mouse button.

### 16.2.16 Align Object Hor

This command enables the user to align nodes and text fields. You first select an object (the anchor) on which you want to align horizontally other objects. Then you click on the objects to align. If you want to unselect the anchor, you need to press the right mouse button.

### 16.2.17 Destroy Object

This command is used when the user wants to destroy some objects on the screen. Note that the destroy operation is done on the mouse up event... On the mouse down event, the OP Editor shows you (by selecting it) the object which is going to be destroyed. However, if (while the button is still pressed) you move the mouse away from the bounding box of the object, it is not destroyed. A recent version of the OP Editor allows the user to destroy the knots of an edge.

'**Caution:**' The current version of the OP Editor does not have an Undo facility... So extreme care should be exercised when using this command.

### 16.2.18 Relevant OP

This command can be used to find out what are the OP relevant to a particular goal. When this mode is selected, click on an edge (actually, the text of the

goal), and the OP Editor will propose a list of relevant OP (among the OP currently loaded in the OP Editor).

# Chapter 17

# OP File Format

Version of the OP Editor prior to 1.3 recognizes three different OP file formats. The OPF File format is the official format used by the various components of the OPRS Development Environment. However, the two other formats are used for compatibility with SRI Lisp OPRS version.

Versions of the OP Editor greater or equal to 1.3 only recognize the OPF format. If you still have old files, use a version of the OP Editor prior to 1.3 to convert them.

## 17.1 OPF Format

The OPF format is the default file format used by ACS OP Editor. It has been designed for OPRS Development Environment. It is the one and only one format recognized by the OPRS Kernel. Moreover, the OP Editor knows how to write OP files only in this format (but can read other formats). There exist different version of this format, but this is an internal flag which allow the modules of the OPRS Development Environment to distinguish between them.

## 17.2 GGRAPH Format

The GGRAPH (Grasper Graph format) is provided for compatibility. The OP Editor is able to read this format and understand the various graphical information it contains. However, several details (usually unnecessary to the OPRS Kernel) are not properly parsed in this format (such as the fonts used on a particular edge).

For various reasons, one cannot guarantee that the OP Editor will properly parse all the Grasper Graph files. If you have any problem with a particular file, check the list of known problems with this format and report your problem to `oprs-bug@ingenia.fr`.

### 17.2.1   How to Get Grasper Graph on your Lisp Machine

In Grasper, load your OP graph, and select it. Make sure you are in the proper pacopge (most likely OPRS), and execute the following code in the Grasper Listener pane:

```
(let ((*print-length* nil)
      (*print-level* nil))
    (with-open-file (foo-graph "foo.ggraph" :direction :output)
                            (format foo-graph "~s" (grasper:describe-graph)))
```

This saves your graph in the file *'foo.ggraph'*.

**'Warning:'** These text-graphs are different from those you get when you execute the command Output-SUN under Grasper (which delivers a SGRAPH format)... The former contains positions and graphical information, the latter does not.

It has been reported that a nickname given to the pacopge `GRASPER` causes Grasper to save the file with the nickname instead of the original `GRASPER` symbol. The OP Editor expects the `GRASPER` symbol... So at worst, you need to remove this nickname before saving, or replace the nickname with `GRASPER` under a text editor.

### 17.2.2   Grasper Graph Incompatibilities

While loading a Grasper Graph, few things are not properly recognized by the OP Editor:

#### Unrecognized Symbol

Replace `|#:|` with `|#` in node names like: `|#:|e7756|` These node names come from the time of ZetaLisp.

In the following forms, the & and ! are not parsable.

`(*FACT (EQUAL & (CAADR (GOAL-STRUCT-STATEMENT (OP-INSTANCE-GOAL (FIRST $X))))))`

`(*FACT (EQUAL ! (CAR (GOAL-STRUCT-STATEMENT (OP-INSTANCE-GOAL (FIRST $X)))))))`

The reason is quite obvious: these symbols should not be used explicitly in OPs and these tests should be replaced with evaluable predicate doing the same thing.

#### Negation as Failure

—Meta  ! (negation as failure)— (which is a OP in default-processes.graph) cannot be parsed because of the absence of negation as failure in OPRS. As a consequence, the expression `(!  (  $X))` cannot be parsed as a Temporal Expression (according to OPRS Grammar).

**Badly Formed Goals**

The following goal is not allowed: `(! (SEND-COMMAND-ACTION (DISPLAY-FOO (((CURRENT $O) (CURRENT $B))))))` because `(((CURRENT $O) (CURRENT $B)))` is not a valid *Composed Term*.

This goal: `( ! (& ( > (FOOBAR $P $A)) (=> (FOOBAR $P $A)))))))` is not allowed anymore..., replace it with: `(& ( > (FOOBAR $P $A)) (=> (FOOBAR $P $A))))))`

## 17.3 SGRAPH Format

The SGRAPH or SUN Graph format is provided for compatibility with a OP Format which has been used on SRI Sun version of the Lisp version of OPRS.

Note that this format does not have any graphical information. Consequently, it should be used in the last resort.

If you load a SGRAPH format file, nothing shows on the screen (this format does not include any graphical information...). You have to save it first (in OPF format) and read it again. The various OP fields are positioned at their default position... But all nodes and edges are grouped in one position (20 20). You can then rearrange them as you wish. A future version of the OP Editor may attempt to display these graphs with a better organization, however, the current one does not (Note that if you had the possibility to generate OP File in SGRAPH format, you could also generate it in GGRAPH format, which is preferred).

# Part IX

# Using OPRS

# Chapter 18

# Introduction to Using OPRS

Now that you have read through the whole documentation (you still have the Appendices. . . ), you may start to wonder how the different parts fit in. You have got this real-time, control and supervision application to solve, and you get the strange but growing feeling that OPRS is the tool you need. Good. This part of the manual go through the various process you have to do to set up your own application.

We are first going to go through a simple OP Editor and X-OPRS Kernel session. We will then examine the various questions you have to ask yourself before getting started. How many kernels do I need? what do I put in the database? how many OPs? do I need meta level reasoning? how can I optimize my application? etc...

Then we will examine various applications and justify the answer to the aforementioned questions.

# Chapter 19

# Setting Up your Environment

To be able to use the OPRS Development Environment you may need to set up a number of environment variables.

```
setenv OPRS_DIR /usr/local/oprs
setenv PATH "${PATH}:${OPRS_DIR}/bin"
if ( $?XFILESEARCHPATH ) then
        setenv XFILESEARCHPATH "${XFILESEARCHPATH}:${OPRS_DIR}/\%T/%N"
else
        setenv XFILESEARCHPATH "${OPRS_DIR}/%T/%N"
endif
setenv OPRS_DOC_DIR ${OPRS_DIR}/doc
setenv OPRS_DATA_PATH  ${OPRS_DIR}/data:./data:.
```

# Chapter 20

# Getting Started

To understand this chapter, the reader is supposed to have some common knowledge about the following subject: Unix, X11, X server, Motif, Xt.

Assuming OPRS Development Environment has been properly installed at your site, and assuming you have the OPRS Development Environment in "your path" (i.e. Unix will find the command if you invoke them), then you can now start to use OPRS.

## 20.1   Getting Started with the OP Editor

To get started, you can probably just call the OP Editor. This tool described in detailed (see [How to Use the OP Editor], §15, page 215), is used to display/edit/create procedures as they are represented in used in the OPRS Development Environment.

The OP Editor is a graphical tool and can only be used if and only if you are running an X server. To call the OP Editor, just type (at the Unix prompt):

```
% op-editor
```

This should bring the OP Editor window. Then, you can either load an already existing OP file, or you can start to create your own procedure.

To load an existing OP File, you must select the Load OP File menu item from the File menu, [Load OP File], §16.1.1, page 221. This will bring a File Selection Dialog Box from which you can select a file to load. Usually, data files are available in the '/usr/local/oprs/data/' directory. If your version of the OP Editor is properly installed, the File Selection Dialog Box should point to this directory.

After the OP file is loaded, you will be asked (with a List Selection Dialog Box) to select which procedure to edit from this file (keep in mind that a OP File can contain more than one procedure). If you select a procedure, then it should appear on the screen. In case, you do not select any procedure, you can still select a procedure later by using the Select OP menu item from the OP Menu [Select OP], §16.1.3, page 225.

You can now browse through the different menus available in the menu bar, [Menubar of the OP Editor], §16.1, page 221. You will see that there are many commands available to select/load/save/unload OP File, as well as command to deal with procedures and OPs.

If you decide to create your own procedure, you must select the Create OP menu item from the OP menu (see [Create OP], §16.1.3, page 225 and see [Creating a OP], §15.3, page 218).

When you have a OP selected on the screen, you need to choose from the Working Menu (the menu on the left of the drawing area) which operation you want to perform. Here also, there are a number of operations available. Each operation is more or less self explanatory, when selected, it stay selected until you select another operation. Note that at any time you can consult the Help Footer (see [Footer and Dialog Box Help], §15.7, page 219) for information on how to proceed.

## 20.2   Getting Started with the X-OPRS Kernel

Before starting a X-OPRS Kernel, you must start a OPRS-Server (this operation will also start a Message Passer). If you attempt to run a X-OPRS Kernel or a OPRS Kernel without any OPRS-Server running, the program will exit with an error message stating that it cannot connect to the Message Passer (which is mandatory for X-OPRS Kernel and OPRS Kernel).

So, before doing anything, you need to call a OPRS-Server by issuing the following command at the Unix prompt:

`% oprs-server`

From then, you can either start a X-OPRS Kernel from the OPRS-Server by issuing the command at the OPRS-Server prompt:

`OPRS-Server> make_x foo`

This will create a X-OPRS Kernel named FOO. Another way to obtain a X-OPRS Kernel is to issue (at the Unix prompt) the command:

`% xoprs foo`

and then to issue (at the OPRS-Server prompt) the command:

`OPRS-Server> accept`

In both case, you should get the X-OPRS Kernel window on your screen.

At this stage, this X-OPRS Kernel is fully functional and can execute any procedure you will load in it. To try a well known example, you can load the file *'/usr/local/oprs/data/fact-meta.inc'*. This include file will load various OP files (*'/usr/local/oprs/data/fact-meta.opf'* and *'/usr/local/oprs/data/new-default.opf'*). To load the include file, you have to select the Include File command from the File Menu (see [Include], §13.3.1, page 180).

You should see in the text pane various information about the procedures which are loaded.

Now, your kernel is running, it has some procedures loaded, you need to post a new goal or a new fact. For this particular example you just loaded, you need to post the goal: `(!  (print-factorial 3))`, or any particular integer

value. If you try with an integer ¿ 30, you will most likely reach the maximum integer value, and will get an erroneous result. You should then use float (`(!` `(print-factorial 150.0)))`. In any case, to post a new goal or a new fact, you need to select the Add Fact or Goal menu item from the OPRS menu (see [Add Fact or Goal], §13.3.2, page 184), type the goal, and click on OK. You will see the result printed on the screen, in the text pane. During the run, you may noticed some activity in the intention graphic trace pane. It is the evolution of the different intentions which appear and then disappear.

You can make as many runs as you want and play with the different trace and execution options available to the user. For example you can select the OP Graphic Trace menu item in the Trace Menu (see [OP Trace/Step], §13.3.4, page 196). This will pop up a List Selection dialog box where you can select the OP for which you want a graphic trace of their execution. For this particular application, it is a good idea to trace the OP: `Print Factorial, Iterative Factorial, Recursive Factorial and Meta Factorial`.

Now that you are becoming more and more familiar with the interface, you can start to play with the See [Control and Status Panel], §13.4, page 201. For example, you can halt the kernel, then add a new goal, and then click on Step of Next to see a step by step execution (see [Control and Status Panel], §13.4, page 201, for an explanation of the differences between Next and Step.

# Chapter 21

# Setting Up a OPRS Application

Setting up a OPRS application is an easy task. However, there are a number of questions you have to ask yourself, or ask the final users or the experts who will bring the knowledge in the system. These questions mainly relates to the structure of the problem and to the way you want to solve it using OPRS. Keep in mind that OPRS is a shell, a tool, and if its functionalities makes it particularly well suited for a certain type of application, you still need to do some work to organize your application.

In this chapter we will examine all these questions you have to ask yourself and put forward some elements to help you to answer them.

## 21.1 How Many OPRS Kernels Does it Takes to Screw a Light Bulb?

The answer is 42.

In fact, this is indeed a question you have to ask yourself before starting to implement your OPRS application. The answer is in your application itself, you have to analyze it and figure out the answer. To help you we can go through a number of criteria which participate to the decision.

**How many agents are involved in your application?** Indeed, the number of agents are currently used to solve the problem is a good indication of the number of kernels you may need to solve it with OPRS. For example, if your problem involved two agents working on their own part of the problem, exchanging information from time to time, then using two kernels seems a wise choice. Another possible situation is when you have a large number of agents doing "more or less" the same task, but either in a distributed way, or for their own objectives.

**Can all the computation be done by one kernel?**   This is an important
consideration. You must evaluate the amount of computation which will be done
for the application and consider its distribution if it appear to be to large for a
single kernel/platform. Keep in mind that the kernel will not only execute the
procedure you loaded but also the evaluable functions, predicates, and actions
you have linked in the kernel. Some of these operations can be rather time
consuming and required to be sliced or distributed.

**Is the application composed of different independent modules or steps?**
When the resolution of the problem is well defined in different modules, or can
be broken down in a number of well separated steps, then one can consider
implementing these different modules or step in separate kernels.

**Is the shared information between modules small?**   This is an important
consideration to take into account. You have to make sure that the amount
of information between the different kernels is not too large. Otherwise, you
will loose in communication time, the time you have spare in distributing your
application.

## 21.2   OPRS Kernels or X-OPRS Kernels

This question is easy to answer. First from a functional point of view, the two
kernels are identical. However, due to some obvious overhead induced by X and
Xt the X-OPRS Kernel is slower than the OPRS Kernel. Keep in mind though
that the X-OPRS Kernel provides some interesting tracing and debugging capa-
bilities which are missing in the OPRS Kernel (the OPRS Kernel still provide
some trace, but they are not as visual as the one provided in the X-OPRS
Kernel).

It is usually a good practice to develop the various OPRS agents with the
X-OPRS Kernel, just to be able to trace and debug the OPs. Then, when a
set of procedure has been debugged and run flawlessly, one can load them in
a OPRS Kernel, you should not have any problem, except with the increased
performance.

Note however, that for some applications, the final user or the operator may
need to "see" the procedures executing. Therefore, it may be necessary to run
these procedures in an X-OPRS Kernel.

Similarly, there are some applications for which there are no need to follow
the graphic execution of the procedure, or to follow the graphic evolution of the
intentions graph. In this situation, one may run these procedure in a OPRS
Kernel.

Of course, one can mix X-OPRS Kernels and OPRS Kernels in the same
application.

An important criterion to evaluate, regarding the use of OPRS Kernel or
X-OPRS Kernel, is the respective size of these two kernels. The size of each

kernel depends of the availability of shared libraries on your system, the debug flag with which the kernels were compiled and more generally the size of binaries on your machine. Nevertheless you will noticed a great difference in size between the X-OPRS Kernel and OPRS Kernel (the ratio can be of one to ten...). Therefore, depending on the size of the swap space and the size of memory on your machines, and the number of kernels involved, you may have to limit the number of X-OPRS Kernel.

## 21.3 The Database: Facts, Only the Facts

The database has a very important and critical role in a OPRS application. It is the memory of the "system", and it is heavily used by the kernel to retrieve, check, conclude information. There are many questions to be asked before you start building your database. This section will go through these questions and will bring some elements of answer.

### 21.3.1 The Representation of Facts

A very common question is what is exactly the proper way to represent facts or beliefs in the database? Which format should be used? What is the proper way to represent such and such information and what is the wrong way to do it?

The format used to represent facts in the database is simple: `(predicate-name <arguments>*)`.

The semantic associated with the predicate name and the arguments (and the order of the arguments) is yours, and should be consistent through the whole database and application.

For example, if you consider the predicate `position`, you probably want to associate it to the idea of devices or objects being in particular position. Such as in: `(position valve open)`, or `(position switch on)`, or even `(position robot unknown)`.

Note that for this particular predicate, we put first the argument to which this information apply, and then the argument which represent the "value" of this predicate for the first argument... However, this is not always the case. For example, a predicate such as `connected` which would represent that two entities are somehow "connected" in your application may not bear any particular meaning on the order of the argument. Therefore, `(connected a b)` would be semantically equivalent to `(connected b a)`. However, from a strict database point of view, these two facts are different and it is the user task to make these 2 equivalent when needed. To summarize, the order of arguments may be meaningful or not, but in any case, it is up to the user to ensure that this order is properly respected through the whole application when it is meaningful, and to ensure that it is properly used when it is meaningless.

An important consideration, is to keep in mind that the predicate name is always first, and cannot be replaced with a variable in any context. For example, it is forbidden to write in a OP something like: `(=> ($x a b))`... Even if the

variable `$x` is bound to the symbol `connected`. The reasons for this limitation take their roots in the "First Order Predicate Calculus".

The last consideration to take into account relate to the goal representation in OPRS. Do not forget that goals are built upon facts. According to the temporal operator used, they modify the semantic of the fact they are qualifying. As a consequence, it is important that the facts underlying the goal representation be consistent with the goal representation.

If one consider the fact `(position door open)`, then we can easily build on this fact to get the following goal: `(!  (position door open))` to close open the door if it is not already open, `(?  (position door open))` to test if it is currently open , `(^ (position door open))` to wait until somebody open the door (if it is not already open), etc... In all these examples, we can see that the fact and the goals are really carrying the wanted information. Moreover, when any of these goal is attempted, the corresponding facts is checked in the database...

### 21.3.2   Which Predicate?

Now that the notion of predicate is clear. Remains the problem of deciding which predicate should be used and for what.

First, the semantic of the predicates is the one the user wants to give to the predicate. If you want to use the word `position` to represent things we have nothing to do with position, you can... This will just make your OP unreadable...

Similarly, you can overload the semantic of a predicate. The predicate `position` can be used in an application to represent the position of various objects, and even different types of objects: switches, valves, lights, etc. An another alternative is to use predicate such as `switch-position` or `valve-position`. Here also, there is no rules, it is question of readability of the OPs and the database.

To help organize a database, very often, one use a `type` predicate to specify the type of the various object. This is very useful for example when one looks for all the open valve of the system:   `(& (type valve $v) (position $v open))`.

### 21.3.3   Which Predicate Should be Declared as Closed World Predicate?

This is an easy question to answer... In general, most if not all predicates should be declared as closed world predicates. Why? Because this is usually the way we tend to think, i.e. if something is not know to be true, then its contrary is usually true. To use the example given in section [Closed World Predicates], §5.5, page 83, if we do not have any information about a direct flight between Toulouse an San Francisco, most likely, no such flight exist, and therefore, its negation is true. However, keep in mind that this information will be looked at (the fact that this specific fact is not in the database, and the fact that it is a CWP) if and only if you consult the negation of the fact. In other words,

there are no need to declare CWP predicate for which you never retrieve the negation.

Last, there may exist some predicates which are not closed world predicate, i.e. for which a request of the negation of something which is not known as true is false...

### 21.3.4 Which Predicate Should be Declared as Functional Facts?

See [Functional Facts], §5.6, page 86 explains in great details the logic behind functional facts (in short FF).

The important things to remember are:

- FF are expensive.

- Only the predicates for which there exist a "functional" representation can be considered as FF.

- Declare predicate as functional fact, if and only if you want to insure an automatic clean up of previous value.

- Remember to order the arguments in such a way that the predicate can be declare as FF.

- Sometime, Basic Events (see [Basic Events], §5.7, page 88) and (see [Basic Events?], §21.3.5, page 259) are enough to represent facts you want to get deleted automatically.

Here are example of predicates which are often declared as FF: `position`, `status`, `connected`. For each of them one usually want to remove the previous value when a new value for a specific object is concluded.

### 21.3.5 Which Predicate Should be Declared as Basic Events?

This is another easy question to answer. You want to declare as Basic Events (see [Basic Events], §5.7, page 88), all the predicate you want to trigger OP applicability or intentions/threads awaking, but that you do not want to keep in the database. This is to represent events which are ephemeral, still need to be considered by the kernel main loop, but do not need to be remembered in the database for future use.

### 21.3.6 Forbidden Things and Things to Avoid with the Database

There are a number of things one should not do or avoid while using the database mechanism. Here is a non exhaustive list of forbidden things:

- Do not conclude facts containing unbound variables.

- Do not consult evaluable predicate with unbound variable (unless you predicate is able to handle the situation, which is unlikely...)

Here is a non exhaustive list of things to avoid as much as possible:

- Avoid consulting big disjunction, which can lead to a huge number of possible results. If you must use disjunction, try to put the most discriminating facts in first.

- Avoid using long argument list for facts.

- Remember to tune the size of the hashtable according to the number of facts you are concluding in the database.

- Do not conclude evaluable predicates.

- Do not conclude facts containing variables bound to objects which are, by their nature, pointing to internal objects, themselves pointing to other internal objects. This will not break the system, but this will result in ever growing kernel (as the Garbage Collector will be unable to collect the internal data concluded in the database).

- Do not consult negation of Closed World predicate with unbound variable, as you will not get binding even if the negation is true.

- Do not forget to clean up your database. Many application grows indefinitely because of a bad database clean up.

## 21.4   Which OP for Which Task?

### 21.4.1   Fact Invoked OPs

Fact invoked OPs are usually written to respond or react to events. They do not have any explicit goal or objective.

These OPs usually correspond to events you have to monitor or you need to check, such as alarm, or change of values, etc.

### 21.4.2   Goal Invoked OPs

Goal invoked OPs are written to achieve a particular goal or objective. Their success is equivalent to achieving the goal which invoked them.

These OPs are usually implementing a goal directed behavior, i.e. the execution of these OP is supposed to achieve the goal for which they are applicable.

### 21.4.3 Predefined OPs

A number of predefined OPs can be loaded in a specific application. The user can pick up OPs in the different OP files provided with the distribution (see [Default OPs], §F, page 315). For example, the OPs in the file *'new-default.opf'* (see [new-default.opf], §F.1, page 315) provide a number of interesting functionalities. In fact, most application load this OP file. Even if it is not required, it is strongly recommended, as OP as basic as = are defined in this file.

Most of the default OPs defined in *'new-default.opf'* are actions OPs. The actions they called are usually documented and can be directly used by the user.

## 21.5 User Defined Evaluable Functions

Do you need to defined your own evaluable functions? This depends of your application. Do not look after them, the need will come in its time.

You may need evaluable functions for one the following reasons:

- to perform some specific computation on some Terms. For example, you may have to define the `max` function which taking to integers return the biggest of the two. Similarly, you may want to defined a function which concatenate two stings, etc... That's up to you.

- to perform some specific computation on some user defined objects (`U_POINTER` Term). For example, you may have defined your own user type, a fancy C structure which hold various information required by your application, and you want now to write a function which return the string contained in a specific field of this object.

In any case, the need for evaluable function is easy to identify, and easy to solve. Keep in mind though, that evaluable function are called whenever they are encountered while posting a goal containing a composed term which function name is an evaluable function. As a consequence, the time taken by these function should be as short as possible. Contrary to Actions, the execution of evaluable functions cannot be time sliced.

Last, there are a number of things to consider when you write evaluable functions:

- Check the number of arguments which is passed to your function.

- Check the type of each argument you are getting. (do not assume that the user will always pass the argument of the right type at the right position).

- Check that all the arguments are bound. This relates to the previous comment as you will get a Term of type `VARIABLE` if a variable is unbound...

See [How to Define your Own Evaluable Functions], §6.2, page 107, for more information on this subject.

## 21.6   User Defined Evaluable Predicates

As evaluable functions, evaluable predicates may be required in some application:

- to perform some specific computation on some Terms. For example, you may have to define the `ordered` predicate which taking a list of integers return `TRUE` if the list is ordered, `FALSE` otherwise.

- to perform some specific computation on some user defined objects (`U_POINTER` Term). For example, you may have defined your own user type, a fancy C structure which hold various information required by your application, and you need a predicate which evaluate to `TRUE` if a this structure satisfy a particular condition.

Last, there are a number of things to consider when you write evaluable predicates:

- Remember that you just need to return a `PBoolean`, i.e. `TRUE` or `FALSE`, not a Term.

- Check the number of arguments which is passed to your function.

- Check the type of each argument you are getting. (do not assume that the user will always pass the argument of the right type at the right position).

- Check that all the arguments are bound. This relates to the previous comment as you will get a Term of type `VARIABLE` if a variable is unbound...

See [How to Define your Own Evaluable Predicates], §5.8.2, page 93, for more information on this subject.

## 21.7   User Defined Actions

Do you need to defined some actions? Here also, this depends of your application. It usually depends of what kind of interaction or "actions" (thus their name) you need to perform with the kernels. If your application only prints some statements (such as advices to an operator), then you may not need to define any actions. On the opposite, if your application need to directly interact or act with subsystems, then you will need to define some action.

Keep in mind that if your application is organized in such a way that there is module which perform the real action on the world, and that this module is connected to the Message Passer, then most if not all the actions are done using the Message Passer (i.e. by sending a message to the module in charge of performing the action (a simulator or an interface to the real system).

Nevertheless, you may need actions for one the following reasons:

- to associate a specific functions to a OP. Thus enabling the action to succeed or fail and make the corresponding OP returning the corresponding result. Many users tend to forget that evaluable functions (as presented above) are evaluated at posting time, i.e. when the goal in which their form is contained is posted. But their evaluation is always meaningful and does not bear any success or failure. For example, in `(!  (foo (+ 3 4)))`, the `(+ 3 4)` will be evaluated right away (unless you have unselected the `eval_on_post` option). However, this evaluation is always feasible and always gives a result. For an action (in fact a standard action), the result can be either `T` or `NIL`, and upon this result depend the success or failure of the OP which calls it.

- to perform some actions using code linked in your kernel. Keep in mind that when you build your application, you will, or may link some code in it. For example, if your application is linked (and by linked we mean the operation of linking in one program a number of functions calling each others) to an application dependent library and that you want to call some of these functions which may fail, then you need to define actions to call these functions.

- to time slice the execution of functions which take to long to execute. Keep in mind that the reactivity of your system depend of the longest action/evaluable functions of your application. Therefore, to increase the reactivity of your kernel, you may have to time slice the execution of these actions.

- to write functions which return a list of terms instead of one term. See [Multi Variable Special Action], §4.3.3, page 67 for more details.

Last, there are a number of things to consider when you write functions:

- Check the number of arguments which is passed to your function.

- Check the type of each argument you are getting. (do not assume that the user will always pass the argument of the right type at the right position).

- Check that all the arguments are bound. This relates to the previous comment as you will get a Term of type `VARIABLE` if a variable is unbound...

- Make sure you return a **'new'** Term, and this recursively... Do not return for example one of the argument passed to the functions "as is", or a `LISP_LIST` containing object which have not been created by you.

- Make sure your function return the Term containing either:

  - The symbol `T`, the symbol `NIL` or the symbol `:wait`, if it is a Standard Action. See [Standard Action], §4.3.3, page 66 for more details.

  - Any kind of term or the symbol `:wait`, if it is a Special Action. See [Special Action], §4.3.3, page 67 for more details.

  – A list (`OPRS_LIST`) of any kind of term or the symbol `:wait`, if it is
    a Multi Variable Special Action. The list should have exactly the
    same number of element as the list of variable it is unify against. See
    [Multi Variable Special Action], §4.3.3, page 67 for more details.

See [How to Define your Own Actions], §7.7.2, page 122, for more information
on this subject.

## 21.8   Do You Need Meta Level?

This question is somewhat difficult to answer. The following points have to be
considered before making a decision.

  • Do you need a particular type of control in the main loop?

  • Do you have to choose among various applicable OPs with a specific heuristic?

  • Do you mind "loosing" or forgetting external events?

In any case, always use the right level of options in the kernel to suit your
meta level needs. For example, if your kernel does not use any meta level
reasoning, you should disable it in the kernel (see [Meta Level Reasoning], §9,
page 131. However, if it does, for example use the `SOAK` meta fact, you should
enable it and enable the conclusion of this particular fact (and disable all the
other one), see [OPRS Kernel Meta Level Option Commands], §2.7, page 33.

## 21.9   Intention Graph Manipulation

## 21.10   Data and Commands

If your OPRS application is embedded in a "real" or simulated environment, you
will probably get some data from this environment and you may send commands
to it.

In most case, these data and these commands will transit through the Message Passer.

Example are provided in the *'demo'* directory of connection between applications and OPRS Kernel through the Message Passer.

However, you can also design your own communication interface, which will
interact with the external application through evaluable predicate, functions or
actions.

## 21.11 Linking C Code in the Kernels

To be able to execute your own code in a OPRS Kernel or X-OPRS Kernel, you need to link it to relocatable kernels.

Technically, a relocatable is an "almost executable" in which are missing a number of functions. The OPRS Development Environment contains two relocatables, *'oprs-relocatable'* and *'xoprs-relocatable'* (and their C++ counterpart, *'c++-oprs-relocatable'* and *'c++-xoprs-relocatable'*) . These files are almost like *'oprs'* and *'xoprs'* except that a number of functions are missing.

Apart from some system libraries needed to build the final executable, the missing functions are:

```
declare_user_eval_funct
declare_user_eval_pred
declare_user_action
start_kernel_user_hook
end_kernel_user_hook
```

For the C++ version, the main is missing (so the user can have a C++ main) and is called oprs_main (and takes the same argument as main).

Somehow, you need to define these functions. Example of these functions are given in the *'pub_src'* directory, in the files: *'user-action.c'*, *'user-action.h'*, *'user-ev-function.c'*, *'user-ev-function.h'*, *'user-ev-predicate.c'*, *'user-ev-predicate.h'*, *'user-external.c'*, *'user-external.h'* and *'user-external_f.h'*.

When these files are defined, you need to write a Makefile more or less like the following one:

```
OPRS_DIR =

include ${OPRS_DIR}/site.make

OPRS_INCL_DIR = ${OPRS_DIR}/include

USER_SRC = user-evaluable-predicate.c user-action.c user-evaluable-function.c user-external.c

USER_OBJ = user-evaluable-predicate.o user-action.o user-evaluable-function.o user-external.o

SRCS = $(USER_SRC)

INCLUDE =  -I$(OPRS_INCL_DIR)

CFLAGS = $(ANSI) $(DEBUG_FLAG) $(OPTIMIZE) $(WARNINGS) $(MACHINE)
         $(CFLAGS1) $(INCLUDE) $(X_INCLUDE) $(PROFILE)

LDFLAGS = $(ANSI) $(STATIC) $(DEBUG_FLAG) $(OPTIMIZE) $(PROFILE)

SYS_LIB = -lm
```

```
all: oprs xoprs

oprs: $(OPRS_DIR)/oprs-relocatable $(USER_OBJ)
        $(CC) -o oprs $(USER_OBJ) oprs-relocatable

xoprs:  $(OPRS_DIR)/xoprs-relocatable $(USER_OBJ) $(LIB_UTIL)
        $(CC) -o xoprs xoprs-relocatable $(USER_OBJ) $(SYS_LIB) $(X_LIB)
```

## 21.12    Miscellaneous Questions

## 21.13    Common Mistakes

There are a number of mistakes which are commonly made. See [Known Problems and Things to Avoid], §M, page 391 for a list of potential pitfalls. Nevertheless, here is a list of common mistakes.

- Do not write OPs with evaluable predicate in their invocation part. They will not become applicable, because of this predicate becoming true.

- If you write a goal such as (!  (& (foo a $x) (boo b $y))), you may achieve it if this conjunction is true in the database, but if it is not, there are absolutely no way this specific goal may be achieved through a OP. Because only texpressions are allowed in Invocation Part as element of their gmexpression.

- When you declare a functional fact, do not forget the integer argument of the declaration.

- Before launching a X-OPRS Kernel or OPRS Kernel, make sure a OPRS-Server and a Message Passer are running.

# Chapter 22

# Simple OPRS Applications

## 22.1 Factorial Example

This fairly simple example introduces many different interesting concepts of OPRS. It uses logical and program variables, it introduces meta level reasoning and also presents some advanced features such as parallel goals intentions. This example can be found in *'data/fact-meta.inc'* and related files.

### 22.1.1 Factorial Example OPs

Here is the list of the procedures provided for this demo. They can be found in the file *'fact-meta.opf'*.

- **—Iterative Factorial—**

  A graphic OP.

  **Invocation:** `(! (FACTORIAL $N $RESULT))`

  **Context:** `()`

  **Effects:** `()`

  **Properties:** `((RECURSIVE NIL))`

  **Documentation:** `"This OP  computes the Factorial of $n and unifies the result with $result.`
         `It uses an iterative algorithm operating on local variables.`
       `Note the RECRUSIVE NIL property which will be used by the`
       `Meta level OP."`

- **—Meta Factorial test—**

  A graphic OP.

  **Invocation:** `(SSSOAK $X)`

**Context:** `((& (EQUAL (LENGTH $X) 2)`
`        (EQUAL`
`            (OP-INSTANCE-GOAL (FIRST $X))`
`            (OP-INSTANCE-GOAL (SECOND $X)))))`

**Effects:** `()`

**Properties:** `((DECISION-PROCEDURE T))`

**Documentation:** `"This Meta OP chooses randomly which Factorial OP to intend.`
`    Note that it is not used (because of its invocation part which`
`    will not trigger)."`

- **—Meta Factorial—**

  A graphic OP.

  **Invocation:** `(SOAK $X)`

  **Context:** `((& (EQUAL (LENGTH $X) 2)`
  `        (EQUAL`
  `            (OP-INSTANCE-GOAL (FIRST $X))`
  `            (OP-INSTANCE-GOAL (SECOND $X)))))`

  **Effects:** `()`

  **Properties:** `((DECISION-PROCEDURE T))`

  **Documentation:** `"This Meta OP chooses which Factorial OP to intend according`
  `    to the presence or not of the recursive property on the`
  `    applicable OPs.`
  `    Do not use this Meta OP in other applications."`

- **—Print Factorial—**

  A graphic OP.

  **Invocation:** `(! (PRINT-FACTORIAL $X))`

  **Context:** `()`

  **Effects:** `()`

  **Documentation:** `"This OP just looks for the factorial of $x and prints the result`

- **—Recursive Factorial—**

  A graphic OP.

  **Invocation:** `(! (FACTORIAL $N $RESULT))`

  **Context:** `()`

  **Effects:** `()`

  **Properties:** `((RECURSIVE T))`

> **Documentation:** "This OP computes the Factorial of $n in a recursive manner.
> Note the RECURSIVE T property which will be used by the
> Meta OP to decide which OP to intend."

- **—Test //—**

  A graphic OP.

  **Invocation:** (TEST_PAR)

  **Context:** ()

  **Effects:** ()

  **Documentation:** "This OP is here to illustrate the mechanism to intend in
  parallel a certain number of goals.
  In this case, the goal (! (print-factorial $x)) will be intended
  in // for all the values 4, 5, 6 and 7."

- **—Test Fact—**

  A graphic OP.

  **Invocation:** (FOO $X $Y)

  **Context:** ()

  **Effects:** ()

  **Documentation:** "This OP is here to demonstrate how one can call a goal
  invoked OP with a fact. Note that it will only print the
  result if $Y was bound to the right value... (you cannot
  conclude a fact with unbound variable)"

- **—Test and Set Fact—**

  A graphic OP.

  **Invocation:** (! (TAS-FACT $X $Y))

  **Context:** ()

  **Effects:** ()

  **Documentation:** "This OP is here to illustrate the TEST-AND-SET mechanism.
  It will post the goal (! (factorial $x $y)) and according to
  the success or the failure will print an appropriate message."

## 22.1.2   Other Factorial Example OPs

In this example, we use the new "if-then-else" node as well as the parallel thread
execution mechanism. These OPs can be found in the file *'fact-meta-if-par.opf'*.

- **—Iterative Factorial—**

  A graphic OP.

**Invocation:** `(! (FACTORIAL $N $RESULT))`

**Context:** `()`

**Properties:** `((RECURSIVE NIL))`

**Documentation:** `"This OP  computes the Factorial of $n and unifies the result wit`
`    It uses an iterative algorithm operating on local variables.`
`   Note the RECRUSIVE NIL property which will be used by the`
`   Meta level OP."`

- **—Meta Factorial Goal—**

  A graphic OP.

  **Invocation:** `(APPLICABLE-OPS-GOAL $GOAL $X)`

  **Context:** `((& (EQUAL (LENGTH $X) 2)`
  `        (== (GOAL-STATEMENT $GOAL)`
  `            (! (FACTORIAL $PAR1 $PAR2)))`
  `        (> $PAR1 10)))`

  **Properties:** `((DECISION-PROCEDURE T))`

  **Documentation:** `"This Meta OP chooses which Factorial OP to intend according`
  `   to the presence or not of the recursive property on the`
  `   applicable OPs.`
  `   Do not use this Meta OP in other applications."`

- **—Meta Factorial—**

  A graphic OP.

  **Invocation:** `(DONOT USE ME SOAK $X)`

  **Context:** `((& (EQUAL (LENGTH $X) 2)`
  `        (EQUAL`
  `            (OP-INSTANCE-GOAL (FIRST $X))`
  `            (OP-INSTANCE-GOAL (SECOND $X)))))`

  **Properties:** `((DECISION-PROCEDURE T))`

  **Documentation:** `"This Meta OP chooses which Factorial OP to intend according`
  `   to the presence or not of the recursive property on the`
  `   applicable OPs.`
  `   Do not use this Meta OP in other applications."`

- **—Print Factorial—**

  A graphic OP.

  **Invocation:** `(! (PRINT-FACTORIAL $X))`

  **Context:** `()`

  **Documentation:** `"This OP just looks for the factorial of $x and prints the result`

- —**Recursive Factorial**—

  A graphic OP.

  **Invocation:** `(! (FACTORIAL $N $RESULT))`

  **Context:** `()`

  **Properties:** `((RECURSIVE T))`

  **Documentation:** `"This OP computes the Factorial of $n in a recursive manner.`
  `    Note the RECURSIVE T property which will be used by the`
  `    Meta OP to decide which OP to intend."`

- —**Test Factorial**—

  A graphic OP.

  **Invocation:** `(! (TEST-FACTORIAL $X $N))`

  **Context:** `()`

  **Documentation:** `"This OP just looks for the factorial of $x and prints the result."`

# Chapter 23

# Complex OPRS Applications

## 23.1  Truck Loading Example

This example is a real application, with a simulator and a user interface.

### 23.1.1  Truck Loading Example Presentation

This example presents a supervision and control problem with various complex temporal constructions. As described on figure 23.1, an operator is in charge of a refilling station. The process to supervise is basically the following. From time to time, tank trucks come to the station. They are queued until a traffic light (Queue Light) turns green. As soon as they are in place, a "truck in place" signal is sent to the operator who has to turn the Queue Light to red again and waits until the truck is ready to be filled (Filling Talkback shows Empty). He then opens the valve controlling the product flow. This valve is opened and closed using a two position switch on the control panel (Switch Valve). To monitor the current position of the valve, two talkbacks with different sensors are used (Talkback Sensor1 and Talkback Sensor2). These talkbacks can display the following information: closed, open, and barber pole. Barber pole is displayed by a talkback whenever the valve is neither open nor closed but in between (most likely it is moving in position). Whenever the operator is filling up a truck and a "truck full" signal is received (the Filling Talkback shows Full), he immediately closes the valve. When the valve is considered as closed, he gives an "OK to go" to the truck driver, by turning the Filling Light to Green. This process goes on for ever during the whole day.

In the plan executed by the operator, different things can go wrong. The most important failure is the valve not closing properly; a slightly less important problem is the valve not opening properly. If the valve reportedly fails to open or close and, if there is a serious doubt about its real position, the operator
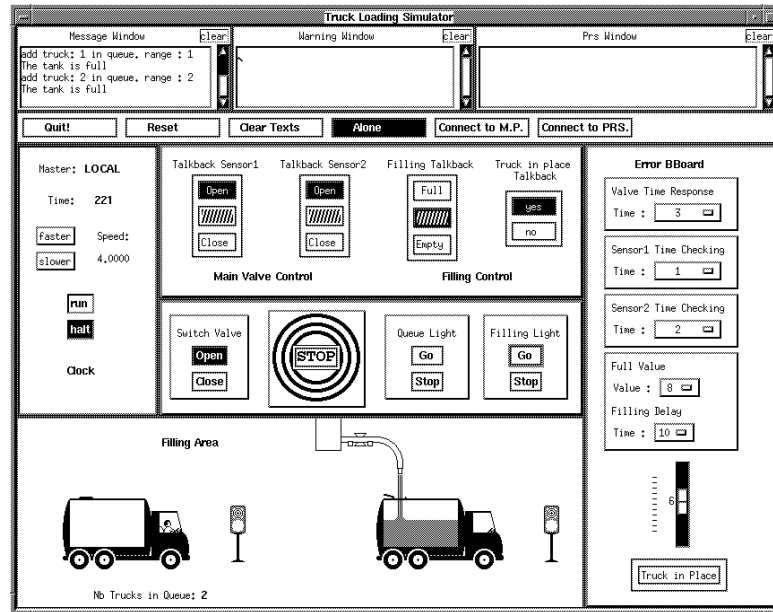
Figure 23.1: Truck Loading Demo

must activate the emergency shutdown (the Stop Button), which is considered as a "last resort" solution.

The inputs (and their respective values) of this control and supervision process are displayed in the Control Board (the blue panel on the top).

The actions the operator can take are available from the Command board (the bottom blue panel).

The operator does not see the bottom panel of the demo window which displays the real world (trucks showing up, filling up, etc). He can only rely on the Control board to analyze the current situation, and on the Command board to control the operations.

The three text windows in the upper part of the truck demo window display information from various sources. Message: normal operation of the demonstration. Warning: abnormal states in the simulation. OPRS: message sent and received from the controlling OPRS Kernel.

The six buttons below the text panes are used to (from left to right) quit the demo, reset the demo, clear the text panes, disconnect from the OPRS Kernel, connect to the Message Passer, and connect to OPRS Kernel.

The clock panel displays the current simulation time (from the simulator when alone, or from OPRS Kernel when connected). It is also used to start or stop the simulation time. There are also buttons to speed up or slow down the simulation in non connected mode.

The Error board on the right can be used to modify a number of internal

parameters of the demo. It is mainly used to create dysfunction.

In general, the operator should monitor the following conditions and take various actions in response to these conditions:

- Talkback 1 and Talkback 2 disagree (one says OPEN, the other says CLOSE). If this condition arises, the talkback which disagrees with the switch is considered as failed.

- Talkback 1 and Talkback 2 show a barber pole for more than x seconds (i.e., there are at least x seconds during which both talkbacks show barber pole). The valve is jammed, emergency stop.

- Talkback 1 or Talkback 2 show a barber pole for more than y seconds. If this condition arises, then the faulty Talkback is concluded as failed.

- z seconds after flipping the switch in one position or the other, the valve is not reported in the right position by both talkbacks. This interval is decreased down to w seconds if one of the talkback is failed. In this case, an emergency shutdown is issued.

### 23.1.2   How to Install the Truck Loading Demo

The *'demo/truck-demo'* directory should be placed or extracted in the *'oprs'* directory. Then just type `% make truck-demo`.

If everything goes well, you should end up with a `truck-demo` executable in the *'demo/truck-demo/bin'* sub directory, as well as a `oprs` and `xoprs` kernels in the *'demo/truck-demo/oprs'* sub directory. Note that you must use these kernels as they define their own evaluable functions and predicates (which are different from the one defined by default in the kernel).

This application is a good example of how to define your own actions, evaluable functions, predicates, etc...

### 23.1.3   How to Run the Truck Loading Demo

You first need to start the truck-demo. To do so, just execute the `truck-demo` program produced by the make command.

At this point, you can play with the simulator as if you were the operator. You are in "Alone" mode.

If you want to get OPRS to manage the "filling station", start a oprs-server, and then start a OPRS Kernel or X-OPRS Kernel with the following options (you must use the kernels from the *'demo/truck-demo/oprs'* directory as it defines some specific evaluable functions or predicates):   `% xoprs -n truck -x data/truck-demo.inc`

You then need to issue an `accept` command in the OPRS-Server.

Go back in the truck-demo window, and click on "Connect to the MP" button, wait until the connection completes (the button is highlighted). Click on the "Connect to OPRS" button. If you have been playing with the demo

before connecting to the OPRS Kernel, it is necessary to reset the simulation
(by clicking on the Reset button) before connecting to OPRS.

From this point, the OPRS Kernel will control and supervise the filling truck
process. You will see the lights and the valve switching upon OPRS request, as
well as OPRS reacting to transients and to real problems.

Feel free to select the X-OPRS Kernel window and trace the OP used for
this demo.

A priori, OPRS Kernel takes care of the filling station "ad vitam eternam".
If you want to produce an error (to check if OPRSwill find it), you can play
with the various settings of the Error board.

You can also improve the demo by adding OPs which will check other faulty
or error conditions. For instance, the *'demo/truck-demo/oprs/data/truck-demo-
plus.opf '* contains a OP which checks that you are not switching a traffic light
on (to green) while the valve is open. (Note that this OP is not loaded by default
but can be added with the Load OP command from the X-OPRS Kernel).

### 23.1.4   Truck Loading Example OPs

Here is a list of the procedures provided for this demo. They can be found in
the file *'truck-demo.opf '*.

- **—Day Plan—**

  A graphic OP.

  **Invocation:** (! (DAY-PLAN))

  **Context:** ()

  **Effects:** ()

  **Documentation:** "This OP is the top level OP of the
       system. It keeps loading truck until
       the end of the day."

- **—Load Truck—**

  A graphic OP.

  **Invocation:** (! (LOAD-TRUCK))

  **Context:** ()

  **Effects:** ((~> (TRUCK-READY))
       (~> (TRUCK-FULL)))

  **Documentation:** "This OP executes the whole plan to load
       a truck."

- **—Miscompare Talkback—**

  A graphic OP.

**Invocation:** (POSITION $T1 $POS1)

**Context:** ((? (& (ALARM YES)
        (TYPE TB $T1)
        (|| (ASSOCIATED-TB $T2 $T1)
           (ASSOCIATED-TB $T1 $T2))
        (STATUS $T1 GOOD)
        (STATUS $T2 GOOD)
        (POSITION $T2 $POS2)
        (|| (& (EQUAL $POS1 OPEN) (EQUAL $POS2 CLOSE))
           (& (EQUAL $POS1 CLOSE) (EQUAL $POS2 OPEN)))
        (POSITION SWITCH-VALVE $POSS))))

**Effects:** ()

**Documentation:** "This OP detects miscomparison in Talkback position.
    When the two talkbacks are opposed ( one open, the
    other close ) the one which is opposed to the switch
    is declared failed."

- **—Move Valve no alarm—**

  A graphic OP.

  **Invocation:** (! (POSITION VALVE $X))

  **Context:** ((? (ALARM NO)))

  **Effects:** ((=> (POSITION VALVE $X)))

  **Documentation:** "This OP tries to put the valve in position $x.
      It waits ($del-2tbs) time units if the two
      sensors are good, or ($del-1tb) time units
      if only one is good.
      After this time, it shutdowns if the trusted
      sensor(s) is not in good position. "

- **—Move Valve—**

  A graphic OP.

  **Invocation:** (! (POSITION VALVE $X))

  **Context:** ((? (& (ALARM YES)
          (DELAY TWO-GOOD $DEL-2TBS)
          (DELAY ONE-GOOD $DEL-1TB))))

  **Effects:** ((=> (POSITION VALVE $X)))

  **Documentation:** "This OP tries to put the valve in position $x.
      It waits ($del-2tbs) time units if the two
      sensors are good, or ($del-1tb) time units
      if only one is good.
      After this time, it shutdowns if the trusted
      sensor(s) is not in good position. "

- **—Reset Demo—**

  A graphic OP.

  **Invocation:** `(! (RESET))`

  **Context:** `()`

  **Effects:** `((=> (STATUS SENSOR1 GOOD))`
  `      (=> (STATUS SENSOR2 GOOD))`
  `      (~> (TRUCK-READY))`
  `      (~> (TRUCK-FULL))`
  `      (=> (TRUCK-PLACED NO))`
  `      (=> (POSITION VALVE CLOSE))`
  `      (=> (POSITION SENSOR2 CLOSE))`
  `      (=> (POSITION SENSOR1 CLOSE))`
  `      (=> (POSITION SWITCH-VALVE CLOSE))`
  `      (=> (POSITION QUEUE OFF))`
  `      (=> (POSITION FILLING OFF)))`

  **Documentation:** `"This op ever succeds, and will`
  `      reset all the used facts to the`
  `      initial value, in the effect part."`

- **—Reset and run truck loading—**

  A graphic OP.

  **Invocation:** `(RUN-DEMO)`

  **Context:** `((? (CLIENT $REQUESTER-NAME)))`

  **Effects:** `()`

  **Documentation:** `"This OP resets all the used facts to the`
  `      initial value; and then posts the goal`
  `      (!(day-plan)) for running the demo.`
  `      "`

- **—Shutdown—**

  A graphic OP.

  **Invocation:** `(! (SHUTDOWN $MESSAGE))`

  **Context:** `((? (CLIENT $REQUESTER-NAME)))`

  **Effects:** `()`

  **Documentation:** `"Send the message to shutdown the demo,`
  `      then an explanation.`
  `      Post the goal echec to fails the OP."`

- **—Talkback in BP—**

  A graphic OP.

**Invocation:** (POSITION $T BP)

**Context:** ((? (& (ALARM YES)
            (TYPE TB $T)
            (STATUS $T GOOD)
            (DELAY TB-FAIL $DEL-FAIL))))

**Effects:** ()

**Documentation:** "This OP detects if a Talkback stays
    too long in barberpole position."

- **—Talkbacks in BP—**

  A graphic OP.

  **Invocation:** (POSITION $T1 BP)

  **Context:** ((? (& (ALARM YES)
            (STATUS $T1 GOOD)
            (|| (ASSOCIATED-TB $T1 $T2)
                (ASSOCIATED-TB $T2 $T1))
            (STATUS $T2 GOOD)
            (POSITION $T2 BP)
            (DELAY TBS-BP $DEL-FAIL))))

  **Effects:** ()

  **Documentation:** "This OP detects the two Talkbacks in barberpole position
    for a too long time."

- **—Time Halt—**

  A graphic action OP.

  **Invocation:** (HALT-TIME)

  **Context:** ()

  **Effects:** ()

  **Action:** (HALT-OPRS-TIME)

  **Properties:** ((PRIORITY 1))

- **—Time Init—**

  A graphic action OP.

  **Invocation:** (! (INIT-TIME $TIME))

  **Context:** ()

  **Effects:** ()

  **Action:** (INIT-OPRS-TIME $TIME)

- —**Time Run**—

  A graphic action OP.

  **Invocation:** `(RUN-TIME)`

  **Context:** `()`

  **Effects:** `()`

  **Action:** `(RUN-OPRS-TIME)`

- —**Time Send**—

  A graphic OP.

  **Invocation:** `(GIVE-ME-TIME $REQUESTER-NAME $INITIAL-TIME)`

  **Context:** `((? (CLIENT $REQUESTER-NAME)))`

  **Effects:** `((~> (GIVE-ME-TIME $REQUESTER-NAME $INITIAL-TIME))`
  `     (~> (STOP-TIME $REQUESTER-NAME)))`

  **Properties:** `((PRIORITY 2))`

- —**Warning Messages**—

  A graphic OP.

  **Invocation:** `(! (WARNING $MESS))`

  **Context:** `((? (CLIENT $CLIENT)))`

  **Effects:** `()`

  **Documentation:** `"Send a warning message to the client."`

- —**client-accept**—

  A graphic OP.

  **Invocation:** `(INIT-DEMO $REQUESTER-NAME)`

  **Context:** `()`

  **Effects:** `()`

- —**client-leave**—

  A graphic OP.

  **Invocation:** `(GOOD-BYE $REQUESTER-NAME)`

  **Context:** `()`

  **Effects:** `((~> (CLIENT $REQUESTER-NAME))`
  `     (~> (STOP-TIME $REQUESTER-NAME)))`

  **Documentation:** `"This OP will kill all other intentions,`
  `     and clean up some facts."`

- —**switch light**—
  A graphic OP.

  **Invocation:** `(! (LIGHT $LIGHT_ID $STATUS))`
  **Context:** `((? (& (CLIENT $REQUESTER-NAME) (TYPE LIGHT $LIGHT_ID))))`
  **Effects:** `()`

- —**switch valve**—
  A graphic OP.

  **Invocation:** `(! (POSITION SWITCH-VALVE $STATUS))`
  **Context:** `((? (CLIENT $REQUESTER-NAME)))`
  **Effects:** `()`

- —**truck-sensor change**—
  A graphic OP.

  **Invocation:** `(TRUCK-IN-PLACE $VAL)`
  **Context:** `((? (TYPE PLACED-STATUS $VAL)))`
  **Effects:** `((~> (TRUCK-IN-PLACE $VAL)))`

One can also find an extra OP, in the file *'truck-demo-plus.opf'*.

- —**Light Change**—
  A graphic OP.

  **Invocation:** `(POSITION $L1 ON)`
  **Context:** `((AND (TYPE LIGHT $L1) (POSITION VALVE OPEN)))`
  **Effects:** `()`
  **Documentation:** `"This OP detects a traffic light switched on while the valve is still open."`

# Chapter 24

# Applications of OPRS

If there exist a priori no exclusive domain of application to Procedural Reasoning, some seem more adapted than other. They can be identified by the following characteristics, which are illustrated with different applications where Procedural Reasoning was applied:

**Control & Supervision of Complex Systems:** In this kind of application, one or several operators are in charge of executing and following, under some well defined conditions, the procedures the system designers established. These procedures cover the nominal mode of the system, as well as emergency and critical situation like alarms, or threshold overshooting. (Example: Telecommunication Network Supervision).

**Operator Assistance:** For various, reasons, replacing operators of complex systems with autonomous systems is not always recommended. However, operators tend to make mistakes, with potentially dramatic consequences, whenever they are overflowed with storms of alarms and data. For this very reason, it is very important to assist them in the most tedious tasks, or in those requiring quick responses, like in power plant supervision, or in threat assessment systems.

**Automation of predefined procedures execution:** In many domains, complete folders of procedures have been written to describe all the reachable states of the system (planes, space shuttle). Here also, operators assure the triggering, retrieval, and execution of these procedures.

**On-line planning and execution control:** These activities combine the planning of task and actions, and the control of their execution. By planning, we mean that the system chooses at run-time a solution among a set of possibilities, and this according to some general criteria. Execution control checks that the chosen plan or procedure is executing properly (example: execution control of a mobile robot, pilot associate).

**Diagnosis & Troubleshooting:** The decision trees used in diagnosis activities show a procedural structure. Thus, troubleshooting is usually done by following the procedures which, test after test, determine the system faulty component (examples: troubleshooting of radar system). Note that these tests can be arbitrary complex and require the execution of other procedures.

**Operator Training:** Given its graphical capabilities, procedural reasoning can also be used for operator training. As a matter of fact, problems can be simulated for the operator to follow the procedures the system is currently running (example: operator training in the space shuttle mission control room).

In every domain presenting these characteristics or properties, procedural reasoning is the solution that improves system safety, speed up response time and reduces operating costs.

# Chapter 25

# Optimizing a OPRS Applications

## 25.1 Optimizing Hashtables

A number of commands (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39) or menus (see [Stat All Hashtables], §13.3.3, page 193, and others) are available to obtain statistics on the use of the various hashtables of the kernel.

When you select these commands, you get the following printout (this one was obtained with a `stat all` command):

```
 The id hashtable contains:
     442 element(s)
     in  355 buckets ( 1024 );
     with a maximal number of 4 element(s) in one bucket.
The database hashtable contains:
     48 element(s)
     in  48 buckets ( 1024 );
     with a maximal number of 1 element(s) in one bucket.
The predicate hashtable contains:
     107 element(s)
     in  51 buckets ( 64 );
     with a maximal number of 5 element(s) in one bucket.
The function hashtable contains:
     146 element(s)
     in  57 buckets ( 64 );
     with a maximal number of 6 element(s) in one bucket.
```

You can then analyze the result and decide if a specific hashtables is overloaded or not. You may then decide to change its size (reduce it or increase it) using the argument of the `oprs` or `xoprs` command (which are the same for this matter) (see [Arguments to the oprs Command], §1.2, page 22).

Note that these statistic are "static" in the sense that they give you snapshots of the use of your hashtables. However, most of them (except the database and id hashtable) are loaded upon starting the kernel and do not evolve much during the execution. Nevertheless, for the database hashtable, you should stat it at different time of the execution to figure out its maximum size and conclude then which size is the most appropriate. As for the id hashtable, or the symbol hashtable, keep in mind that this hashtable cannot be cleared by any mean. As a result, if you keep adding new symbols in the kernel, then this will be an ever growing hashtable.

Last, you may consider reducing the size of the hashtables (after all , the default values may far exceed your need). This will have the advantage of reducing the size of the kernel, by the amount of memory saved.

Keep in mind that since release 1.1, all these hashtable are mostly used at compile time. The id, the function and the predicate hashtables are use at runtime just to parse coming external events, and user commands (which explain why they are kept).

## 25.2   Just the Right Level of Meta Level

This is probably one of the easiest optimization one can do... and if it has not been done, it can be one of the most profitable optimization.

You should always use the right level of options in the kernel to suit your meta level needs. For example, if your kernel does not use any meta level reasoning, you should disable it in the kernel (see [Meta Level Reasoning], §9, page 131) with a `set meta off` command or with the option menu of the X-OPRS Kernel. However, if it does, for example use the `SOAK` meta fact, you should enable it and enable the conclusion of this particular fact (and disable all the other one).

## 25.3   Database Organization

Database optimization can be of different type. First you should optimize the size of its hashtable (not too big, not too small), see [Optimizing Hashtables], §25.1, page 285.

## 25.4   Slicing your Action

One of the important point to consider when optimizing an application is the time taken by actions. If their execution takes too long time, you should consider recoding them to slice their execution in smaller time chunk. This can be done using the action slicing mechanism (see [Action Slicing], §10.11, page 143).

See `action_first_call` and `action_number_called`, [Intention Manipulation Functions], §G.1.8, page 344.

# Part X

# Appendices

# Appendix A

# Principal Differences Between C-PRS and OPRS

# Appendix B

# Principal Differences with SRI PRS

One of the principal concern while writing C-PRS and OPRS was to improve some of the known problems in SRI Lisp PRS. As a consequence, there are minor differences between the two versions.

1. The syntax is far more rigorous in OPRS. As a result, a number of expressions perfectly correct in the Lisp PRS are not recognized by OPRS. For example, the following goal is not allowed: `(! (SEND-COMMAND-ACTION (DISPLAY-FOO (((CURRENT $O) (CURRENT $B))))))` because `(((CURRENT $O) (CURRENT $B)))` is not a valid Composed Term. Similarly, this goal: `( ! (& ( > (FOOBAR $P $A)) (=> (FOOBAR $P $A)))))))` is not allowed any longer... It should be replaced with: `(& ( > (FOOBAR $P $A)) (=> (FOOBAR $P $A))))))`

2. The following syntax is still allowed: `( ! (=> (INITIAL-VALUE $P $O $D $V)))` but should be replaced with the `=>` temporal operator (a similar remark holds for `>`) .

3. The `@` variables have the expected behavior. In SRI PRS, the only way to rebind a variable was to use a goal such as `(! (= @x ...))`. This limitation has been removed in OPRS, and the `@` variables can be rebound in any context.

4. The database uses a term indexing mechanism which has been extended recursively (SRI PRS version did not handle embedded composed terms properly).

5. The wait temporal operator is cleaner in OPRS as it succeeds only when it has been achieved... In SRI PRS, it always succeeded (but it sleep until the condition was satisfied).

6. The database file has not quite the same format. You need to give a list of facts, not just the facts in a file. To convert your already existing files, just add an open parenthesis at the beginning, and a closing one at the end, and remove all the declarations of functional facts, basic events, and so on.

7. OPRS does not support the negation as a failure, nobody ever used it anyway.

8. Closed world predicates, evaluable functions and evaluable predicates are "local" to a OPRS agent which is much better.

9. SOAK and other meta facts are now basic event (see [Basic Events], §5.7, page 88).

10. OPRS has no support for bignum... As a general rule, extreme caution should be exercised regarding the availability of Lisp specific functions or features. We do not plan to rewrite a complete Lisp interpreter in OPRS... (this would defeat the main argument for writing OPRS).

11. OPRS checks predicates, functions and symbols when you compile OPs.

12. Basic events are just declared with their predicates, not the predicates and the number of arguments, since we considered this feature useless.

13. The achiever field in OPs is not supported anymore. If it exists, it does not even shows up in the OP Editor or in the X-OPRS Kernel, but a warning is issued. Moreover, its value is not taken into account by the kernel (here also, this feature has hardly been used... so it was not worth putting in the C version.)

14. Lisp lists are not "standard" objects: you need to specify them with a different reader syntax (. .) to distinguish them from composed terms.

15. The default OPs are not loaded by default in the OPRS Kernel. In fact the notion of default OPs is a little weak, and we consider that it is up to the end-user to decide which OPs are required in the kernel. Of course, we still provide a number of OP the user can load in its kernel(see [Default OPs], §F, page 315).

16. The `APPLICABLE-OPS-FACT` and the `APPLICABLE-OPS-GOAL` facts have different arguments in OPRS Kernel. Their syntax is: `(APPLICABLE-OPS-FACT fact list-of-op-instances)` and `(APPLICABLE-OPS-GOAL goal list-of-op-instances)`. In SRI PRS it was: `(APPLICABLE-OPS-FACT list-of-op-instances)` and `(APPLICABLE-OPS-GOAL list-of-op-instances)`.

17. By default, the OPRS Kernel always evaluates what is evaluable in a goal at posting time, unless the evaluable function is quoted (see [Current and Quote], §10.7, page 141). This greatly simplify the syntax of the OPs as one do not need to put `current` all over the place... Note that there

is a flag in case you prefer the old form, see [OPRS Kernel Run Option Commands], §2.6, page 32 for more details.

18. The `*FACT` and `*GOAL` marker in the general meta expression are not recognized in recent version of OPRS. Version of the OP-Editor prior to 1.3 will recognize them and will thus convert your old OP to the new format.

# Appendix C

# Principal Differences Between Subsequent Versions of C-PRS

## C.1 Changes Between Version 1.0 and Version 1.1

There are a number of small changes which can be of interest to the user. For more information, consult the *'NEWS'* file int the C-PRS distribution.

- Change the string pattern to allow " in it. We are now using the C string syntax... But for now, just the
  ", such as in `"foo bar`
  `"asd asd`
  `""`. Internally, strings now do not have double quote around as they used to. Now they are just C strings.

- Added a printf function, more like the C one. Example: `(printf (format "The %d of %s is %f." $x $y $z))`

- Added a list of symbols in the OP Editor and in the OPF format.

- The kernel is now able to read pointer values.

- Added a new way of scheduling intention. Instead of giving a sorting predicate, it is given the list of runnable intentions, and it can be ordered as the user want. This make it easier to make time sharing stuff for example.

- Added an extern `x_oprs_top_level_widget` for the user to create its own widget tree in X-OPRS Kernel.

- Defined the `OPRS_DATA_PATH` environment variable and the corresponding `-d` argument. This is a list of directories where C-PRS will look for data files.

- Defined `OPRS_MP_PORT`, `OPRS_SERVER_PORT`, `OPRS_SERVER_HOST` and `OPRS_MP_HOST` environment variables, and changed the various programs accordingly to take them into account. (command arguments have a higher priority than environment variables).

- Defined a `OPRS_DOC_DIR` environment variable to point to the documentation directory.

- Implementation of `INT_ARRAY` and `FLOAT_ARRAY`. Added [ and ] as reader characters to recognize arrays. The type of the array is determined from the first element. Subsequent elements are casted appropriately.

- Implementation of `U_POINTER` user pointer to user defined objects.

## C.2   Changes Between Version 1.1 and Version 1.2

There are a number of important changes between version 1.1 and version 1.2. Only the important one are listed in the following sections.

### C.2.1   Changes in the Commands Syntax of the OPRS Kernel

To clarify it and to make it more easy to use, the command set of the OPRS Kernel has been greatly improved and unified.

Here is a table of correspondence between the 1.1 command and their new syntax in 1.2 and above. The commands which are not referenced have not changed.

A shell script *'update-inc-file'* in the *'util'* directory of the distribution, is provided to allow the user to automatically translate its *'.inc'* and *'.sym'* files from versions previous to 1.1 to version 1.2.

### C.2.2   Miscellaneous Changes Between Version 1.1 and Version 1.2

There are a number of small changes which can be of interest to the user. For more information, consult the *'NEWS'* file in the C-PRS distribution.

- Added a new option to the OPRS Kernel and to the X-OPRS Kernel to start without registering to the OPRS-Server (`-a`).

- Added a new command in the OP Editor (relevant OP) to find the OP relevant for a particular edge.

| 1.0 | 1.1.1 | Section |
|---|---|---|
| show_db | show db | 2.2 |
| save_db | save db | 2.2 |
| empty_db | empty fact db | 2.2 |
| declare_cwp | declare cwp | 2.9 |
| declare_be | declare be | 2.9 |
| undeclare_be | undeclare be | 2.2 |
| declare_ff | declare ff | 2.9 |
| load_db | load db | 2.2 |
| delete_op | delete op | 2.3 |
| unload_opf | delete opf | 2.3 |
| print_op | show op | 2.3 |
| traceg_op | trace graphic op | 2.3 |
| tracet_op | trace text op | 2.3 |
| traceg_opf | trace graphic opf | 2.3 |
| tracet_opf | trace text opf | 2.3 |
| list_ops | list op | 2.3 |
| list_opfs | list opf | 2.3 |
| compile_ops | load opf | 2.3 |
| empty_op | empty op db | 2.4 |
| trace rop | trace relevant op | 2.5 |
| trace opc | trace load op | 2.5 |
| trace soak | trace applicable op | 2.5 |
| trace fact | trace fact | 2.5 |
| trace goal | trace goal | 2.5 |
| trace db_frame | trace db frame | 2.5 |
| set fact_inv | set meta fact | 2.7 |
| set goal_inv | set meta goal | 2.7 |
| set app_ops_fact | set meta fact op | 2.7 |
| set app_ops_goal | set meta goal op | 2.7 |
| set eval_on_post | set eval post | 2.6 |
| set par_post | set parallel post | 2.6 |
| set par_intend | set parallel intend | 2.6 |
| consult_rop | consult relevant op | 2.13 |
| consult_aop | consult applicable op | 2.13 |
| show_sleep_int | show intention | 2.13 |
| show_mem | show memory | 2.13 |
| declare | declare id | 2.9 |
| unifie | unify | 2.13 |
| stat_db | stat db | 2.13 |
| stat_op | stat op | 2.13 |
| stat_id | stat id | 2.13 |
| stat_all | stat all | 2.13 |
| reset_kernel | reset kernel | 2.13 |

Table C.1: Commands Equivalence Between Version 1.1 and 1.2

- Added an option to the OP Editor to convert op files to the newest format `op-editor -c <file(s)>`.

- Time stamping is now controlled by a flag (it is too expensive to keep it all the time).

- Wrote a default op to broadcast a message. Wrote a broadcast-message action. Added a broadcast mode to the message passer. I separated the modules which can receive and the one which can send (the OP Editor and the OPRS-Server do not receive). Therefore, the Message Passer never sends anything to the OPRS-Server and the OP Editor.

- Added a `val` evaluable function which returns the value on which variable points... In fact it is the identity function, but it can be very useful to force getting the value of program variables.

- Added a number of new commands and associated tokens: `list cwp`, `list be`, `list ff`, `list ep`, `list ef`, `list action` and `list all`.

- Wrote a `show intention` command (also available as a menu item) which gives an extensive status of the intentions and their component (thread, status, waiting condition, joining, etc).

- Added code to execute all the root of the intention graph in //. This is controlled with a flag (`set parallel intention on/off`) and in the option menu. Default value of the flag is on.

- Added op predicate, i.e. predicate which can only be satisfied by OP not in the database.

- Now the Message Passer is started automatically by the application which want to connect to it. In other words, if a connection failed because nobody is listening on the host/port, then a Message Passer is started on this host/port. We use `rsh` to start the mp on a remote host, and fork to start it on the same host.

- Added a new program `kill-mp` to kill the Message Passer. It registers to the Message Passer and kill it.

- To prevent conflict with types defined in VxWorks system includes... the type `LIST` and `NODE` have been renamed `OPRS_LIST` and `OPRS_NODE`.

- Change the registration to the Message Passer mechanism to pass the registration protocol as argument... and to return a status to deny/allow the registration. This change makes the registration mechanism incompatible with the previous version.

# C.3 Changes Between Version 1.2 and Version 1.3

There are a number of important changes between version 1.2 and version 1.3. The most important change between version 1.2 and 1.3 is the text OPs and is described in the See [Text OPs], §4.3.4, page 68 section.

## C.3.1 Miscellaneous Changes Between Version 1.2 and Version 1.3

There are a number of small changes which can be of interest to the user. For more information, consult the *'NEWS'* file in the C-PRS distribution.

- Added CONS-TAIL and LAST evaluable function.

- ope-graphic.c (scroll_bars_moved): Wrote a function to get the horizontal and vertical motion at the same time (to avoid this the awful steppy look of the redrawing).

- mp-oprs.c: added a -l "file_name" arg to specify an log file (in which mp will log all the message which went thru it).

- xp-op-graphic.c xp-rop.c: Extend the relevant OP mechanism, to the invocation and context parts in the X-OPRS Kernel and in the OP Editor.

- Added a "stepped" and "all" button in the Trace OP dialog box.

- Added the next, step, halt and run commands in the oprs kernel.

- Added a "show run status" command in the OPRS Kernel.

- The step by step execution of OP is now controled by a separate flag (not the graphic traced one anymore). The commands "trace step op ... on/off" et "trace step opf ... on/off" have been added.

- Added == as evaluable predicate to allow fancy construction such as (? (V (== (..) (..)) (== (...) (...)))).

- Added a scheduler for parallel intententions.

- Added evaluable functions to return user, system and user+system clock tick: USER-SYS-CLOCK-TICK, USER-CLOCK-TICK, SYS-CLOCK-TICK. These do not work under VxWorks.

- Added a "transmit_all" command in the oprs-server to transmit the same command to all connected kernel.

- Added an option to have symbol in lower case or any case. The default is still upper case. The authorized options are "upper", "lower" or "none". You may specify this option by setting the OPRS_ID_CASE environment variable: setenv OPRS_ID_CASE none  or in the command line with the '-l' option: xoprs -n Foo -l none -x ... The command line has precedence on the envionment variable. This option is avaible for the OPRS-Server, the PK, and the OP Editor.

- Change the command `list op predicate` in `list op_predicate` and `declare op predicate` in `declare op_predicate`.

- Report the expr being concluded when an attempt is made to conclude such expr containing a variable.

- Added a LISP_LIST intersection and a LISP_LIST union functions.

- `mp_socket` has default value of -1 (to detect unitialization for example).

- Added a memq_ep evaluable predicate MEMQ.

- `start_kernel_user_hook` is now executed before the initial_command (i.e. load with the -x).

- The name of the opfile appears in all op lists presented to the user.

- xp-dialog.c: Added the Conclude from Parser trace in X-OPRS Kernel.

- Added a `find_atom` function which does like `make_atom` but warn you if the symbol had not been declared previously.

- Many prompt dialog box have been changed to provide an history of the previously entered commands.

- Added a `build_string` function to build a Term from a string, making a copy of the string. This is the function external users should use.

- Added a `show oprs_data_path` and a `set oprs_data_path <string>` command.

- Added a `trace conclude` command to trace conclude operation performed by the user.

- The X-OPRS Kernel menus have been reorganized.

- Added a `Display Next OP` and `Display Previous OP` in X-OPRS Kernel.

- The Message Passer registering functions have been changed. This change make them incompatible with previous versions of the Message Passer.

- In the various parsers, we have added some code to check we are not opening a directory instead of a file.

- Added extern "C" declaration in all the public include files which may be used by a c++ compiler.

- Added a verbose argument `-v` to the Message Passer to trace all messages.

- An error handler for external code errors is now provided.

- One can log in a file all the output made in the X-OPRS text window using the `-log` argument to X-OPRS.

- Now we can have the text trace associated to a particular intention in its own window... To create this particular window, right click on the intention in the intention graphic pane. Middle click gives an overview of the intention current status.

- The database is sorted before being printed.

- Some trace messages have been shortened to make the trace less verbose.

- Added `build_nil` and `build_t` which return a Term with t or nil as atom. Useful for evaluable functions and actions.

- Reorganised the option menu in X-OPRS. We now have three different option dialog boxes (Run, Compiler/Parser and Meta Level).

- Implementation of a = ev-predicate... It works... it is faster... and it allows setting variable in invocation part parsing. (requested by rachid@laas.fr).

- Added a Show Global Variable menu and a "show variable" command to print the global variables list.

- The Message Passer kills itself after an hour with no new connection and when nobody is registered.

- Added a show memory menu.

- Added a `delete fact` and `conclude fact` command in the X-OPRS Kernel.

- Memory consumption has been improved.

- The list of variables do not appear anymore in the frame bindings.

- Added a `require` command which does like include except that it check this file has not been already loaded with another `require` command (see [OPRS Kernel Loading Commands], §2.4, page 30).

- When broadcasting, do not send the message to the sender.

- Added two new flags to control predicate and function declaration. They can be changed with `set predicate|function on|off` (see [OPRS Kernel Compiler/Parser Option Commands], §2.8, page 34).

- Added two new commands: `declare predicate` and `declare function` commands (see [OPRS Kernel Declaration Commands], §2.9, page 35).

- Added a delete fact (see [Delete Fact Database], §13.3.2, page 185) and a conclude fact (see [Conclude Fact Database], §13.3.2, page 185) command in X-OPRS Kernel.

- There is a semaphore OP library available (see [semaphore.opf], §F.4, page 335).

- the Write TeX Doc File command is now public (see [Write TeX Doc File], §16.1.1, page 224).

- ope/Parser.l: Dropped the *GOAL and *FACT syntax... If you still have OP using this syntax, use the 1.2 OP editor to convert your OP file.

- Modified the parser to count lines in files and in strings so we can report where are the errors more accurately.

- Kernels and op-editor can now print the temporal operators in english... This is controlled by a flag which can be set at startup time with the "-peo" (PrintEnglishOperator) option for xoprs and op-editor and "-p" option for oprs. This can also be specified as a ressource named *print-EnglishOperator (True or False). Default value is always false... and the OP written in files are written with the short version. Clean up all the TEST symbol in the various test file... (remember that TEST is now a reserved word).

- XOprs.ad: Added a resource setting to enable/disable word wraparound in the textWindow.

- All grammar share the same basic files. So the lexical grammar is now consistent other the 4 modules using it (oprs, xoprs,op-editor and oprs-server).

- Removed support for grasper graph and sun graph.

- Created accelators in the op editor to quickly access all the edit modes. This is user customizable with the default file. By default, they are accelerated with one letter (which is present as a mnemonic on the text of the button). No modifier are used (meta,control, nor shift).

- The id cannot start with a : character. (reserved for keywords in text op).

- data/new-default.opf: Removed test and test-and-set which are now obsolete and conflict with the test symbol used as a temporal operator.

- ev-predicate.c: defined some new predicates: NUMBERP, CONSP, STRINGP, and ATOMP.

- mp-register.c and send-message.c are using the standard malloc...

- Grammars allows temporal operator in clear (achieve, test, preserve, maintain, wait, conclude and retract).

- The function `make_and_declare_action`, `make_and_declare_eval_func` and `make_and_declare_eval_pred` do not take a hash table argument anymore. Please, modify your code accordingly.

- macro-pub.h and macro.h (MALLOC): is now defined in macro-pub for the final user to allocate object for the kernel.

- Added OPRS_DATA_PATH support under VxWorks.

- ope-save.c (write_opfile_header): Added the version status in the OP file header.

- XOprs.ad. The linotype helvetica looks ugly... I had to force the adobe one in the font set.

## C.4  Changes Between Version 1.3 and Version 1.4

### C.4.1  Main Changes Between Version 1.3 and Version 1.4

There are a number of important changes between version 1.3 and version 1.4. The most important changes between version 1.3 and 1.4 are:

- The possibility to dump various internal data (OPs and database for now), and the various dump commands. Note that the dump format is architecture independant. Note also, that the garphical information may be dumped or not, according to the kernel from which you dump the datas. While loading OP, if a OP with a name and a filename is already loaded, the newly loaded OP is ignored. See [OPRS Kernel Dumping/Loading Commands], §2.11, page 36 for more information on this.

- A better support for the VxWorks version (which now supports the Message Passer and the `kill-mp` programs). See [VxWorks], §D.1, page 307 for more information on this.

- Support for multiple languages (French and English for now) and for iso-latin characters set.

- The non graphical version of the various OPRS Development Environment programs have been ported and work under Windows 95 and Windows NT. See [Windows95-NT], §D.4, page 309 for more information on this.

- No more `LISP_CAR`. `LISP_LIST` now contains Terms, and Terms can now be Intention, Fact, Goal and Op_Instance. They are not readable though. One can remove all the `term_to_car` and `car_to_term` fuctions, these are obsolete and NO-OP now.

- Introduction of a `contrib` directory in the distribution which contains code contributed by various people but which is not part of the OPRS distribution.

## C.4.2 Miscellaneous Changes Between Version 1.3 and Version 1.4

There are a number of small changes which can be of interest to the user. For more information, consult the *'NEWS'* file in the C-PRS distribution.

- Added a `-x` option to the Message Passer to specify that new connection with already registered name should lead to the disconnection of the older client.

- Node name are printed (in graphic) without the vertical bar.

- Replace the `LEXPRESSION` term type with the `GEXPRESSION` as they are easier to manipulate...

- When printing the value of a variable, still print the variable name if no value was found/bound.

- Allow id starting with : (like keywords).

- Added evaluable functions `mention` and `all-pos`.

- Added macro to warn the user of the use of obsolete user functions.

- Added functions to allow the posting of fact from user call function. Under VxWorks, this even can been done from another process (the posting is then protected with a semaphore to ensure mutual exclusion). See [Fact Posting Functions], §G.1.7, page 342 for more information on this.

- oprs-type-pub.h: Due to some stupid type definition in Microsoft Windows, we had to replace the type FLOAT with TT_FLOAT and ATOM with TT_ATOM.

- Big clean up in the VxWorks version to free all the memory used when one exit/kill a kernel. This way, one can restart another oprs on the same board, without loosing any memory allocation. In fact, this mechanism is used on all kernels, which explain why it takes sometime a long time for a kernel to exit. However, this mechanism is a good leak detector.

- Check all conditions after loading a database.

- Two different commands to print graphics OP and text OPs.

- The OP printed in the file are now pretty printed.

- Do not allow id starting with a + or -

- Just one "

  ```
  n.
  ```

  n" should be necessary to reset the OPRS-Server parser.

- ASSERT is now equivalent to CONCLUDE, and it prints ASSERT instead of CONCLUDE.

- Added an intention failure trace which reports when an intention failed, with the stack of goals which lead to the failure.

- XOprs.adlang: Language dependent ressource file.

# Appendix D

# Hardware and Software Dependancies

Although OPRS is portable and runs on a large number of platforms, there exist some differences which are presented in this chapter.

## D.1   VxWorks

The VxWorks version of OPRS has a lot of particulariries. Although there are no main programs under VxWorks, we distinguish between task spawnable code and library code (i.e. which may be used by more than one processes). Files with a '.o' suffix are considered as library and can be used by more than one processes. Files without any suffix usually contain a `main` which can be called with `taskSpawn` or from the shell.

The VxWorks distribution is composed of three programs and two libaries:

**'c_toolkit.o'** This library is used by the *'vxoprs'* and *'vx-mp-oprs'* components. It can be shared by more than one programs. The proper variable are taskVar'ed.

**'vx-mp-lib.o'** This library is used by any application which want to connect to the message passer. It can be shared by more than one programs. The proper variable are taskVar'ed. *'vxoprs'* require this library.

**'vxoprs'** This is the default VxWorks OPRS Kernel. It requires *'c_toolkit.o'* and *'vx-mp-lib.o'* to be loaded on the same board. The entry point is `oprs_main` (see below).

**'vxoprs-relocatable'** This is the default VxWorks OPRS Kernel minus the entry points to allow the user to link its own code. It requires *'c_toolkit.o'* and *'vx-mp-lib.o'* to be loaded on the same board. The entry point is `oprs_main` (see below). See [Linking C Code in the Kernels], §21.11, page 265 for more on this.

**'vx-mp-oprs'** This is the VxWorks Message Passer program. It requires *'c_toolkit.o'* to be loaded on the same card. The entry point is `mp_oprs_main` (see below).

**'vx-kill-mp'** This is the VxWorks `kill-mp` program. It requires *'vx-mp-lib.o'* to be loaded on the same card. The entry point is `kill_mp_main` (see below).

In VxWorks, there is no `main` in the code. The OPRS main function has therefore been renamed and is called `oprs_main` and has the following prototype:

```
int oprs_main(char *name_arg, char *server_hostname_arg,
              int server_port_arg, char *mp_hostname_arg,
              int mp_port_arg, char *include_filename_arg)
```

The arguments are explained below:

`name_arg` is the name of the OPRS Kernel.

`server_hostname_arg` is the hostname on which the OPRS-Server is running.

`server_port_arg` is the port on which the OPRS-Server is listening. If 0, no connection is made to the OPRS-Server.

`mp_hostname_arg` is the hostname on which the Message Passer is running.

`mp_port_arg` is the port on which the Message Passer is listenning. If 0, then no connection is made to the Message Passer.

`include_filename_arg` is an include file name to load upon starting. If NULL, or empty string no file is loaded.

The OPRS Kernel cannot be run more than once on the same board (VxWorks processes share the same symbol space). However, the *'vx-mp-lib.o'* library is shareable. More than one program can use it on the same board. The proper variables have been put in VxWorks `taskVar`.

Most environment variables are ignored under VxWorks, except for the `OPRS_DATA_PATH` variable.

The Message Passer main function has been renamed and is called `mp_oprs_main` and has the following prototype:

```
int mp_oprs_main(int mp_port_arg, int verbose_arg, char *mp_log_filename_arg,
int exclude_arg)
```

The arguments are explained below:

`mp_port_arg` is the port on which the Message Passer is listening.

`verbose_arg` is the flag which says if yes or no the Message Passer should be verbose on the messages exchanged.

**mp_log_filename_arg** is file to which Message Passer will log the messages exchanged (if NULL or an empty string is given, no log are done).

**exclude_arg** is the exclude flag which says if the Message Passer should, upon a newer registration, disconnect the former client with the same name.

The **kill-mp** main function has been renamed and is called **kill_mp_main** and has the following prototype:

```
int kill_mp_main(int mp_port_arg)
```

The argument is explained below:

**mp_port_arg** is the port on which the Message Passer you want to kill is listenning.

Note that all OPRS Kernel should be started (using **taskSpawn**) with the **VX_FP_TASK** set. Some operations on floating point in the kernel require to have this option set up, as to enable the process context switch to save the FPU registers.

When properly exiting, the OPRS Kernel and the other OPRS Development Environment program return the memory allocated. However, this memory does not show as freed in the standard **memShow**, as OPRS Kernel uses its own allocation mechanism and will keep its memory allocated for future use. Use the **show memory** command of the OPRS Kernel (see [OPRS Kernel Miscellaneous Commands], §2.13, page 39) to see what it keeps for its own use.

## D.2  C++ Relocatables

The relocatables are given in two format, the standard relocatable, and one which can be linked to some C++ functions, in which case the **main** is not defined to allow a C++ **main** to be used (the C++ main performs some initialization required by C++ functions). The main is then named:

```
int oprs_main(int argc, char **argv, char ** envp);
```

## D.3  SparcStation

There exist a multi-thread version of the OPRS-Server available under Solaris 2.4 which do auto accepting of new OPRS Kernel clients.

## D.4  Windows95-NT

Under Windows 95... The **Term_Type** symbol **FLOAT** and **ATOM** are already defined. Therefore they are renamed **TT_FLOAT** and **TT_ATOM**.

The various program are statically linked.

Due to the lack of signal timer under Windows, the conditions echanism is currently more CPU consuming than under Unix.

Only the non graphical program have been ported.

# Appendix E

# Commands Equivalence between the OPRS Kernel and the X-OPRS Kernel

Most commands are available in both kernel (the X-OPRS Kernel and the CPK). The following tables will allow the user to find out which command correspond to which menu and vice verca. Note that some commands are not available in both interface. This is the case for most declaration commands (`declare be`, `declare ff`, etc) which one usually put in an include file. In any case, you can always transmit a command from the OPRS-Server (providing the kernel has been properly connected to the OPRS-Server upon startup).

| Command Name | Section | Menu Item | Section |
|---|---|---|---|
| `add goal\|fact` | 2.13 | Add Fact or Goal | 13.3.2 |
| `conclude expression` | 2.2 | None | |
| `consult gexpression` | 2.2 | Consult Fact Database | 13.3.3 |
| `consult applicable op goal\|fact` | 2.3 | Consult Applicable OP | 13.3.3 |
| `consult relevant op goal\|fact` | 2.3 | Consult Relevant OP | 13.3.3 |
| `declare be predicate` | 2.9 | None | |
| `declare cwp predicate` | 2.9 | None | |
| `declare ff predicate integer` | 2.9 | None | |
| `declare function function` | 2.9 | None | |
| `declare id symbol` | 2.13 | None | |
| `declare op_predicate predicate` | 2.9 | None | |
| `declare predicate predicate` | 2.9 | None | |
| `delete expression` | 2.2 | None | |
| `delete op op_name` | 2.3 | Delete a particular OP | 13.3.2 |
| `delete opf file_name` | 2.3 | Unload OP File... | 13.3.1 |
| `disconnect` | 2.13 | None | |
| `echo (g\|gt\|gm)expression` | 2.13 | None | |
| `empty fact db` | 2.2 | Empty Fact Database | 13.3.2 |
| `empty op db` | 2.4 | Empty OP Library | 13.3.2 |
| `help\|h\|?` | 2.13 | Help | 13.3.7 |
| `include file_name` | 2.4 | Include... | 13.3.1 |

Table E.1: Commands Equivalence Between the Kernels (First Part)

| Command Name | Section | Menu Item | Section |
|---|---|---|---|
| `list action` | 2.10 | List Action | 13.3.3 |
| `list all` | 2.10 | List All | 13.3.3 |
| `list be` | 2.10 | List Basic Event Predicate | 13.3.3 |
| `list cwp` | 2.10 | List Closed World Predicate | 13.3.3 |
| `list evaluable function` | 2.10 | List Evaluable Function | 13.3.3 |
| `list evaluable predicate` | 2.10 | List Evaluable Predicate | 13.3.3 |
| `list ff` | 2.10 | List Functional Fact Predicate | 13.3.3 |
| `list function` | 2.10 | List Function | 13.3.3 |
| `list op_predicate` | 2.10 | List OP Predicate | 13.3.3 |
| `list opfs` | 2.3 | List Loaded OP Files | 13.3.1 |
| `list op` | 2.3 | List Loaded OPs | 13.3.3 |
| `list predicate` | 2.10 | List Predicate | 13.3.3 |
| `load db 'file_name'` | 2.2 | Load Database... | 13.3.1 |
| `load opf op_graph\|file_name` | 2.4 | Load OP File... | 16.1.1 |
| `q\|quit\|exit\|EOF` | 2.13 | Quit | 13.3.1 |
| `reload opf 'file_name'` | 2.4 | Reload OP File... | 13.3.1 |
| `require file_name` | 2.4 | None | |
| `reset kernel` | 2.13 | Reset | 13.4.2 |
| `save db 'file_name'` | 2.2 | Save Database... | 13.3.1 |
| `send name message` | 2.13 | None | |
| `set action on\|off` | 2.8 | Compiler Check Action | 13.3.5 |
| `set eval post on\|off` | 2.6 | Eval On Post | 13.3.5 |
| `set function on\|off` | 2.8 | Compiler Check Function | 13.3.5 |
| `set meta fact op on\|off` | 2.7 | Post Meta Fact: (APPLICABLE-OPS-FACT ...) | 13.3.5 |
| `set meta fact on\|off` | 2.7 | Post Meta Fact: (FACT-INVOKED-OPS ...) | 13.3.5 |
| `set meta goal op on\|off` | 2.7 | Post Meta Fact: (APPLICABLE-OPS-GOAL ...) | 13.3.5 |
| `set meta goal on\|off` | 2.7 | Post Meta Fact: (GOAL-INVOKED-OPS ...) | 13.3.5 |
| `set meta on\|off` | 2.7 | Meta Level | 13.3.5 |
| `set parallel intend on\|off` | 2.6 | Parallel Intend | 13.3.5 |
| `set parallel intention on\|off` | 2.6 | Parallel Intention Execution | 13.3.5 |
| `set parallel post on\|off` | 2.6 | Parallel Goal Posting | 13.3.5 |
| `set predicate on\|off` | 2.8 | Compiler Check Predicate | 13.3.5 |
| `set soak on\|off` | 2.7 | Post Meta Fact: (SOAK ...) | 13.3.5 |
| `set symbol on\|off` | 2.8 | Compiler Check Symbol | 13.3.5 |
| `set time_stamping on\|off` | 2.6 | Time Stamping | 13.3.5 |
| `show copyright` | 2.13 | None | |
| `show db` | 2.2 | Show Database | 13.3.3 |
| `show variable` | 2.13 | Show Global Variables | 13.3.3 |
| `show intention` | 2.13 | Show Intentions | 13.3.3 |
| `show memory` | 2.13 | Show Memory Usage | 13.3.3 |
| `show op op_name` | 2.3 | Display a Particular OP | 13.3.6 |
| `show version` | 2.13 | None | |
| `stat all` | 2.13 | Stat All Hashtables | 13.3.3 |
| `stat db` | 2.13 | Stat Database Hashtables | 13.3.3 |
| `stat id` | 2.13 | Stat Symbol Hashtable | 13.3.3 |

Table E.2: Commands Equivalence Between the Kernels (Second Part)

| Command Name | Section | Menu Item | Section |
|---|---|---|---|
| `trace all on\|off` | 2.5 | None | |
| `trace applicable op on\|off` | 2.5 | Soak | 13.3.4 |
| `trace db frame on\|off` | 2.5 | Database Frames | 13.3.4 |
| `trace db on\|off` | 2.5 | Database operations | 13.3.4 |
| `trace fact on\|off` | 2.5 | Fact Posting | 13.3.4 |
| `trace feature on\|off` | 2.5 | Trace | 13.3.4 |
| `trace goal on\|off` | 2.5 | Goal Posting | 13.3.4 |
| `trace graphic on\|off` | 2.5 | OP Graphic | 13.3.4 |
| `trace graphic op op_name on\|off` | 2.3 | OP Trace/Step... | 13.3.4 |
| `trace graphic opf file_name on\|off` | 2.3 | None | |
| `trace intend on\|off` | 2.5 | Intention | 13.3.4 |
| `trace load op on\|off` | 2.5 | OP Compiler | 13.3.4 |
| `trace receive on\|off` | 2.5 | Message Reception | 13.3.4 |
| `trace relevant op on\|off` | 2.5 | Relevant OP | 13.3.4 |
| `trace send on\|off` | 2.5 | Message Sent | 13.3.4 |
| `trace suc_fail on\|off` | 2.5 | OP Success Failure | 13.3.4 |
| `trace intention failure on\|off` | 2.5 | Intention Failure | 13.3.4 |
| `trace text on\|off` | 2.5 | OP Text | 13.3.4 |
| `trace text op op_name on\|off` | 2.3 | OP Text Trace... | 13.3.4 |
| `trace text opf file_name on\|off` | 2.3 | None | |
| `trace thread on\|off` | 2.5 | Thread Forking/Joining | 13.3.4 |
| `undeclare be predicate` | 2.9 | None | |
| `unify expression expression` | 2.13 | None | |

Table E.3: Commands Equivalence Between the Kernels (Third Part)

# Appendix F

# Default OPs

Some default OPs are provided as is in the standard OPRS distribution. However, unlike SRI PRS, these default OPs are not loaded by default in all OPRS Kernels. Most of these files comes with a *'.sym'* and *'.inc'* companion. If this is the case, always load the coresponding *'.inc'* file.

## F.1 *'new-default.opf'*

Here is the list of default procedures provided with the current revision of OPRS Development Environment. They can be found in the file *'new-default.opf'*. Final user are encourage to make a copy of this file and select the OP they need and they want to keep for their application.

- —**Apply-Sort-Predicate-To-All**—

  A graphic action OP.

  **Invocation:** (! (APPLY-SORT-PREDICATE-TO-ALL))

  **Call:** (APPLY-SORT-PREDICATE-TO-ALL)

  **Context:** ()

  **Action:** (APPLY-SORT-PREDICATE-TO-ALL)

  **Properties:** ((DECISION-PROCEDURE T))

  **Documentation:** "This OP will apply the sort intentions function, to
      all the intentions."

- —**Asleep Intention Cond**—

  A graphic action OP.

  **Invocation:** (! (ASLEEP-INTENTION-COND $X $COND))

  **Call:** (ASLEEP-INTENTION-COND $X $COND)

**Context:** `()`

**Action:** `(ASLEEP-INTENTION-COND $X $COND)`

**Documentation:** `"This OP will asleep the intention in $x the LISP_CAR containing`
`intention passed as argument.`
`It will add the gexpr built from the TermComp $COND`
`in the activation condition."`

- **—Asleep Intentions Cond—**

  A graphic action OP.

  **Invocation:** `(! (ASLEEP-INTENTIONS-COND $X $COND))`

  **Call:** `(ASLEEP-INTENTIONS-COND $X $COND)`

  **Context:** `()`

  **Action:** `(ASLEEP-INTENTIONS-COND $X $COND)`

  **Documentation:** `"This OP will asleep all the intentions in $x the LISP_LIST`
  `of intentions passed as argument.`
  `It will add the gexpr built from the TermComp $COND`
  `in the activation condition."`

- **—Asleep Intentions—**

  A graphic action OP.

  **Invocation:** `(! (ASLEEP-INTENTIONS $X $ID))`

  **Call:** `(ASLEEP-INTENTIONS $X $ID)`

  **Context:** `()`

  **Action:** `(ASLEEP-INTENTIONS $X $ID)`

  **Documentation:** `"This OP will asleep all the intentions in $x the LISP_LIST of`
  `intentions passed as argument.`
  `It will add the fact (WAKE-UP-INTENTION $ID) in the activation condition."`

- **—Asleep Intention—**

  A graphic action OP.

  **Invocation:** `(! (ASLEEP-INTENTION $X $ID))`

  **Call:** `(ASLEEP-INTENTION $X $ID)`

  **Context:** `()`

  **Action:** `(ASLEEP-INTENTION $X $ID)`

  **Documentation:** `"This OP will asleep the intention in $x the LISP_CAR containing`
  `intention passed as argument.`
  `It will add the fact (WAKE-UP-INTENTION $ID) in the activation condition."`

- —**Broadcast Message**—

  A graphic action OP.

  **Invocation:** `(! (BROADCAST-MESSAGE $MESSAGE))`

  **Call:** `(BROADCAST-MESSAGE $MESSAGE)`

  **Context:** `()`

  **Action:** `(BROADCAST-MESSAGE $MESSAGE)`

  **Documentation:** `"Send the message $MESSAGE to all the agents registered to the message passer, except the sender."`

- —**End Critical Section**—

  A graphic action OP.

  **Invocation:** `(! (END-CRITICAL-SECTION))`

  **Call:** `(END-CRITICAL-SECTION)`

  **Context:** `()`

  **Action:** `(END-CRITICAL-SECTION)`

  **Documentation:** `"This OP will quit the critical section of the current intention."`

- —**Execute Command**—

  A graphic action OP.

  **Invocation:** `(! (EXECUTE-COMMAND $COMMAND))`

  **Call:** `(EXECUTE-COMMAND $COMMAND)`

  **Context:** `()`

  **Action:** `(EXECUTE-COMMAND $COMMAND)`

  **Documentation:** `"This OP execute the command given as argument."`

- —**Fail**—

  A graphic action OP.

  **Invocation:** `(! (FAILED))`

  **Call:** `(FAILED)`

  **Context:** `()`

  **Action:** `(FAIL)`

  **Documentation:** `"This action will fail... This OP can be used to fail the branch of a OP, for example"`

- —**Get All Intentions**—

  A graphic action OP.

**Invocation:** `(! (GET-ALL-INTENTIONS $LI))`

**Call:** `(GET-ALL-INTENTIONS $LI)`

**Context:** `()`

**Action:** `(*=* $LI`
`        (GET-ALL-INTENTIONS))`

**Documentation:** `"This OP will return the LISP-LIST of`
`    all the intentions."`

- **—Get Current Intention—**

  A graphic action OP.

  **Invocation:** `(! (GET-CURRENT-INTENTION $CI))`

  **Call:** `(GET-CURRENT-INTENTION $CI)`

  **Context:** `()`

  **Action:** `(*=* $CI`
  `        (GET-CURRENT-INTENTION))`

  **Documentation:** `"This OP will return the current`
  `    intention in the LISP_CAR $CI ."`

- **—Get Float Array—**

  A graphic action OP.

  **Invocation:** `(! (GET-FLOAT-ARRAY $ARRAY $INDEX $VALUE))`

  **Call:** `(GET-FLOAT-ARRAY $ARRAY $INDEX $VALUE)`

  **Context:** `()`

  **Action:** `(*=* $VALUE`
  `        (GET-FLOAT-ARRAY $ARRAY $INDEX))`

  **Documentation:** `"This OP get the $VALUE contained in the float (double) array`
  `    $ARRAY at indice $INDEX."`

- **—Get Int Array—**

  A graphic action OP.

  **Invocation:** `(! (GET-INT-ARRAY $ARRAY $INDEX $VALUE))`

  **Call:** `(GET-INT-ARRAY $ARRAY $INDEX $VALUE)`

  **Context:** `()`

  **Action:** `(*=* $VALUE`
  `        (GET-INT-ARRAY $ARRAY $INDEX))`

  **Documentation:** `"This OP get the $VALUE contained in the int array`
  `    $ARRAY at indice $INDEX."`

- **—Get Intention Priority—**

  A graphic action OP.

  **Invocation:** `(! (GET-INTENTION-PRIORITY $I $P))`

  **Call:** `(GET-INTENTION-PRIORITY $I $P)`

  **Context:** `()`

  **Action:** `(*=* $P`
  `(GET-INTENTION-PRIORITY $I))`

  **Documentation:** `"This OP will return the Priority of the`
  `intention in the LISP_CAR $I ."`

- **—Get Intention Time—**

  A graphic action OP.

  **Invocation:** `(! (GET-INTENTION-TIME $I $T))`

  **Call:** `(GET-INTENTION-TIME $I $T)`

  **Context:** `()`

  **Action:** `(*=* $T`
  `(GET-INTENTION-TIME $I))`

  **Documentation:** `"This OP will return the Time (Date of Creation in sec)`
  `of the intention in the LISP_CAR $I ."`

- **—Get Other Intentions—**

  A graphic action OP.

  **Invocation:** `(! (GET-OTHER-INTENTIONS $LI))`

  **Call:** `(GET-OTHER-INTENTIONS $LI)`

  **Context:** `()`

  **Action:** `(*=* $LI`
  `(GET-OTHER-INTENTIONS))`

  **Documentation:** `"This OP will return the LISP-LIST of`
  `the other intentions."`

- **—Get Root Intentions—**

  A graphic action OP.

  **Invocation:** `(! (GET-ROOT-INTENTIONS $LI))`

  **Call:** `(GET-ROOT-INTENTIONS $LI)`

  **Context:** `()`

  **Action:** `(*=* $LI`
  `(GET-ROOT-INTENTIONS))`

**Documentation:** `"This OP will return the LISP-LIST of`
`the root intentions."`

- —**Get Sleeping Intentions**—

  A graphic action OP.

  **Invocation:** `(! (GET-SLEEPING-INTENTIONS $LI))`

  **Call:** `(GET-SLEEPING-INTENTIONS $LI)`

  **Context:** `()`

  **Action:** `(*=* $LI`
  `(GET-SLEEPING-INTENTIONS))`

  **Documentation:** `"This OP will return the LISP-LIST of`
  `the sleeping intentions."`

- —**Kill Intentions**—

  A graphic action OP.

  **Invocation:** `(! (KILL-INTENTIONS $X))`

  **Call:** `(KILL-INTENTIONS $X)`

  **Context:** `()`

  **Action:** `(KILL-INTENTIONS $X)`

  **Documentation:** `"This OP will kill all the intentions in $x the LISP_LIST of`
  `intentions passed as argument."`

- —**Kill Intention**—

  A graphic action OP.

  **Invocation:** `(! (KILL-INTENTION $X))`

  **Call:** `(KILL-INTENTION $X)`

  **Context:** `()`

  **Action:** `(KILL-INTENTION $X)`

  **Documentation:** `"This OP will kill the intention in $x the LISP_CAR containing an`
  `intentions passed as argument."`

- —**Kill other intentions**—

  A graphic action OP.

  **Invocation:** `(! (KILL-OTHER-INTENTIONS))`

  **Call:** `(KILL-OTHER-INTENTIONS)`

  **Context:** `()`

  **Action:** `(KILL-OTHER-INTENTIONS)`

**Documentation:** `"This OP will kill all the other intentions in the intention graph.`
`(it is a very dangerous action OP). Note that it does not`
`kill itself."`

- —**Make Float Array**—

  A graphic action OP.

  **Invocation:** `(! (MAKE-FLOAT-ARRAY $SIZE $ARRAY))`

  **Call:** `(MAKE-FLOAT-ARRAY $SIZE $ARRAY)`

  **Context:** `()`

  **Action:** `(*=* $ARRAY`
  `(MAKE-FLOAT-ARRAY $SIZE))`

  **Documentation:** `"This OP create a float (in fact double) array of size $SIZE and return`
  `the value in $ARRAY."`

- —**Make Int Array**—

  A graphic action OP.

  **Invocation:** `(! (MAKE-INT-ARRAY $SIZE $ARRAY))`

  **Call:** `(MAKE-INT-ARRAY $SIZE $ARRAY)`

  **Context:** `()`

  **Action:** `(*=* $ARRAY`
  `(MAKE-INT-ARRAY $SIZE))`

  **Documentation:** `"This OP create an int array of size $SIZE and return`
  `the value in $ARRAY."`

- —**Meta Intend After**—

  A graphic action OP.

  **Invocation:** `(! (INTENDED-OP-AFTER $X $INTENDED-LIST))`

  **Call:** `(INTENDED-OP-AFTER $X $INTENDED-LIST)`

  **Context:** `()`

  **Action:** `(INTEND-OP-AFTER $X $INTENDED-LIST)`

  **Documentation:** `"To intend an applicable OP ($X a Op Instance LISP_CAR),`
  `after a list of already intended procedure."`

- —**Meta Intend All OPs As Root**—

  A graphic action OP.

  **Invocation:** `(! (INTENDED-ALL-OPS-AS-ROOT $X))`

  **Call:** `(INTENDED-ALL-OPS-AS-ROOT $X)`

  **Context:** `()`

**Action:** `(INTEND-ALL-OPS-AS-ROOT $X)`

**Documentation:** `"Intend all the OPs in $X (a LISP_LIST of OP Instance)`
`as roots of the intention graph."`

- **—Meta Intend All OPs—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-ALL-OPS $X))`

  **Call:** `(INTENDED-ALL-OPS $X)`

  **Context:** `()`

  **Action:** `(INTEND-ALL-OPS $X)`

  **Documentation:** `"Intend all the OPs in the $x (a LISP_LIST of OP Instance) list."`

- **—Meta Intend All Ops After—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-ALL-OPS-AFTER $X $INTENDED-LIST))`

  **Call:** `(INTENDED-ALL-OPS-AFTER $X $INTENDED-LIST)`

  **Context:** `()`

  **Action:** `(INTEND-ALL-OPS-AFTER $X $INTENDED-LIST)`

  **Documentation:** `"To intend all the OPs in $X ( a LISP_LIST of Op Instances),`
  `after a list of already intended procedure."`

- **—Meta Intend with Priority After—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-OP-WITH-PRIORITY-AFTER $X $P $INTENDED-LIST))`

  **Call:** `(INTENDED-OP-WITH-PRIORITY-AFTER $X $P $INTENDED-LIST)`

  **Context:** `()`

  **Action:** `(INTEND-OP-WITH-PRIORITY-AFTER $X $P $INTENDED-LIST)`

  **Documentation:** `"To intend an applicable OP ($X a Op Instance LISP_CAR)`
  `based upon PRIORITY ($P a Term Integer LISP_CAR), after`
  `a list of already intended procedure."`

- **—Meta Intend with Priority—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-OP-WITH-PRIORITY $X $P))`

  **Call:** `(INTENDED-OP-WITH-PRIORITY $X $P)`

  **Context:** `()`

  **Action:** `(INTEND-OP-WITH-PRIORITY $X $P)`

**Documentation:** "The simplest way to intend an applicable OP ($X a Op Instance LISP_CAR)
based upon PRIORITY ($P a Term Integer LISP_CAR)."

- **—Meta Intend—**

  A graphic action OP.

  **Invocation:** (! (INTENDED-OP $X))

  **Call:** (INTENDED-OP $X)

  **Context:** ()

  **Action:** (INTEND-OP $X)

  **Documentation:** "The simplest way to intend an applicable OP
  ($X a Op Instance LISP_CAR)."

- **—Multicast Message—**

  A graphic action OP.

  **Invocation:** (! (MULTICAST-MESSAGE $AGENTS $MESSAGE))

  **Call:** (MULTICAST-MESSAGE $AGENTS $MESSAGE)

  **Context:** ()

  **Action:** (MULTICAST-MESSAGE $AGENTS $MESSAGE)

  **Documentation:** "Multicast the message $MESSAGE to the oprs in the $AGENTS lisp list."

- **—Print (just print an object)—**

  A graphic action OP.

  **Invocation:** (! (PRINT $X))

  **Call:** (PRINT $X)

  **Context:** ()

  **Action:** (PRINT $X)

  **Documentation:** "Print the value of $x in the Text Pane"

- **—Print Window C Format—**

  A graphic action OP.

  **Invocation:** (! (PRINTF-WINDOW $w $X))

  **Call:** (PRINTF-WINDOW $w $X)

  **Context:** ()

  **Action:** (PRINTF-WINDOW $W $X)

  **Documentation:** "This OP prints a (format ) statement in an existing window (under X only).
  It accepts some of the % C directives (%d %s %f %g %%).
  Example : (printf 0x12345 (format \"The %d of %s is %f.\" $x $y $z))"

- **—Manage Window—**

  A graphic action OP.

  **Invocation:** `(! (MANAGE-WINDOW $w))`

  **Call:** `(MANAGE-WINDOW $w)`

  **Context:** `()`

  **Action:** `(MANAGE-WINDOW $w)`

  **Documentation:** `"This OP manage an existing window"`

- **—Rename Window—**

  A graphic action OP.

  **Invocation:** `(! (RENAME-WINDOW $w $t))`

  **Call:** `(RENAME-WINDOW $w $t)`

  **Context:** `()`

  **Action:** `(RENAME-WINDOW $w $t)`

  **Documentation:** `"This OP rename an existing window"`

- **—Unmanage Window—**

  A graphic action OP.

  **Invocation:** `(! (UNMANAGE-WINDOW $w))`

  **Call:** `(UNMANAGE-WINDOW $w)`

  **Context:** `()`

  **Action:** `(UNMANAGE-WINDOW $w)`

  **Documentation:** `"This OP unmanage an existing window"`

- **—Delete Window—**

  A graphic action OP.

  **Invocation:** `(! (DELETE-WINDOW $w))`

  **Call:** `(DELETE-WINDOW $w)`

  **Context:** `()`

  **Action:** `(DELETE-WINDOW $w)`

  **Documentation:** `"This OP delete an existing window"`

- **—Print C Format—**

  A graphic action OP.

  **Invocation:** `(! (PRINTF $X))`

  **Call:** `(PRINTF $X)`

**Context:** ()

**Action:** (PRINTF $X)

**Documentation:** "This OP prints a (format ) statement.
    It accepts some of the % C directives (%d %s %f %g %%).
    Example : (printf (format \"The %d of %s is %f.\" $x $y $z))"

- **—Print List (Format like printing)—**

  A graphic action OP.

  **Invocation:** (! (PRINT-LIST $X))

  **Call:** (PRINT-LIST $X)

  **Context:** ()

  **Action:** (PRINT-INSIDE $X)

  **Documentation:** "This OP prints a (format nil ....) statement. This is a
      remanence of the LISP version of OPRS. It does not accept
      all the ˜ directives, but accepts some of the % C directives."

- **—Read Inside Id Var—**

  A graphic action OP.

  **Invocation:** (! (READ-INSIDE-ID-VAR $X $VAL))

  **Call:** (READ-INSIDE-ID-VAR $X $VAL)

  **Context:** ()

  **Action:** (READ-INSIDE-ID-VAR $X $VAL)

  **Documentation:** "This OP will asleep the current intention
      until we got the fact (READ-RESPONSE $X $VAL)."

- **—Read Inside Id—**

  A graphic action OP.

  **Invocation:** (! (READ-INSIDE-ID $X $VAL))

  **Call:** (READ-INSIDE-ID $X $VAL)

  **Context:** ()

  **Action:** (*=* $VAL
          (READ-INSIDE-ID $X))

  **Documentation:** "This OP will asleep the current intention
      until we got the fact (READ-RESPONSE $X $VAL)."

- **—Read Inside—**

  A graphic action OP.

  **Invocation:** (! (READ-INSIDE $VAL))

**Call:** `(READ-INSIDE $VAL)`

**Context:** `()`

**Action:** `(*=* $VAL`
`          (READ-INSIDE))`

**Documentation:** `"This OP will asleep the current intention`
`    until we got the fact (READ-RESPONSE $VAL)."`

- **—Send Message—**
  A graphic action OP.

  **Invocation:** `(! (SEND-MESSAGE $OPRS $MESSAGE))`

  **Call:** `(SEND-MESSAGE $OPRS $MESSAGE)`

  **Context:** `()`

  **Action:** `(SEND-MESSAGE $OPRS $MESSAGE)`

  **Documentation:** `"Send the message $MESSAGE to the oprs $OPRS agent."`

- **—Send String—**
  A graphic action OP.

  **Invocation:** `(! (SEND-STRING $AGENT $STRING))`

  **Call:** `(SEND-STRING $AGENT $STRING)`

  **Context:** `()`

  **Action:** `(SEND-STRING $AGENT $STRING)`

  **Documentation:** `"Send the string $STRING to the $AGENT."`

- **—Set Float Array—**
  A graphic action OP.

  **Invocation:** `(! (SET-FLOAT-ARRAY $ARRAY $INDEX $VALUE))`

  **Call:** `(SET-FLOAT-ARRAY $ARRAY $INDEX $VALUE)`

  **Context:** `()`

  **Action:** `(SET-FLOAT-ARRAY $ARRAY $INDEX $VALUE)`

  **Documentation:** `"This OP set the $VALUE contained in the float array`
  `    $ARRAY at indice $INDEX."`

- **—Set Int Array—**
  A graphic action OP.

  **Invocation:** `(! (SET-INT-ARRAY $ARRAY $INDEX $VALUE))`

  **Call:** `(SET-INT-ARRAY $ARRAY $INDEX $VALUE)`

  **Context:** `()`

**Action:** (SET-INT-ARRAY $ARRAY $INDEX $VALUE)

**Documentation:** "This OP set the $VALUE contained in the int array
    $ARRAY at indice $INDEX."

- —**Set Intention Priority**—
  A graphic action OP.

  **Invocation:** (! (SET-INTENTION-PRIORITY $I $P))

  **Call:** (SET-INTENTION-PRIORITY $I $P)

  **Context:** ()

  **Action:** (SET-INTENTION-PRIORITY $I $P)

  **Documentation:** "This OP will set the Priority $P to the
    intention in the LISP_CAR $I ."

- —**Sort Intention None**—
  A graphic action OP.

  **Invocation:** (! (SORT-INTENTION-NONE))

  **Call:** (SORT-INTENTION-NONE)

  **Context:** ()

  **Action:** (SORT-INTENTION-NONE)

  **Properties:** ((DECISION-PROCEDURE T))

  **Documentation:** "This OP will unset the sort intentions function."

- —**Sort Intention Priority Time**—
  A graphic action OP.

  **Invocation:** (! (SORT-INTENTION-PRIORITY-TIME))

  **Call:** (SORT-INTENTION-PRIORITY-TIME)

  **Context:** ()

  **Action:** (SORT-INTENTION-PRIORITY-TIME)

  **Properties:** ((DECISION-PROCEDURE T))

  **Documentation:** "This OP will set the sort intentions function
    to Priority then Time."

- —**Sort Intention Priority**—
  A graphic action OP.

  **Invocation:** (! (SORT-INTENTION-PRIORITY))

  **Call:** (SORT-INTENTION-PRIORITY)

  **Context:** ()

**Action:** (SORT-INTENTION-PRIORITY)

**Properties:** ((DECISION-PROCEDURE T))

**Documentation:** "This OP will set the sort intentions function
    to Priority."

- —**Sort Intention Time**—

  A graphic action OP.

  **Invocation:** (! (SORT-INTENTION-TIME))

  **Call:** (SORT-INTENTION-TIME)

  **Context:** ()

  **Action:** (SORT-INTENTION-TIME)

  **Properties:** ((DECISION-PROCEDURE T))

  **Documentation:** "This OP will set the sort intentions function
      to Time."

- —**Start Critical Section**—

  A graphic action OP.

  **Invocation:** (! (START-CRITICAL-SECTION))

  **Call:** (START-CRITICAL-SECTION)

  **Context:** ()

  **Action:** (START-CRITICAL-SECTION)

  **Documentation:** "This OP will put the current intention in a critical section."

- —**Succeed**—

  A graphic action OP.

  **Invocation:** (! (SUCCEED))

  **Call:** (SUCCEED)

  **Context:** ()

  **Action:** (SUCCEED)

  **Documentation:** "This action will succeed... This OP can be used to
      make a branch which always succeeds"

- —**Tag Current Intention**—

  A graphic action OP.

  **Invocation:** (! (TAG-CURRENT-INTENTION $X))

  **Call:** (TAG-CURRENT-INTENTION $X)

  **Context:** ()

**Action:** `(TAG-CURRENT-INTENTION $X)`

**Documentation:** `"This OP when executed will tag the current intention, i.e. the intention in which it is executed with the ID $X passed in argument."`

- —**Wake-Up Intention**—

  A graphic action OP.

  **Invocation:** `(! (WAKE-UP-INTENTION $ID))`

  **Call:** `(WAKE-UP-INTENTION $ID)`

  **Context:** `()`

  **Action:** `(WAKE-UP-INTENTION $ID)`

  **Documentation:** `"This OP will wake-up all the intentions asleep with this ID ."`

## F.2  *'meta-intended-goal.opf'*

Here is the list of meta OP to intend a goal directly distributed with the current revision of OPRS Development Environment. They can be found in the file *'meta-intended-goal.opf'*.

- —**// Apply to all after Roots**—

  A graphic OP.

  **Invocation:** `(! (//-APPLY-TO-ALL-AFTER-ROOTS (LAMBDA $VAR $GTEXPR) $LIST))`

  **Context:** `()`

  **Documentation:** `"Apply the same goal to a LISP_LIST list of variables. Note the construction of the goal-list LISP_LIST. It is a kind of MAP-OP."`

- —**// Apply to all as roots with priority**—

  A graphic OP.

  **Invocation:** `(! (//-APPLY-TO-ALL-AS-ROOTS-WITH-PRIORITY (LAMBDA $VAR $GTEXPR) $LIST $L-PR))`

  **Context:** `()`

  **Documentation:** `"Apply the same goal to a LISP_LIST list of variable and a LISP_LIST of pric Note the construction of the goal-list. It is a kind of MAP-OP."`

- —**Apply to all (already built goals)**—

  A graphic OP.

  **Invocation:** `(! (INTENDED-ALL $GOAL-LIST))`

**Context:** ()

**Documentation:** `"Post the goal separately."`

- —**Apply to all as roots**—

  A graphic OP.

  **Invocation:** `(! (APPLY-TO-ALL-AS-ROOTS (LAMBDA $VAR $GTEXPR) $LIST))`

  **Context:** ()

  **Documentation:** `"Apply the same goal to a LISP_LIST list of variables,`
  `    and intend its as root (before all intentions)."`

- —**Apply to all before me**—

  A graphic OP.

  **Invocation:** `(! (APPLY-TO-ALL-BEFORE-ME (LAMBDA $VAR $GTEXPR) $LIST))`

  **Context:** ()

  **Documentation:** `"Apply the same goal to a LISP_LIST list of variables,`
  `    and intend its as root (before the current intention)."`

- —**Apply to all before other**—

  A graphic OP.

  **Invocation:** `(! (APPLY-TO-ALL-BEFORE-OTHER (LAMBDA $VAR $GTEXPR) $LIST))`

  **Context:** ()

  **Documentation:** `"Apply the same goal to a LISP_LIST list of variables,`
  `    and intend its as root (before all intentions)."`

- —**Apply to all with priority**—

  A graphic OP.

  **Invocation:** `(! (APPLY-TO-ALL-WITH-PRIORITY (LAMBDA $VAR $GTEXPR) $LIST $PRIOR))`

  **Context:** ()

  **Documentation:** `"Apply the same goal to a LISP_LIST list of variables,`
  `    and intend its respectively with the priority in $PRIOR."`

- —**Apply to all**—

  A graphic OP.

  **Invocation:** `(! (APPLY-TO-ALL (LAMBDA $VAR $GTEXPR) $LIST))`

  **Context:** ()

  **Documentation:** `"Apply the same goal to a LISP_LIST list of variables."`

- —**Build goal list without var**—

  A graphic OP.

  **Invocation:** (! (BUILD-GOAL-LIST-NO-VAR $LIST-GTEXPR $LIST-GOAL))

  **Context:** ()

  **Documentation:** "Build a Goal List using a LISP_LIST list of gtexpr."

- —**Build goal list**—

  A graphic OP.

  **Invocation:** (! (BUILD-GOAL-LIST (LAMBDA $VAR $GTEXPR) $LIST-VAR $LIST-GOAL))

  **Context:** ()

  **Documentation:** "Build a Goal List using the same goal to
      a LISP_LIST list of variables."

- —**Meta-Intend all goals // after Roots**—

  A graphic action OP.

  **Invocation:** (! (INTENDED-ALL-GOALS-//-AFTER-ROOTS $X))

  **Context:** ()

  **Action:** (INTEND-ALL-GOALS-//-AFTER-ROOTS $X)

  **Documentation:** "Intend all the Goals in $X (a Goal LISP_LIST)
      after all the roots of the intention graph."

- —**Meta-Intend all goals // after**—

  A graphic action OP.

  **Invocation:** (! (INTENDED-ALL-GOALS-//-AFTER $X $AFTER))

  **Context:** ()

  **Action:** (INTEND-ALL-GOALS-//-AFTER $X $AFTER)

  **Documentation:** "Intend all the Goals in $X (a Goal LISP_LIST)
      after all the intentions in $after."

- —**Meta-Intend all goals // as Roots with priority**—

  A graphic action OP.

  **Invocation:** (! (INTENDED-ALL-GOALS-//-AS-ROOTS-WITH-PRIORITY $X $P))

  **Context:** ()

  **Action:** (INTEND-ALL-GOALS-//-AS-ROOTS-WITH-PRIORITY $X $P)

  **Documentation:** "Intend all the Goals in $X (a LISP_LIST of Goal) with
      the priority in $P (a LISP_LIST of integer priority)
      as  roots of the intention graph."

- **—Meta-Intend all goals // as roots—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-ALL-GOALS-//-AS-ROOTS $X))`

  **Context:** `()`

  **Action:** `(INTEND-ALL-GOALS-//-AS-ROOTS $X)`

  **Documentation:** `"Intend all the Goals in $X (a Goal LISP_LIST)`
  `    as roots of the intention graph."`

- **—Meta-Intend all goals //—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-ALL-GOALS-// $X))`

  **Context:** `()`

  **Action:** `(INTEND-ALL-GOALS-// $X)`

  **Documentation:** `"Intend all the Goals in $X (a Goal LISP_LIST)`
  `    after the current intention."`

- **—Meta-Intend goal after before with priority—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-GOAL-WITH-PRIORITY-AFTER-BEFORE $X $P $AFTER $BEFORE))`

  **Context:** `()`

  **Action:** `(INTEND-GOAL-WITH-PRIORITY-AFTER-BEFORE`
  `       $X $P $AFTER $BEFORE)`

  **Documentation:** `"Intend the Goal in $X (a LISP_CAR Goal)`
  `    with the priority $P, after all the intentions`
  `    in $AFTER, and before all in $BEFORE."`

- **—Meta-Intend goal after before—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-GOAL-AFTER-BEFORE $X $AFTER $BEFORE))`

  **Context:** `()`

  **Action:** `(INTEND-GOAL-AFTER-BEFORE $X $AFTER $BEFORE)`

  **Documentation:** `"Intend the Goal in $X (a LISP_CAR Goal)`
  `    after all the intentions in $AFTER, and before all in $BEFORE."`

- **—Meta-Intend goal with priority—**

  A graphic action OP.

  **Invocation:** `(! (INTENDED-GOAL-WITH-PRIORITY $X $P))`

**Context:** ()

**Action:** (INTEND-GOAL-WITH-PRIORITY $X $P)

**Documentation:** "Intend the Goal in $X (a LISP_CAR Goal)
    with the priority $P, after the current intention."

- **—Meta-Intend goal—**

  A graphic action OP.

  **Invocation:** (! (INTENDED-GOAL $X))

  **Context:** ()

  **Action:** (INTEND-GOAL $X)

  **Documentation:** "Intend the Goal in $X (a LISP_CAR Goal),
      after the current intention."

# F.3   *'new-meta-ops.opf'*

Here is the list of meta procedures provided with the current revision of OPRS Development Environment. They can be found in the file *'new-meta-ops.opf'*.

   **'This file contains more than on meta level OP, do not load all of them at the same time in one application, you would get a very weird behavior. Just pick up the one which seems to be appropriate to your application.'**

- **—Called From Meta Selector With Priority—**

  A graphic OP.

  **Invocation:** (! (META-INTENDED-ALL-WITH-PRIORITY
         $OPS-TO-INTEND $INTENDED-DECISION-PROCEDURES))

  **Context:** ()

  **Effects:** ()

  **Documentation:** "This meta OP will intend applicable
     OPs based on their priority."

- **—Meta Selector (facts preferred and ordered)—**

  A graphic OP.

  **Invocation:** (SOAK $X)

  **Context:** ((|| (> (LENGTH $X) 1)
        (& (EQUAL (LENGTH $X) 1)
          (IS-FACT-INVOKED (FIRST $X))
          (NOT-AN-INSTANCE-OF-ME (FIRST $X)))))

  **Effects:** ()

**Properties:** ((DECISION-PROCEDURE T))

**Documentation:** "This Meta OP is used whenever there are one or more than
 one OP applicable. If there are fact invoked OPs, they
 are all intended (and the goal invoked OPs are discarded), otherwise
 it will randomly choose one of the goal invoked OPs.
 It intends the fact-invoked-ops which are decision-procedure
 as root of the graph, and the other ones after the already
 intended op-instance which are decision procedure."

- **—Meta Selector (facts preferred)—**

  A graphic OP.

  **Invocation:** (SOAK $X)

  **Context:** ((> (LENGTH $X) 1))

  **Effects:** ()

  **Properties:** ((DECISION-PROCEDURE T))

  **Documentation:** "This Meta OP is used whenever there are more than one OP applicabl
   If there are fact invoked OPs, they are all intended (and the goal
   invoked OPs are discarded), otherwise it will randomly choose one of
   the goal invoked OPs. As a result, the behavior of the kernel is
   very reactive. Note that it can apply to itself without any problem
   because this very Meta OP *is* fact invoked so will be intented by
   its grand brother."

- **—Meta Selector (only facts)—**

  A graphic OP.

  **Invocation:** (FACT-INVOKED-OPS $F)

  **Context:** ((|| (> (LENGTH $F) 1)
     (& (EQUAL (LENGTH $F) 1)
      (NOT-AN-INSTANCE-OF-ME (FIRST $F)))))

  **Effects:** ()

  **Properties:** ((DECISION-PROCEDURE T))

  **Documentation:** "This Meta OP is used whenever there are one or more than
   one Fact Invoked OP applicable.
   It intends the fact-invoked-ops which are decision-procedure
   as root of the graph, and the other ones after the already
   intended op-instance which are decision procedure."

- **—Meta Selector With Priority—**

  A graphic OP.

  **Invocation:** (FACT-INVOKED-OPS $FACT-INVOKED-OPS)

**Context:** ((|| (> (LENGTH $FACT-INVOKED-OPS) 1)
         (& (EQUAL (LENGTH $FACT-INVOKED-OPS) 1)
           (~ (PROPERTY-P
                META-SELECTOR-WITH-PRIORITY
                (FIRST $FACT-INVOKED-OPS))))))

**Effects:** ()

**Properties:** ((DECISION-PROCEDURE T)
    (META-SELECTOR-WITH-PRIORITY T))

**Documentation:** "Meta OP used whenever there is more than
    one fact invoked OP applicable."

# F.4   *'semaphore.opf'*

Here is the list of procedures, provided with the current revision of OPRS Development Environment, which implement semaphores. They can be found in the file *'semaphore.opf'*. Make sure you do not load the *'.opf'* directly but include the *'semaphore.inc'* instead. Indeed, some important declaration are made in the *'semaphore.sym'* file which is loaded by the *'semaphore.inc'* file.

This OP library provides two type of semaphores: SEM-BASIC semaphores and SEM-FIFO semaphores. For both types, there is a give, a take and a take with timeout OPs. See the OP documentation for more details.

- —**Semaphore Create**—

  A graphic OP.

  **Invocation:** (! (SEM-CREATE $SEM $NUM $TYPE))

  **Context:** ()

  **Documentation:** "Create a semaphore called $SME
      and intialize it with $NUM."

- —**Semaphore FIFO Give**—

  A graphic OP.

  **Invocation:** (! (SEM-V $SEM))

  **Context:** ((SEMAPHORE-TYPE $SEM SEM-FIFO))

  **Documentation:** "This OP releases the semaphore $sem.
      Note it checks it has it first."

- —**Semaphore FIFO Take Timeout**—

  A graphic OP.

  **Invocation:** (! (SEM-P-TIMEOUT $SEM $TIMEOUT))

  **Context:** ((SEMAPHORE-TYPE $SEM SEM-FIFO))

**Documentation:** "This OP will get the FIFO semaphore $SEM.
    Note it waits at most $TIMEOUT to get it."

- **—Semaphore FIFO Take—**

  A graphic OP.

  **Invocation:** (! (SEM-P $SEM))

  **Context:** ((SEMAPHORE-TYPE $SEM SEM-FIFO))

  **Documentation:** "This OP will get the FIFO semaphore $SEM.
    Note it waits for ever to get it."

- **—Semaphore Give—**

  A graphic OP.

  **Invocation:** (! (SEM-V $SEM))

  **Context:** ((SEMAPHORE-TYPE $SEM SEM-BASIC))

  **Documentation:** "This OP releases the semaphore $sem.
    Note it checks it has it first."

- **—Semaphore Reset—**

  A graphic OP.

  **Invocation:** (! (SEM-RESET $SEM $NUM))

  **Context:** ()

  **Documentation:** "Reset a semaphore called $SME
      and intialize it with $NUM."

- **—Semaphore Take Timeout—**

  A graphic OP.

  **Invocation:** (! (SEM-P-TIMEOUT $SEM $TIMEOUT))

  **Context:** ((SEMAPHORE-TYPE $SEM SEM-BASIC))

  **Documentation:** "This OP will get the semaphore $SEM.
    It waits at most $TIMEOUT to get it."

- **—Semaphore Take—**

  A graphic OP.

  **Invocation:** (! (SEM-P $SEM))

  **Context:** ((SEMAPHORE-TYPE $SEM SEM-BASIC))

  **Documentation:** "This OP will get the semaphore $SEM.
    Note it waits for ever to get it."

# Appendix G

# Library and Kernel Functions

Few libraries (currently two) come with OPRS Development Environment. All of them are needed to write module to connect to the Message Passer. However, to write your own kernel, i.e. to extend relocatable, you may need access to functions defined in these relocatable (most likely to write evaluable functions, evaluable predicates and actions). In this chapter, we shall describe the libraries and the kernel functions which can or must be used by the user.

## G.1    Kernel Functions

The `oprs-relocatable` and the `xoprs-relocatable` files contains already a number of functions the user can use. Considering that these functions are already present in the relocatable, there is no special action to use them (no new file to link).

### G.1.1    Data Structures and Types Used

Most data structures are hidden behind opaque pointers. Access functions are provided when required. However, some structure definitions are given to the end user.

Here is the definition of the `Oprs_Date` structure:

```
#include <sys/time.h>

typedef struct timeval PDate;
```

Here is the definition of the `Term` structure:

```
/* Definition of the types used in a Term */
/* Under Windows 95... FLOAT and ATOM are already defined. Therefore
```

```
   they are renamed TT_FLOAT and TT_ATOM */
typedef enum {INTEGER, FLOAT, STRING, ATOM, TERM_COMP, VARIABLE, GTEXPRESSION,
      LEXPRESSION, LENV, LISP_LIST, INT_ARRAY, FLOAT_ARRAY, U_POINTER} Term_Type;

/* Definition of a composed Term. For example in (P a (f g h) c), (f g h) is a
 * composed term. */
typedef struct term_comp {
 Function function;
 int  n_arg;
 TermList  terms;
} Term_Comp;

/* Definition of a term (a typed union) */
typedef struct term Term;

struct term {
 Term_Type type;
 union {
   int  intval;
   double doubleval;
   char *string;
   char *id;
   Term_Comp *term;
   Gtexpression *gtexpr;
   Lexpression *lexpr;
   VarList var_list;
   Envar *var;
   L_Car l_car;
   L_List l_list;
   Int_array *int_array;
   Float_array *float_array;
   void *u_pointer;
 }u;
};
```

### G.1.2   Important Variables

The following symbols can be found in *'oprs-type-pub.h'.* (see [Special Symbols],
§3.3, page 49)

**nil_sym**                                      **Kernel Variable**

> **extern Symbol nil_sym** is the **nil** symbol.  This is the symbol you
> should return (in a Term) when an action fails.

**lisp_t_sym**                                   **Kernel Variable**

> **extern Symbol lisp_t_sym** is the T symbol.

## wait_sym                                    **Special Symbols**

**extern Symbol wait_sym** is the **:WAIT** symbol as returned by actions when they have not completed their computation.

## current_oprs                                   **Kernel Variable**

**extern Oprs * current_oprs** is the global variable which points at the current OPRS kernel. It is used whenever you want to access some specific modules of the kernel, like the intention graph.

## current_tib                                    **Kernel Variable**

**extern Thread_Intention_Block * current_tib** is the global variable which points at the current tib.

## current_intention                              **Kernel Function**

**extern Intention* current_intention** is the current intention of the intention graph **ig** (**NULL** if not applicable).

## main_loop_pool_sec                             **Kernel Variable**

**extern long main_loop_pool_sec** is the global variable which points at the number of seconds the OPRS Kernel will wait before checking the sleeping conditions of the sleeping intentions. It is used in conjunction with **main_loop_pool_usec** presented below. It is defined in *'default-user-external.h'*.

## main_loop_pool_usec                            **Kernel Variable**

**extern long main_loop_pool_usec** is the global variable which points at the number of micro seconds the OPRS Kernel will wait before checking the sleeping conditions of the sleeping intentions. It is used in conjunction with **main_loop_pool_sec** presented above. It is defined in *'default-user-external.h'*.

Example of use (withdrawn from *'default-user-external.c'*):

```
void start_kernel_user_hook()
{
 intention_list_sort_predicate = &my_intention_list_sort;
 main_loop_pool_sec = 0L;
 main_loop_pool_usec = 10000L; /* 10 milliseconds */

}
```

## x_oprs_top_level_widget                        **Kernel Variable**

**extern Widget x_oprs_top_level_widget** is the variable which points at the X-OPRS top level widget. It can be used by the user to hook its own Xt widget in the widget tree, for its own interface. It is defined in *'xp-main-pub.h'*.

### G.1.3   Important Constants

The 2 following constants can be found in *'constant-pub.h'*.

```
#define TRUE 1
#define FALSE 0
```

### G.1.4   Oprs Manipulation Functions

These function prototypes can be found in the file *'oprs_f-pub.h'*.

oprs_intention_graph                              **Kernel Function**

> `Intention_Graph * oprs_intention_graph (Oprs *oprs)` returns
> a pointer to the intention graph of the OPRS passed as an argument
> (most likely `current_oprs`).

### G.1.5   Array Manipulation Functions

These function prototypes can be found in the file *'oprs-array_f-pub.h'*.

make_float_array_from_array                    **Kernel Function**

> `Term * make_float_array_from_array (int size, double *array)`
> returns a `Term *` of type `FLOAT_ARRAY` containing the `array` of size
> `size` passed in argument.

make_int_array_from_array                      **Kernel Function**

> `Term * make_int_array_from_array (int size, int *array)` re-
> turns a `Term *` of type `INT_ARRAY` containing the `array` of size `size`
> passed in argument.

get_array_from_float_array                     **Kernel Function**

> `double * get_array_from_float_array (Term *t)` returns a dou-
> ble `*` pointer to the array of the `FLOAT_ARRAY` contained in the term
> `t`.

get_array_from_int_array                       **Kernel Function**

> `int * get_array_from_int_array (Term *t)` returns a int `*` pointer
> to the array of the `INT_ARRAY` contained in the term `t`.

get_float_array_size                           **Kernel Function**

> `int  get_float_array_size (Term *t)` returns an int which is the
> size of the `FLOAT_ARRAY` contained in the term `t`.

get_int_array_size                             **Kernel Function**

> `int  get_int_array_size (Term *t)`  returns an int which is the
> size of the `INT_ARRAY` contained in the term `t`.

## G.1.6  Fact and Goal Manipulation Functions

These function prototypes can be found in the file *'fact-goal_f-pub.h'*.

### fprint_goal                                      **Kernel Function**

void fprint_goal (FILE *file, Goal* goal) prints the goal in
the file file.

### print_goal                                       **Kernel Function**

void print_goal (Goal *goal) prints the goal on stdout.

### fprint_fact                                       **Kernel Function**

void fprint_fact (FILE *file, Fact *fact) prints the fact in
the file file.

### print_fact                                        **Kernel Function**

void print_fact (Fact *fact) prints the fact on stdout.

### fact_soak                                         **Kernel Function**

Oprs_Date fact_soak (Fact *fact)  returns the date at which
this fact was taken into account by the soak mechanism.  This
is only meaningful if Time Stamping has been allowed (see [OPRS
Kernel Run Option Commands], §2.6, page 32, set time_stamping
on|off).

### fact_sender                                       **Kernel Function**

PString fact_sender (Fact *fact) returns the name of the sender
if this fact was a message, NULL if it is an internally generated fact.
This function can in fact be used to check if the fact is an external
message or a fact.

### fact_soak                                         **Kernel Function**

Oprs_Date fact_soak (Fact *fact)  returns the date at which
this fact was taken into account by the soak mechanism.  This
is only meaningful if Time Stamping has been allowed (see [OPRS
Kernel Run Option Commands], §2.6, page 32, set time_stamping
on|off).

### fact_creation                                     **Kernel Function**

Oprs_Date fact_creation (Fact *fact) returns the date at which
this fact was created.This is only meaningful if Time Stamping has
been allowed (see [OPRS Kernel Run Option Commands], §2.6, page
32, set time_stamping on|off).

### fact_reception                                    **Kernel Function**

`Oprs_Date fact_reception (Fact *fact)` returns the date at which this `fact` was received by the kernel.This is only meaningful if Time Stamping has been allowed (see [OPRS Kernel Run Option Commands], §2.6, page 32, `set time_stamping on|off`).

### fact_response                                   **Kernel Function**

`Oprs_Date fact_response (Fact *fact)` returns the date at which all the applicable OPs have completed because of this `fact` (if no OP was applicable, it returns a zero date). This is only meaningful if Time Stamping has been allowed (see [OPRS Kernel Run Option Commands], §2.6, page 32, `set time_stamping on|off`).

### goal_soak                                       **Kernel Function**

`Oprs_Date goal_soak (Goal *goal)` returns the date at which this `goal` was taken into account by the soak mechanism. This is only meaningful if Time Stamping has been allowed (see [OPRS Kernel Run Option Commands], §2.6, page 32, `set time_stamping on|off`).

### goal_creation                                   **Kernel Function**

`Oprs_Date goal_creation (Goal *goal)` returns the date at which this `goal` was created. This is only meaningful if Time Stamping has been allowed (see [OPRS Kernel Run Option Commands], §2.6, page 32, `set time_stamping on|off`).

### goal_reception                                  **Kernel Function**

`Oprs_Date goal_reception (Goal *goal)` returns the date at which this `goal` was received by the kernel. This is only meaningful if Time Stamping has been allowed (see [OPRS Kernel Run Option Commands], §2.6, page 32, `set time_stamping on|off`).

### goal_response                                   **Kernel Function**

`Oprs_Date goal_response (Goal *goal)` returns the date at which all the applicable OPs have completed because of this `goal` (if no OP was applicable, it returns a zero date). This is only meaningful if Time Stamping has been allowed (see [OPRS Kernel Run Option Commands], §2.6, page 32, `set time_stamping on|off`).

## G.1.7   Fact Posting Functions

The kernel provide a number of functions to allow the user to add a new fact from its own code. This mechanism can be very useful on a VxWorks implementation as it will allow the user to post/add a fact from a different process (one the same board though). Under traditional Unix system, these functions work too but

can be more simply emulated using the `send_command_to_parser` function (see [Miscellaneous Kernel Functions], §G.1.14, page 349).

**'One should make sure that a OPRS Kernel is running and is alive before calling these functions. If no OPRS Kernel is running, these functions will most likely crash the calling process.'**

These function prototypes can be found in the file *'oprs_f-pub.h'*.

## add_external_fact                    Kernel Function

> void  add_external_fact (char *predicat, TermList param_list) This function will add the fact built with `predicat` and `param_list` in the `current_oprs` kernel. The `param_list` should be used only once. The `predicat` argument can be freed after the call. Under VXWOrks, this function can be called from another process. Proper variable modification is protected by mutex semaphores.

## make_external_term_list              Kernel Function

> TermList make_external_term_list (int nb_arg, ...)    Build and return a "use once only" TermList containing `nb_arg` elements built with the subsequent paire `TermType`, `Term *` in the argument list. See the example below. All the element are duplicated, i.e. strings, symbols are appropriately copied, except for those produce with `make_external_term_comp` and `make_external_lisp_list`. You should therefore appropriately free the objects passed to this function.

## make_external_term_comp              Kernel Function

> Term_Comp * make_external_term_comp (char *function, TermList param_list) This function will build a "use once only" TermComp built with `function` and `param_list`. The `param_list` should be used only once. The `function` argument can be freed after the call. This function is usually used inside `make_external_term_list`, in which case its result should not be freed.

## make_external_lisp_list              Kernel Function

> L_List make_external_lisp_list (TermList param_list) This function will build a "use once only" L_List built with `param_list`. The `param_list` should be used only once. This function is usually used inside `make_external_term_list`, in which case its result should not be freed.

Here is a complete example to illustrate the use of the functions presented above:

```
add_external_fact(
    "boo",
    make_external_term_list(
```

```
            6,
            FLOAT, 3.1415,
            INTEGER, 5,
            U_POINTER, 0x123456,
            ATOM, "foobar",
            STRING, "This is a string in the fact",
            LISP_LIST, make_external_lisp_list(
                 make_external_term_list(
                      2,
                      FLOAT, 0.0,
                      ATOM, "second")),
            TERM_COMP, make_external_term_comp(
                 "bar",
                 make_external_term_list(
                      2,
                      INTEGER, -1,
                      ATOM, "atom"))));
```

a call to the function above will post the fact: `(BOO 3.1415 5 0x123456 FOOBAR "This is a string in the fact" (. 0.0 SECOND. ) (BAR -1 ATOM))`

## G.1.8  Intention Manipulation Functions

These function prototypes can be found in the file *'intention_f-pub.h'*.

### fprint_intention                                    **Kernel Function**

  `void fprint_intention (FILE *file, Intention *intention)` prints the `intention` in `file`.

### intention_priority                                  **Kernel Function**

  `int intention_priority (Intention *intention)` returns the priority of `intention`.

### intention_fact                                      **Kernel Function**

  `Fact * intention_fact (Intention *intention)` returns the fact which caused this `intention` to arise (`NULL` if not applicable).

### action_first_call                                   **Kernel Function**

  `PBoolean action_first_call ()` when executed in a C action code, returns `TRUE` if this is the first time this action is called, `FALSE` otherwise. This is used for action slicing to distinguish the first call from the subsequent call.

### action_number_called                                **Kernel Function**

`int action_number_called ()` when executed in a C action code, returns the number of time this action has been called in this context (it is equal to 0 in the first call) . This is used for action slicing to distinguish subsequent calls.

### `intention_goal`          **Kernel Function**

`Goal * intention_goal (Intention *intention)` returns the goal which caused this `intention` to arise (`NULL` if not applicable).

### `intention_bottom_op_instance`      **Kernel Function**

`Op_Instance * intention_bottom_op_instance (Intention *intention)` returns the bottom OP instance of this intention stack, `NULL` if the stack is empty.

## G.1.9   OP Instance Manipulation Functions

These function prototypes can be found in the file *'op-instance_f-pub.h'*.

### `fprint_op_instance`          **Kernel Function**

`void fprint_op_instance (FILE *f, Op_Instance *opi)` prints the `opi` in `file`.

### `op_instance_op`          **Kernel Function**

`Op_Structure * op_instance_op (Op_Instance *opi)` returns the Op_Structure pointed by this `opi`, i.e. the OP from which this OP_instance is an instance...

### `op_instance_goal`          **Kernel Function**

`Goal * op_instance_goal (Op_Instance *opi)` returns the goal which caused this `opi` to arise (`NULL` if not applicable).

### `op_instance_fact`          **Kernel Function**

`Fact * op_instance_fact (Op_Instance *opi)` returns the fact which caused this `opi` to arise (`NULL` if not applicable).

## G.1.10   OP Manipulation Functions

OPs and OP structures are the same type of object. When we say OP, we mean OP structure, this is explicited to avoid any confusion with OP instances (which are instances of applicable OP structures).

These function prototypes can be found in the file *'op-structure_f-pub.h'*.

### `op_name`          **Kernel Function**

`PString op_name (Op_Structure *op)` returns the name of the `op`. This function can be used to build an evaluable predicate which indicates (in a Meta OP) if a OP instance is an instance of itself...

`op_file_name`                                    **Kernel Function**

> `PString op_file_name (Op_Structure *op)` returns the file name
> of the `op`.

## G.1.11   Intention Graph Manipulation Functions

These function prototypes can be found in the file *'int-graph_f-pub.h'*.

`intention_graph_roots`                          **Kernel Function**

> `Intention_List intention_graph_roots (Intention_Graph *ig)`
> returns the current roots of the intention graph `ig`.

## G.1.12   Allocation Functions

**'Memory Allocation Rules'** The standard allocation functions are of course
available for the user to program its evaluable functions, predicates and actions
(as well as all the linked code to the kernels). However, all the objects handled
by the kernel itself (mostly terms and their components) must be allocated/freed
using the following macros. Objects pointed by USER_POINTER can be allo-
cated/freed as the user want. In any situation, never mix the different allocation
mechanism (in particular on systems where the standard memory allocator is
used: VxWorks, Purified version, etc). Mixing them will lead to allocation/free
error.

`OPRS_MALLOC`                                     **Kernel Macro**

> `void * OPRS_MALLOC (size_t nBytes)` This macro allocate `nBytes`
> of memory and return a `void *`pointer to it. The allocated mem-
> ory must be freed with `OPRS_FREE`. **'Caution:'** All objects (and
> their components) returned to the kernel must be allocated using
> the `OPRS_MALLOC` macro or the following macro `MAKE_OBJECT`.

```
Term *toto_eval_func(TermList terms)
{
    Term *res;

    res = MAKE_OBJECT(Term);

    res->type = STRING;
    res->u.string = (char *)OPRS_MALLOC(20);

    ....

    return res;
}
```

`MAKE_OBJECT`                                     **Kernel Macro**

`type * MAKE_OBJECT (type)` This macro allocates memory to store an object of type `type`, and returns a pointer to this memory block.

The parameter is a type specification. The macro uses it to allocate the right amount of memory and to generate the proper recasting instructions to make both the ıC-compiler and ılint(1) happy.

**'Caution:'** Do not allocate using your own memory allocator (providing it is linked in the OPRS Kernel). Always use the `MAKE_OBJECT` macro for your memory use.

```
Term *toto_eval_func(TermList terms)
{
 Term *t1, *res;

 res = MAKE_OBJECT(Term);
....
 return res;
}
```

## OPRS_FREE                                    **Kernel Macro**

`void OPRS_FREE (void *object)` This macro frees the memory allocated with `MAKE_OBJECT`. **'Caution:'** Do not attempt to free memory you did not allocate... You should only free temporary variables you created for your own use.

```
Term *toto_eval_func(TermList terms)
{
 Term *temporary, *res;

 res = MAKE_OBJECT(Term);
 temporary = MAKE_OBJECT(Term);
....
 OPRS_FREE(temporary);
 return res;
}
```

The prototypes of the following functions are defined in *'oprs-type_f-pub.h'*.

## make_atom                                 **Kernel Function**

`char * make_atom (char *atom)` return an atom which can then be stored in a built Term.

## find_atom                                 **Kernel Function**

`char * find_atom (char *atom)` it does exactly like `make_atom` but warn you if the symbol has not been declared previously.

### declare_atom                                        **Kernel Function**

char * declare_atom (char *atom) it does exactly like make_atom
but does not warn you if the symbol had not been declared previ-
ously.

### build_string                                        **Kernel Function**

Term * build_string (char *string) build a Term containing a
copy of the (STRING) string. It returns the pointer to this Term.

### build_integer                                       **Kernel Function**

Term * build_integer (int i) build a Term containing the inte-
ger (INTEGER) i. It returns the pointer to this Term.

### build_float                                         **Kernel Function**

Term * build_float (double i) build a Term containing the dou-
ble (FLOAT) i. It returns the pointer to this Term.

### build_l_list                                        **Kernel Function**

Term * build_l_list (L_List l)  build a Term containing the
L_List (LISP_LIST) i. It returns the pointer to this Term.

### build_c_list                                        **Kernel Function**

Term * build_c_list (OPRS_LIST l) build a Term containing the
OPRS_LIST (OPRS_LIST) l. It returns the pointer to this Term.

### build_qstring                                       **Kernel Function**

Term * build_qstring (char *i)  build a Term containing the
string (STRING) i. It returns the pointer to this Term.

### build_id                                            **Kernel Function**

Term * build_id (char *id) build a Term containing the id (ATOM)
id. It returns the pointer to this Term.

### build_t                                             **Kernel Function**

Term * build_t (void)  build a Term containing the (T) id.  It
returns the pointer to this Term.

### build_nil                                           **Kernel Function**

Term * build_nil (void)  build a Term containing the (NIL) id.
It returns the pointer to this Term.

### free_term                                           **Kernel Function**

void  free_term (Term *term)  Free a Term.

### G.1.13  LISP_LIST Manipulation Functions

These function prototypes can be found in the file *'lisp-list_f-pub.h'*.

**l_car**                                         **Kernel Function**

   `L_Car l_car (L_List l)` returns the car of an L_List.

**l_cdr**                                           **Kernel Function**

   `L_List l_cdr (L_List l)` returns the cdr of an L_List.

**l_cons**                                       **Kernel Function**

   `L_List l_cons (L_Car car, L_List cdr)` returns a new L_List, created by consing the car with the cons. The car is duplicated, but not the cdr.

**l_add_to_tail**                              **Kernel Function**

   `L_List l_add_to_tail (L_List list, L_Car car)` adds a car at the end of an L_List. The car is duplicated, not the list.

**l_length**                                   **Kernel Function**

   `int l_length (L_List l)` returns and int, the length of the L_List.

**l_nth**                                         **Kernel Function**

   `L_Car l_nth (L_List l, int i)` returns the nth L_Car element of the L_List.

**make_l_car_from_term**                   **Kernel Function**

   `L_Car make_l_car_from_term (Term *t)` returns a L_Car containing the copy of the term t.

**get_term_from_l_car**                    **Kernel Function**

   `Term * get_term_from_l_car (L_Car l)` returns a pointer to the term contained in the L_Car.

**copy_l_list**                               **Kernel Function**

   `L_List copy_l_list (L_List l)` returns a copy of the L_List.

### G.1.14  Miscellaneous Kernel Functions

**send_command_to_parser**                **Kernel Function**

   `void send_command_to_parser (PString command)` is used to get the kernel to execute a particular command. This can be used to conclude something in the database, or adding a fact, or any command parsable by the parser. You should make sure that the command has the proper syntax, or you may hang the kernel for ever... It is strongly advised to terminate the command with a new line.

## G.2 Registration and Communication Functions, *'libmp.a'*

See [How to Connect from an External Module], §12.6, page 159, for a description of the functions available to connect to the Message Passer. See [Messages Format], §12.7, page 160, for a description of the functions available to exchange messages with the Message Passer.

## G.3 *'liblist.a'* library

The *'liblist.a'* library is a very general library which contains the functions dealing with lists of objects, among other things. It is heavily used by the various parts of the OPRS Development Environment and is needed if you plan to write your own evaluable predicates or functions (to parse the argument list for example which is a `TermList`).

We present here a subset of this library. We do advise the user to stick with the functions we present below, and not to use undocumented functions from this library.

The ılistPack library provides an abstract data type called a `OPRS_LIST`, and a complete set of operations to manipulate objects of that type. In its simplest form, a `OPRS_LIST` is an ordered collection of objects which the user creates. ılistPack functions allow the user to add objects at any point in a `OPRS_LIST`, to retrieve objects from `OPRS_LIST`s, to delete objects from `OPRS_LIST`s, and to apply user functions to `OPRS_LIST`s. Furthermore, the notion of current object exists, and each `OPRS_LIST` maintains the required information a step forward and backward from the current object.

ılistPack provides the single data type `OPRS_LIST`. No operation on variables of type `OPRS_LIST` is allowed, except for assigning them the values returned by ılistPack functions and for passing them as parameters to ılistPack functions. However, it is permitted (and safe) to assign variables of type `OPRS_LIST` to other variables of type `OPRS_LIST`.

### G.3.1 Creating Lists

Before the user may place objects in a `OPRS_LIST`, he must create it with the function - `make_list()` - which returns a pointer to the newly created list.

**make_list**                                             **listPack Function**

> `OPRS_LIST make_list ()` simply creates a new list and returns a pointer to it. The value returned by `make_list()` ımust be saved by the caller in order to use that `OPRS_LIST` with subsequent ılistPack functions.

### G.3.2 Destroying Lists

When the user no longer needs a `OPRS_LIST`, he may destroy it.

### FREE_OPRS_LIST                                    **listPack Macro**

> `void FREE_OPRS_LIST (OPRS_LIST list)` deallocates the storage
> needed for a `OPRS_LIST`. After a `OPRS_LIST` has been passed to
> `FREE_OPRS_LIST` it may not be used in any subsequent ιlistPack func-
> tion. `list` is the list which is to be freed.

## G.3.3   Placing Elements in a List

ιlistPack provides many different ways of inserting elements into `OPRS_LISTs`.
The one to use depends on where in the list the user wishes to place the element.

Conventionally, every list has two ends called the `head` and the `tail`. An
element may be added at either end and thus becomes the new head or tail.

Alternatively, each list may be seen as an array of elements. The head is the
first element, the tail is the `list_length(list)-th`. Elements may be added
in the middle of a list by specifying the position before which they are to be
inserted. The position is then simply the distance from the head of the list.

### add_to_head                                      **listPack Function**

> `OPRS_NODE add_to_head (OPRS_LIST list, OPRS_NODE element)` The
> element is added before the head of the specified list, thus becoming
> the new head. For convenience, `add_to_head()` returns (`OPRS_NODE`
> `element`). This allows the user to perform operations like
>
> `save = add_to_head( list_1, read_next_element() );`
>
> or anything else. ιlistPack tries to impose no hidden restriction on
> the use of lists.

### add_to_tail                                      **listPack Function**

> `OPRS_NODE add_to_tail (OPRS_LIST list, OPRS_NODE element)` The
> element is added after the tail of the specified list and thus becomes
> the new tail. For convenience, `add_to_tail()` returns (`OPRS_NODE`
> `element`).

### insert_list_pos                                  **listPack Function**

> `OPRS_NODE insert_list_pos ( OPRS_LIST list, OPRS_NODE element,`
> `int pos)` (`OPRS_NODE element`) is inserted into (`OPRS_LIST list`)
> before an element number `int pos`. The positions of all elements
> subsequent to the inserted element are thus incremented.
>
> If `pos` is less than or equal to one, it is inserted before the head of
> `list`. If `pos` is greater than `list_length( list )`, it is added after
> the tail.
>
> For convenience, `insert_list_pos()` returns (`OPRS_NODE element`).

### replace_list                                     **listPack Function**

int replace_list ( OPRS_LIST list, OPRS_NODE old, OPRS_NODE
new)   The first appearance of OPRS_NODE old in the specified list
is replaced with (OPRS_NODE new). No other part of (OPRS_LIST
list) is affected. replace_list() returns a boolean TRUE if the
replacement succeeded and a boolean FALSE otherwise. Thus, re-
placing all occurrences of an element in a list may be done via

```
while ( replace_list( list, old, new ) )    /* null body */ ;
```

## append_list                                      listPack Function

OPRS_LIST append_list ( OPRS_LIST first, OPRS_LIST second)
append_list() adds all the elements of (OPRS_LIST second) after
the tail of (OPRS_LIST first).

The algorithm used to append lists *destroys the second list*. There-
fore, (OPRS_LIST second) may not be used in subsequent ιlistPack
functions. It may be thus desirable to use append_list() as:

```
append_list( first, copy_list(second, flag) );   /* append a copy of list 2 */
```

append_list() returns (OPRS_LIST first) to the caller. This al-
lows the following convenient operation:

```
new_list = append_list( copy_list(first, flag), l2 );    /* l2 is destroyed! */
```

Of course, these are also valid uses of append_list():

```
new_list = append_list( copy_list(first, flag), copy_list(second, flag) );
append_list( first, second );
append_list( first, append_list( second, third ) );
```

### G.3.4   Examining the Elements of a List

Once elements have been placed in a OPRS_LIST, they are accessible to the
user through any pointer pointing to them that the user has maintained. The
user does not often want to maintain his own pointers, and relies on ιlistPack
functions to return pointers to the elements in a list.

## get_list_head                                    listPack Function

OPRS_NODE get_list_head (OPRS_LIST list)

get_list_head() returns a pointer to the first element in the spec-
ified list. If the OPRS_LIST is empty, NULL is returned.

## get_list_tail                                    listPack Function

OPRS_NODE get_list_tail (OPRS_LIST list)

get_list_tail() returns a pointer to the last element in the speci-
fied list. If the OPRS_LIST is empty, NULL is returned.

### get_list_pos                                    **listPack Function**

> OPRS_NODE get_list_pos (OPRS_LIST list, int pos) get_list_pos()
> returns a pointer to the **pos-th** elements in the specified list. If the
> OPRS_LIST is empty, **pos** is less than one, or **pos** is greater than the
> length of the list, NULL is returned.

## G.3.5  Removing Elements from Lists

Elements may be removed from OPRS_LISTs with the appropriate ılistPack func-
tions. All these functions return a pointer to the element removed from the list,
so the user does not need to save the pointer before removing the element.

### get_from_head                                   **listPack Function**

> OPRS_NODE get_from_head (OPRS_LIST list) get_from_head() re-
> turns a pointer to the first element in the specified list. The element
> is removed from the list, and the next element becomes the new
> head. If the OPRS_LIST is empty, NULL is returned.

### get_from_tail                                   **listPack Function**

> OPRS_NODE get_from_tail (OPRS_LIST list) get_from_tail() re-
> turns a pointer to the last element in the specified list. The element
> is removed from the list, and the previous element becomes the new
> tail. If the OPRS_LIST is empty, NULL is returned.

### delete_list_pos                                 **listPack Function**

> OPRS_NODE delete_list_pos (OPRS_LIST list, int pos) delete_list_pos()
> returns a pointer to the **pos-th** element in the specified list. The
> element is removed from the list, and the position of subsequent el-
> ements is decremented. If the OPRS_LIST is empty, **pos** is less than
> one, or **pos** is greater than the length of the list, then NULL is re-
> turned.

### delete_list_node                                **listPack Function**

> OPRS_NODE delete_list_node (OPRS_LIST list, OPRS_NODE element)
> The first appearance of (OPRS_NODE element) in the specified OPRS_LIST
> is deleted. For convenience, (OPRS_NODE element) is returned to
> the caller, but if the list is empty or (OPRS_NODE element) is not
> present, then NULL is returned.

## G.3.6  Examining the Lists

ılistPack provides several functions which return boolean values based on in-
quiries being done upon OPRS_LISTs.

### list_length                                     **listPack Function**

int list_length (OPRS_LIST list) This function returns the number of elements in the specified list.

### in_list                                          listPack Function

int in_list ( OPRS_LIST list, OPRS_NODE element) This function returns 0 (boolean FALSE) if (OPRS_NODE element) does not appear in the specified (OPRS_LIST list). Otherwise, it returns the position of the element in the list (in the range 1..list_length(list)).

### list_empty                                       listPack Function

int list_empty (OPRS_LIST list) This function returns boolean TRUE if the specified OPRS_LIST has no element, it returns boolean FALSE otherwise.

### first_in_list                                    listPack Function

int first_in_list (OPRS_LIST list, OPRS_NODE element) It returns TRUE if the head of the OPRS_LIST is the specified (OPRS_NODE element).

### last_in_list                                     listPack Function

int last_in_list (OPRS_LIST list, OPRS_NODE element) It returns TRUE if the tail of the OPRS_LIST is the specified (OPRS_NODE element).

## G.3.7   Applying Functions to Lists

The ıC language provides the capability to pass functions as parameters to other functions. This is utilized by ılistPack to allow users to write list processing functions. Instead of having the user walk through each OPRS_LIST (i.e. with get_list_pos()) and pass each returned element to some function, the function is passed to ılistPack and calls the user function.

List processing always begins at the head of the OPRS_LIST, and proceeds towards the tail.

**'Warning:'** It is extremely dangerous to add or remove elements from a OPRS_LIST ıwhile a function is being applied to it. ılistPack can handle most of the common cases that occur when this happens, however such code should be rigorously exercised.

### for_all_list                                     listPack Function

int for_all_list ( OPRS_LIST list, (OPRS_NODE) ptr, (PFI) func) The user may have some functions like (int) func( (OPRS_NODE) ptr, OPRS_NODE element) he wishes to call with every element in the (OPRS_LIST list). for_all_list() invokes func() for the user (as often as necessary), adds up the returned values, and returns the total value to the user. (OPRS_NODE ptr) is a pointer to whatever

the user wishes to pass to the function (i.e. it is a free pointer
with which the user may play). There are two related forms of
`for_all_list`:

```
(int) for_all_2list( (OPRS_LIST) list, (OPRS_NODE) ptr, ptr2, (PFI) func )
(int) for_all_3list( (OPRS_LIST) list, (OPRS_NODE) ptr, ptr2, ptr3, (PFI) func )
```

They provide one or two additional pointers for the user convenience.
In all forms, the pointer is passed to `func()` in order, followed by a
(`OPRS_NODE element`).

### search_list <span style="float:right">listPack Function</span>

`OPRS_NODE search_list (OPRS_LIST list, (OPRS_NODE) ptr, (PFI)`
`func)` `search_list()` is useful for finding an element in a `OPRS_LIST`
which matches some criteria. The user writes a function (`int`)
`func(OPRS_NODE ptr, OPRS_NODE element)` which returns boolean
`TRUE` when it matches the criteria on an element in the specified list.
List processing stops after the first `TRUE` returns, and the element
on which `func()` succeeded is returned. `NULL` is returned if `func()`
never succeeds.

### list_equal <span style="float:right">listPack Function</span>

`OPRS_LIST list_equal (OPRS_LIST list1, OPRS_LIST list2, PFI`
`func)` This function compares two `OPRS_LIST`s, and returns a boolean
quantity indicating whether they are equivalent or not. (`PFI func`)
is a function which passes one element from each list, and returns
the boolean quantity describing their equality. The elements in each
`OPRS_LIST` are passed (in order) to `func`, until `list_equal()` deter-
mines the lists are not equal. If `func` is `NULL`, `list_equal()` simply
tests that the elements are the same.

```
int test_records( rec1, rec2 )  /* are two records equal? */
struct record *rec1, *rec2; /* both are rec pointers */
    {
    /* We define them to be equivalent iff height and weight are the same */
    return (rec1->height == rec2->height) && (rec1->weight == rec2->weight);
    }

...
    /* Print the records in the first list, and if the second list is
   not the same, print the records in that list */
   for_all_list( rec_list1, stdout, print_record );
   if (! (list_equal( rec_list1, rec_list2, test_records )))
   for_all_list( rec_list1, stdout, print_record );
```

## G.3.8   Changing the Order of the Elements

Since one of the fundamental tasks of programming entails sorting, 1listPack
provides primitives to reorder a list of elements to the user specifications.

### `reverse_list`                                   **listPack Function**

> `OPRS_LIST reverse_list (OPRS_LIST list)` The order of elements
> in the `OPRS_LIST` is reversed. For example, the tail becomes the first
> element, and the head becomes the last one. (`OPRS_LIST list`) is
> returned for the convenience of the caller.

### `sort_list`                                      **listPack Function**

> `OPRS_LIST sort_list (OPRS_LIST list)` The elements of the spec-
> ified `list` are sorted in the ascending order of their tag values. The
> `list` is returned so the user can do:
>
> `new = reverse_list( sort_list( eval_list( copy_list(my_list, flag),`
> `my_func)) );`

### `sort_list_func`                                 **listPack Function**

> `OPRS_LIST sort_list_func (OPRS_LIST list, PFI func)` This func-
> tion sorts a `OPRS_LIST` with more complicated criteria than can easily
> be specified by an integer tag variable. The user specified functions,
> `func( element_a, element_b )`, return boolean `TRUE` if they are
> ordered properly (i.e. element_a precedes element_b) and `FALSE` oth-
> erwise. `sort_list_func()` returns (`OPRS_LIST list`) for the user
> convenience. *Note:* For efficiency reasons, `func()` should return
> `TRUE` if the elements are equivalent.

### `merge_sort_list_func`                           **listPack Function**

> `OPRS_LIST merge_sort_list_func (OPRS_LIST list, PFI func)` Iden-
> tical to `sort_list_func()`, but uses merge sort rather than bubble
> sort.

## G.3.9   Marking Current Position in a OPRS_LIST

A spare pointer in the `OPRS_LIST` header record can be used to point at a
particular place in a `OPRS_LIST`. This feature may be used to walk forwards or
backwards along a `OPRS_LIST`, doing incremental searches or loop iteration.

Note that there is only one such pointer in each `OPRS_LIST`. Thus the use
of any function which changes that pointer will affect any other software that
uses it. In particular, it means that the `loop_through_list` macro may not be
used recursively on the same `list`. If this is necessary, use the `for_all_list()`
function or `for_list_loop()` macro.

### `get_list_next`                                  **listPack Function**

OPRS_NODE get_list_next (OPRS_LIST list, OPRS_NODE current)
This function advances the current OPRS_LIST element to be the one
immediately following (OPRS_NODE current), and that element is
returned.

If current is not in (OPRS_LIST list), NULL is returned. If current
is NULL, the head of the list becomes the current element, and is re-
turned.

This function is particularly useful for iterating on two lists simul-
taneously:

```
ptr1 = get_list_next( list1, NULL );    /* get first element of list1 */
ptr2 = get_list_next( list2, NULL );    /* get first element of list2 */

while ( ptr1 != NULL && ptr2 != NULL){    /* loop on each element */

    printf( "Nodes %x and %x\n", ptr1, ptr2 );    /* code to deal with elements */

    ptr1 = get_list_next( list1, ptr1 );    /* now get next from list1 */
    ptr2 = get_list_next( list2, ptr2 );    /* now get next from list2 */
    }
```

## get_list_prev                              listPack Function

OPRS_NODE get_list_prev ( OPRS_LIST list, OPRS_NODE current)
This function retreats the current OPRS_LIST element to be the one
immediately preceding (OPRS_NODE current), and that element is
returned.

If current is not in (OPRS_LIST list), NULL is returned. If current
is NULL, the tail of the list becomes the current element, and is re-
turned.

## get_list_next_func                         listPack Function

OPRS_NODE get_list_next_func ( OPRS_LIST list, OPRS_NODE current,
OPRS_NODE ptr, PFI func)   This function advances the current
OPRS_LIST element to be the first one in (OPRS_LIST list) af-
ter (OPRS_NODE current) upon which the function func returns
boolean TRUE. That element is returned.

If current is not in (OPRS_LIST list), NULL is returned. If current
is NULL, processing starts with the first element.

(OPRS_NODE ptr) is passed to the function, func as a free parameter.
Hence the signature for func should look like:

```
int my_function( parameter, list_element )
OPRS_NODE parameter, list_element;    /* or any other pointer type */
    {
    }
```

This function is useful for processing selected elements in a `OPRS_LIST`:

```
ptr = NULL;
while ((ptr = get_list_next_func( list, ptr, NULL, my_function )) != NULL)
{

    ...    /* process each element that my_function found */

    }
```

## `loop_through_list`         **listPack Macro**

void `loop_through_list` (`OPRS_LIST` list, type pointer, type)
This macro is provided by ılistPack to serve as a `for` loop in normal programming. A ıC-language `for` statement is generated and iterates `pointer` through the elements in (`OPRS_LIST` list).

Note that only one `loop_through_list()` macro can be active on a given list at any time. If recursion is desired, use the `for_list_loop()` macro.

The last parameter is a type specification, and is used by the macro to generate the proper recasting instructions to make both the ıC-compiler and ılint(1) happy.

```
struct foo {
    int x, y;
} * temp;        /*temp is a pointer */
OPRS_LIST my_list;

...

        /* for each pointer in the list */
loop_through_list( my_list, temp, struct foo *){
            /* process the element */
    printf( "%d, %d\n", temp->x, temp->y );
    }
```

## `for_list_loop`         **listPack Macro**

void `for_list_loop` (`OPRS_LIST` list, `OPRS_LIST` temp, type pointer, type)  This macro is provided by ılistPack to serve as a `for` loop in normal programming. A ıC-language `for` statement is generated and iterates `pointer` through the elements in (`OPRS_LIST` list).

The last parameter is a type specification, and is used by the macro to generate the proper recasting instructions to make both the ıC-compiler and ılint(1) happy.

It differs from `loop_through_list()` in that it allows recursive access to the list, and thus requires a local variable. The above example would read.

```
OPRS_LIST loop_temp;
        /* for each pointer in the list */
for_list_loop( my_list, loop_temp, temp, struct foo *){
            /* process the element */
    printf( "%d, %d\n", temp->x, temp->y );
    }
```

# Appendix H

# Lisp and Lisp-like Functions

There is no a priori reason for the OPRS Development Environment to provide Lisp like functions, or a Lisp environment. However, for upward compatibility reasons, and to allow the user familiar with Lisp lists manipulation functions to use lisp functions in the OPRS Kernel, we added various functions and data structures in the OPRS Development Environment.

In this chapter, we describe the different functions and the mechanisms which are provided to use lispisms in the OPRS Development Environment.

## H.1   LISP_LIST

First of all, it is important to note that the overall syntax used in the OPRS Development Environment is "Lisp like", and looks like Lisp. In other words, the various expressions are currently given in a syntax which looks a lot like the Lisp one (parenthesis are certainly here to remind you of this). But it does not mean there is a lisp interpreter behind the reader. In fact, whatever is read from the keyboard, or from a file is fed in a parser (or a command interpreter) and transformed in internal structures (the details of which are of no interest to the reader). Most of these structures are accessed by the user or the kernel in a transparent way. For example, when you conclude the fact (`foo (bar 3 4)`) in the database, you cannot do a `car` on the inner expression (`bar 3 4`) to conclude let say (`foo bar`). The term (`bar 3 4`) is not a Lisp list, it is a composed term.

However, one can define Lisp like structures. They are subject to a special reader syntax. In Lisp, when the reader encounters a left parenthesis (`,` he builds a list until he finds the matching closing parenthesis `)`. In OPRS, the left and right parentheses do not create any Lisp list object in any situation. To create a `LISP_LIST` (note the Term type) (on which you will be able to apply Lisp functions such as `car`, `cdr`, `cons`), you need to use (`.` and a matching `.`). As a mnemonic, you can remember that the dot `.` is the cons operator in Lisp. The OPRS writer uses the same syntax to write `LISP_LIST`. So, if you

conclude (`foo (.  bar 3 4 .)`) in the database, you have a (`.   bar 3 4 .`) term which is a `LISP_LIST`, not a composed term, and you can apply to it any function requiring a `LISP_LIST` as an operand.

One interesting thing in Lisp is the lack of type. The various elements of a list are typed internally by the Lisp kernel, but you can cons anything with anything, and do an eval on it if you wish to. In OPRS, it is the same, most objects are typed when read and the syntax defines exactly what can go where. For example, the expression (`foo (1 2 3 4)`) is not legal because `1` is recognized as an integer and not as a symbol. However, one may want to manipulate integer lists, or lists of objects unknown to the reader a priori (probably internal structures such as OP-instances, intentions, goals, etc). In this case, one uses `LISP_LIST`s: because of their lack of type requirement, they can basically handle whatever terms. When you build a list like in the expression: (`foo (.  1 2 3 4 .)`) you actually build a `LISP_LIST` of terms which are all integers. In fact, this is how you can build `LISP_LIST` of `LISP_LIST`s... The expression (`foo (.  (.  1 2 3 4 .)  2 3 4 .)  `) is a predicate with one term, a `LISP_LIST` which contains terms, the first of which is a `LISP_LIST` itself. Even if it is not obvious at a first glance, a list of terms is the only list you can build from the reader. All other lists and the objects they contain are built by internal operations. For example, the `soak` meta fact contains a `LISP_LIST` of OP-instances.

Keep in mind that the elements of a `LISP_LIST` are `Terms`.

The empty LISP OPRS_LIST is defined as follow in the file *'oprs-type-pub.h'*.

```
extern const L_List l_nil;  /* The lisp nil constant. */
```

## H.2   Standard Lisp Functions

Standard lisp functions are predefined evaluable functions (see [Predefined Evaluable Functions], §6.1, page 95) used to manipulate `LISP_LIST`.

### length                                          Evaluable Function

`INTEGER length (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### select-randomly                                 Evaluable Function

`Any Term select-randomly (LISP_LIST)`  See [Predefined Evaluable Functions], §6.1, page 95.

### cons                                            Evaluable Function

`LISP_LIST cons (Any Term, LISP_LIST)`  See [Predefined Evaluable Functions], §6.1, page 95.

### first                                           Evaluable Function

`Any Term first (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### car
**Evaluable Function**

`Any Term car (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### cdr
**Evaluable Function**

`LISP_LIST cdr (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### caar
**Evaluable Function**

`Any Term caar (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### cadr
**Evaluable Function**

`Any Term cadr (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### cdar
**Evaluable Function**

`LISP_LIST cdar (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### cddr
**Evaluable Function**

`LISP_LIST cddr (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### caaar
**Evaluable Function**

`Any Term caaar (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### cadar
**Evaluable Function**

`Any Term cadar (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### cdaar
**Evaluable Function**

`LISP_LIST cdaar (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### cddar
**Evaluable Function**

`LISP_LIST cddar (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

### caadr
**Evaluable Function**

`Any Term caadr (LISP_LIST)` See [Predefined Evaluable Functions], §6.1, page 95.

## caddr                                    **Evaluable Function**

`Any Term caddr (LISP_LIST)` See [Predefined Evaluable Functions],
§6.1, page 95.

## cdadr                                    **Evaluable Function**

`LISP_LIST cdadr (LISP_LIST)`   See [Predefined Evaluable Func-
tions], §6.1, page 95.

## cdddr                                    **Evaluable Function**

`LISP_LIST cdddr (LISP_LIST)`   See [Predefined Evaluable Func-
tions], §6.1, page 95.

## second                                   **Evaluable Function**

`Any Term second (LISP_LIST)` is the second function. It is defined
for 1 term `LISP_LIST`. It returns the second (or cadr) `Any Term` of
the `LISP_LIST`.

## reverse                                  **Evaluable Function**

`LISP_LIST reverse (LISP_LIST)` is the reverse function. It is de-
fined for 1 term `LISP_LIST`. It returns the reverse of the `LISP_LIST`.

## second                                   **Evaluable Function**

`Any Term second (LISP_LIST)`   See [Predefined Evaluable Func-
tions], §6.1, page 95.

# Appendix I

# Examples

Several examples are included in the OPRS distribution. We shall describe them in this chapter in an increasing complexity order. All examples are available in the *'data'* sub directory of the OPRS Development Environment distribution. Feel free to look at these examples and to play with the various files contained in this directory (make a copy if you need to modify them).

## I.1  Message Example

This example presents one important feature of OPRS: the message passing mechanism. To run it, you create three OPRS, or X-OPRS Kernels named `foo`, `bar` and `boo`, in which you load *'data/foo.inc'*, *'data/bar.inc'* and *'data/boo.inc'* respectively.

### I.1.1  Message Example OPs

Here are the procedures from one of the OP files provided for this demo. It can be found in the file *'bar.opf'*.

- **—test message—**

  A graphic OP.

  **Invocation:** `(BAR)`

  **Context:** `()`

  **Effects:** `()`

  **Documentation:** `"This OP is used in the foo, bar and boo demo to send a message to the 'next' OPRS when it receives one of its name."`

The other OP files are similar.

## I.2    Test Examples

Other OPs files are provided in the standard distribution as examples. Most of these examples are located in the *'data/test'* directory.

### I.2.1    Wait OPs

These OPS can be found in the file *'test/wait.opf'*.

- **—test wait—**

  A graphic OP.

  **Invocation:** `(TEST-WAIT $N)`

  **Context:** `()`

  **Documentation:** `"Just a test on the wait temporal operator."`

- **—test2 @—**

  A graphic OP.

  **Invocation:** `(! (FOO1 $X))`

  **Context:** `()`

  **Documentation:** `"Just to illustrate a test on variables."`

### I.2.2    LISP_LIST manipulation OPs

These OPS can be found in the file *'example-cons.opf'*.

- **EXAMPLE**

  A graphic OP.

  **Invocation:** `(! (REVERSE $LIST))`

  **Context:** `()`

  **Documentation:** `"Reverse a LISP_LIST list of variable using CONS, CAR, CDR."`

### I.2.3    Fibonacci OPs

These OPS can be found in the file *'fib.opf'*.

- **—Fibonacci 2—**

  A graphic OP.

  **Invocation:** `(! (FIBONACCI2 $N $RESULT))`

  **Context:** `()`

  **Effects:** `((=> (FIBONACCI2 $N $RESULT)))`

**Documentation:** "This OP computes the Fibonacci of $n and remember the
previous value.
By the way, FIBONACCI2 should NOT be declared as a op_predicate."

- **—Fibonacci—**
A graphic OP.

  **Invocation:** (! (FIBONACCI $N $RESULT))
  **Context:** ()
  **Documentation:** "This OP computes the Fibonacci of $n."

- **—Print Fibonacci 2—**
A graphic OP.

  **Invocation:** (! (PRINT-FIBONACCI2 $X))
  **Context:** ()
  **Documentation:** "This OP just looks for the fibonacci of $x and prints the result."

- **—Print Fibonacci—**
A graphic OP.

  **Invocation:** (! (PRINT-FIBONACCI $X))
  **Context:** ()
  **Documentation:** "This OP just looks for the fibonacci of $x and prints the result."

## I.2.4 Parallel Fibonacci OPs

These OPS can be found in the file *'fib-par.opf'*.

- **—Fibonacci—**
A graphic OP.

  **Invocation:** (! (FIBONACCI $N $RESULT))
  **Context:** ()
  **Effects:** ()
  **Documentation:** "This OP computes the Fibonacci of $n."

- **—Print Fibonacci—**
A graphic OP.

  **Invocation:** (! (PRINT-FIBONACCI $X))
  **Context:** ()
  **Effects:** ()
  **Documentation:** "This OP just looks for the fibonacci of $x and prints the result."

# Appendix J

# How to Install the OPRS Development Environment

This section is completely out of date and need to be updated since I moved the code in Open Source...

The installation procedure is different (although very similar), according to the license you have. One general comment is that most Makefile are written for GNU Make [FSF93], and may not work with other `make` program. Similarly, a number of tools (such as Perl [Wal93]) may be required to build the system (this is particularly true for the source distribution).

## J.1   Unpacking the Distribution

1. Choose a directory where you want to install the distribution. `# cd /usr/local/oprs (or your preferred directory)`

2. Unpack the distribution with: `# tar xvf <tape-dev>` or, if you got the distribution as a compressed tar file: `# gnutar zxvf <dist-file>`

## J.2   Description of the Distribution

Assuming OPRS Development Environment is installed in the directory *'/usr/local/oprs/'* you will find in it the directories: *'bin'*, *'lib'*, *'include'*, *'pub_src'*, *'data'*, *'doc'*, *'util'*, *'app-defaults'*, *'site.make'*, *'contrib'* and *'demo'* (*'src'* if you have a source license). If your distribution contains more than one architecture distribution, the *'bin'* and *'lib'* directory as well as the *'site.make'* file (which contain machine dependant stuff) will be located in a `$TARGET` directory (as defined by tcsh, i.e sun4 for SunOS 4.1.3, sparc for Solaris 2.x, VxWorks, etc).

According to your license, some of these directories may be present or not (if you do not have a source license, the *'src'* directory is not present).

Here is an explanation of what you will find in each of these directories (after installation if you have a source license):

**‘bin’** contains all the binaries. If your license is valid for more than one CPU type, the various binaries (in *‘bin’* and *‘lib’* should be in their `$TARGET` directory.

> If you plan to use OPRS Development Environment, your `PATH` environment variable should contain the appropriate bin directory. Consequently, you should add the following code to your *‘.login’* or *‘.cshrc’* file.

```
setenv OPRSDIR /usr/local/oprs
setenv PATH "${PATH}:${OPRSDIR}/${TARGET}/bin"
```

> Here is a list of the files contained in this directory:

> **‘oprs’** is the OPRS Kernel program.

> **‘oprs-server’** is the OPRS-Server program.

> **‘xoprs’** is the X-OPRS Kernel program.

> **‘mp-oprs’** is the Message Passer program.

> **‘oprs-cat’** is a program which is used by the X-OPRS Kernel.

> **‘op-editor’** is the OP Editor program.

**‘lib’** contains a number of libraries and relocatable files which you may need to build your application and to allow it to connect to the Message Passer.

> If your license is valid for more than one CPU type, the various binaries/libraries should be in the `$TARGET` directory.

> The relocatable are given in two format, one which can be linked to some C++ functions, in which case the `main` is not defined to allow a C++ `main` to be used (the C++ main performs some initialization required by C++ functions).

> > **‘libmp.a’** is a library you can use if you want your own program to connect to the Message Passer.

> > **‘libopaque.a’** is a library which can be necessary if the following symbols are missing: `user_time, user_time_note, user_call_from_parser`.

> > **‘oprs-relocatable’** is a relocatable OPRS Kernel, you can use to build your own version of the OPRS Kernel (i.e. containing your own evaluable functions and predicates).

> > **‘xoprs-relocatable’** is a relocatable X-OPRS Kernel, you can use to build your own version of the X-OPRS Kernel (i.e. containing your own evaluable functions and predicates).

**'c++-oprs-relocatable'** is a relocatable OPRS Kernel, you can use to build your own version of the OPRS Kernel (i.e. containing your own evaluable functions and predicates) with C++ code. In this relocatable the oprs `main` is called `oprs_main` and takes the same argument as the real `main`: `argc`, `argv` and `envp`. `oprs_main` does not return.

**' c++-xoprs-relocatable'** is a relocatable X-OPRS Kernel, you can use to build your own version of the X-OPRS Kernel (i.e. containing your own evaluable functions and predicates) with C++ code. In this relocatable the oprs `main` is called `oprs_main` and takes the same argument as the real `main`: `argc`, `argv` and `envp`. `oprs_main` does not return.

**'include'** contains include files needed to build your own version of the X-OPRS Kernel and the OPRS Kernel.

**'pub_src'** contains some source example files needed to build your own version of the X-OPRS Kernel and the OPRS Kernel.

**'user-action.c'** contains examples of how to defined your own actions and an example of a call to `declare_user_action`,

**'user-action.h'** contains definitions needed for *'user-actions.c'*.

**'user-ev-function.c'** contains examples of how to defined your evaluable functions and example of a call to `declare_user_eval_funct`.

**'user-ev-function.h'** contains definitions needed for *'user-ev-function.c'*.

**'user-ev-predicate.c'** contains examples of how to defined your own evaluable and an example of a call to `declare_user_eval_pred`.

**'user-ev-predicate.h'** contains definitions needed for *'user-ev-predicate.c'*.

**'user-external.c'** contains examples of how to call `start_kernel_user_hook` and `end_kernel_user_hook`.

**'user-external.h'** contains definitions needed for *'user-external.c'*.

**'user-external_f.h'** contains function definitions needed for *'user-external.c'*.

**'doc'** contains the documentation of the OPRS Development Environment. Here is a list of the files contained in this directory:

**'dir'** Info top level dir for the documentation.

**'oprs'** is the oprs master info file.

**'oprs-[0-9][0-9]'** are the oprs info files.

**'oprs.dvi'** is the oprs.dvi file (dvi format).

**'oprs.ps'** is the postscript version of the documentation.

**'fig'** is a directory containing the postscript figures of the manual.

**'contrib'** contains some code (source) contributed by other people. For now, one can find a library which ease the extraction of Terms parameters from TermLists (as passed to evaluable functions, predicates and actions), but ans the building of Terms to return value from these functions.

**'site.make'** contains a the Makefile variable used to produce the OPRS Development Environment you are using. This can be usefull if you have to produce your own OPRS Kernels.

**'util'** contains a number of utilities, shells, perl scripts and C programs to build OPRS.

> **'Makefile'** a makefile...
>
> **'ad2c'** is a shell script to transform a *'.ad'* resource file in a string suitable for inclusion in a *'.c'* file (to be defined as a fall back resources string).
>
> **'update-inc-file'** is a shell script which upgrade your *'.inc'* file from the pre 1.1.1 version format to the newest format.
>
> **'mkdep'** is a perl script which can be used to create dependencies files.
>
> **'vw-script'** is an exemple of a VxWorks script to load and run a OPRS Kernel, a Message Passer or a Message Passer Killer.
>
> **'lex-includer.l'** is a lex program which is used during the OPRS compilation to build the various lex and yacc grammar files. This file is only present in the source distribution.

**'data'** contains a number of data files examples (*'.opf'* files, *'.inc'* files, *'.db'* files, *'.sym'* files, etc)

**'data/test'** contains a number of test files (*'.opf'* files, *'.inc'* files, *'.db'* files, *'.sym'* files, etc)

**'app-defaults'** contains some application default files for the X-OPRS Kernel application (*'XOprs'*) and the OP Editor application (*'Op-editor'*). Both applications are Xt based, and can be customized using the resources file [Sta91a, Sta91b, Fou]. See [OP Editor Motif Widgets Hierarchy and Resources], §L.3, page 389 and [X-OPRS Motif Widgets Hierarchy and Resources], §L.2, page 387 for a list of the widget and resources available. In any case, you should either put this directory in your `XFILESEARCHPATH` environment variable as in:

```
setenv OPRSDIR /usr/local/oprs
if ( $?XFILESEARCHPATH ) then
    setenv XFILESEARCHPATH "${XFILESEARCHPATH}:${OPRSDIR}/%T/%N%C"
else
    setenv XFILESEARCHPATH "${OPRSDIR}/%T/%N%C"
endif
```

The `%C` suffix is very important if you want to take advantage of the multi language support of OPRS (see [Xt/Motif Widgets Hierarchy and Resources], §L, page 385).

or copy these two files in your own app default directory pointed by the environment variable `XAPPLRESDIR` (in this case, remove the '.*ad*' suffix).

```
setenv  XAPPLRESDIR ${HOME}/X/app-defaults/
```

*'demo'* contains some demonstration applications. In particular it contains the truck-demo.

*'src'* contains all the source of the OPRS Development Environment. This directory is only present when you have a source license.

## J.3   Installation for Demonstration License

The demonstration license is basically provided to "play" with the OPRS Development Environment. Therefore, some critical features are disabled to prevent the user from making a real use of the kernel. For example, the garbage collector is modified in such a way that the kernel will grow indefinitely after a period of time, so do not be surprised if you run out of swap space or if your application is becoming slower and slower, this is due to the swapping activity.

Moreover, you may not get the relocatable files... As a result, you will not be able to include your evaluable functions or predicates in the kernels.

## J.4   Installation for Binary License

Binary license still comes with some sources. Some include files are provided, and some examples of C code are also given to help the user define his own evaluable predicates or functions. In fact, when you get a binary license, you get relocatable executables which must be linked to other object files to be really executable.

## J.5   Installation for Source License

If you have a source license, you have the source of all the OPRS Development Environment. The distribution contains the *'src'* directory. You may need to compile and install the OPRS Development Environment.

For this go in the oprs directory and create a *'build'* directory at the same level than the *'src'* directory. Cd in the *'build'* directory and then call the *'util/mkoprsbintree'* utility. Then check the site.make file to customize it (check the installation directory for example) and then do a:

```
make depend
make
make install
```

The `make depend` command will create the dependencies between the sources files... This step is really require if and only if you plan to modify part of the sources, and you want then to recompile only the appropriate files. Nevertheless, it is recommended to do it.

The `make` command will build the executables, libraries, etc.

The `make install` command will install the various programs, libraries, defaults files, data, include in the appropriate directories.

Note that you may have some problem while generating the lex grammar C files. For example, on some Sun OS version (4.1.3 for example), or under Ultrix (4.2a), the lex program fails to handle such a large lex grammar. If this is the case, you should either request a revised version of lex from th OS vendor or generate the C files on another machine. For this, you may defined something like:

```
LEX = rsh <other-host> lex
```

where ¡other-host¿ is a machine which is able to parse the OPRS lex grammar. Do not use flex as a lex replacement. This will not work (unless you hack a fair amount of the grammar and the macros).

# Appendix K

# Grammar Used in the OPRS Development Environment

## K.1 Syntaxic Grammar Used in the OPRS Development Environment

Here is the yacc-style grammar used in the different parsers of the OPRS Development Environment.

```
goal:
        gtexpr
        ;

fact:
        gexpr
        ;

invocation:
            gmexpr
            ;

context:
        |
          gmexpr
        ;

setting:
        |
```

375

```
         gmexpr
      ;

properties:
      | OP_TK properties_list CP_TK
      ;

properties_list:
       | properties_list property
      ;

documentation:
      | QSTRING_TK
      ;

effects:
      | list_par_gtexpr
      ;

action:
           action_expr
      ;

action_expr:
      OP_TK SPEC_ACT_TK  variable
             simple_action CP_TK
      | OP_TK SPEC_ACT_TK  OP_TK var_list CP_TK
             simple_action CP_TK
      | simple_action
;

simple_action:
      OP_TK   function
             term_list CP_TK
;

op_name:
      SYMBOL_TK
      ;


top: OP_TK DEFOP_TK op_name
             fields_list
      CP_TK
;
```

```
fields_list: invocation_field other_fields_list

other_fields_list: field
           | other_fields_list field
;

field: body_field
     | action_field
     | context_field
     | setting_field
     | properties_field
     | documentation_field
     | effects_field
;

invocation_field: TFT_INVOCATION_TK invocation
;

context_field: TFT_CONTEXT_TK context
;

setting_field: TFT_SETTING_TK setting
;

properties_field: TFT_PROPERTIES_TK properties
;

documentation_field: TFT_DOCUMENTATION_TK documentation
;

effects_field: TFT_EFFECTS_TK effects
;

action_field: TFT_ACTION_TK action
;

body_field: TFT_BODY_TK  body
;

body: OP_TK list_inst CP_TK
;

list_inst:
        | list_inst inst
;
```

```
inst: top_gtexpr
    | if_inst
    | while_inst
    | do_inst
    | par_inst
    | comment
    | goto_inst
    | label_inst
    | break_inst
;

top_gtexpr: gtexpr
;

comment: COMMENT_TK
;

goto_inst: GOTO_TK SYMBOL_TK
;

label_inst: LABEL_TK SYMBOL_TK
;

break_inst: BREAK_TK
;

if_part_inst: gtexpr list_inst
| gtexpr list_inst ELSE_TK list_inst
| gtexpr list_inst ELSEIF_TK if_part_inst
;

if_inst: OP_TK IF_TK if_part_inst CP_TK
;

while_inst: OP_TK WHILE_TK gtexpr list_inst CP_TK
;

do_inst: OP_TK DO_TK list_inst WHILE_TK gtexpr CP_TK
;

par_inst: OP_TK PAR_TK body_list CP_TK
;

body_list:
        | body_list body
;
```

```
gmexpr: lmexpr
        | mexpr
        ;

lmexpr:
        OP_TK META_NOT_TK gmexpr CP_TK
        | OP_TK  META_OR_TK list_gmexpr CP_TK
        | OP_TK META_AND_TK list_gmexpr CP_TK
        ;

list_gmexpr:
        gmexpr
        | list_gmexpr gmexpr
        ;

mexpr:
        gexpr
        | gtexpr
        ;

gtexpr:
        ltexpr
        | texpr
        ;

ltexpr:
        OP_TK NOT_TK gtexpr CP_TK
        | OP_TK OR_TK list_gtexpr CP_TK
        | OP_TK AND_TK list_gtexpr CP_TK
        ;

list_par_gtexpr:
        OP_TK list_gtexpr CP_TK
        | OP_TK CP_TK
        ;

list_gtexpr:
        gtexpr
        | list_gtexpr gtexpr
        ;

texpr:
        OP_TK temp_op gexpr CP_TK
        ;
```

```
temp_op:
         WAIT_TK
       | ACHIEVE_TK
       | MAINTAIN_TK
       | PRESERVE_TK
       | TEMP_CONCLUDE_TK
       | RETRACT_TK
       | TEST_TK
       ;

gexpr:
        lexpr
       | expr
       | variable
       ;

lexpr:
       OP_TK NOT_TK OP_TK OR_TK  list_gexpr CP_TK CP_TK
      | OP_TK  NOT_TK OP_TK AND_TK  list_gexpr CP_TK CP_TK
      | OP_TK OR_TK list_gexpr CP_TK
      | OP_TK AND_TK list_gexpr CP_TK
      ;

list_gexpr:
       gexpr
      | list_gexpr gexpr
      ;

expr:
       OP_TK predicate term_list CP_TK
      |  OP_TK NOT_TK OP_TK predicate
            term_list CP_TK CP_TK
      |  OP_TK NOT_TK OP_TK NOT_TK OP_TK
            predicate term_list CP_TK CP_TK CP_TK
      ;

predicate:
       SYMBOL_TK
      ;

function:
       SYMBOL_TK
      ;

term_list:
      | term_list term
```

```
        ;

file_name: QSTRING_TK
        ;

term:
        INTEGER_TK
        | POINTER_TK
        | REAL_TK
        | QSTRING_TK
        | SYMBOL_TK
        | variable
        | OP_TK function term_list CP_TK
        | OP_TK var_list CP_TK
        | OP_LISP_TK term_list CP_LISP_TK
        | OP_ARRAY_TK term_list CP_ARRAY_TK
        | gtexpr
        | lexpr
        ;

var_list:
        variable
        | var_list variable
        ;

variable:
        LOGICAL_VAR_TK
        | PROGRAM_VAR_TK
        ;

property:
        OP_TK prop_name term CP_TK
        ;

prop_name:
        SYMBOL_TK
        ;
```

# K.2  Lexical Grammar Used in the OPRS Development Environment

Here is the lex-style token grammar used in the different parsers of the OPRS Development Environment.

```
ws        [ \t]+
```

```
real       [-\+]?[0-9]+\.[0-9]*
exp        [eE][-\+]?[0-9]+
integer    [-\+]?[0-9]+
pointer    0x[0123456789abcdefABCDEF]+
comment    ;.*$
qstring    \"([^"]|\\[")*\"
id         ([^ \t\n~0123456789"&\(\)\$@;\|\.[:\]][^ \t\n"\(\)\]]*)
numberid   ([^ \t\n"&\(\)\$@;\|\.\]][^ \t\n"\(\)\]]*)
idbar      (\|[^|]+\|)
nl         \n
or         V
and        &
op         \(
cp         \)
opar       \[
cpar       \]
oplisp     \(\.
cplisp     \.\)
lvar       \${numberid}
pvar       @{numberid}
gpvar      @@{numberid}


{ws}                    ;
:[Ii][Nn][Vv][Oo][Cc][Aa][Tt][Ii][Oo][Nn]
        {return TFT_INVOCATION_TK;}
:[Bb][Oo][Dd][Yy]
        {return TFT_BODY_TK;}
:[Cc][Oo][Nn][Tt][Ee][Xx][Tt]
        {return TFT_CONTEXT_TK;}
:[Ss][Ee][Tt][Tt][Ii][Nn][Gg]
        {return TFT_SETTING_TK;}
:[Pp][Rr][Oo][Pp][Ee][Rr][Tt][Ii][Ee][Ss]
        {return TFT_PROPERTIES_TK;}
:[Dd][Oo][Cc][Uu][Mm][Ee][Nn][Tt][Aa][Tt][Ii][Oo][Nn]
        {return TFT_DOCUMENTATION_TK;}
:[Ee][Ff][Ff][Ee][Cc][Tt][Ss]
        {return TFT_EFFECTS_TK;}
:[Aa][Cc][Tt][Ii][Oo][Nn]
        {return TFT_ACTION_TK;}


[Dd][Ee][Ff][Kk][Aa]
        {return DEFOP_TK;}
\/\/
        {return PAR_TK;}
[Ww][Hh][Ii][Ll][Ee]
```

```
        {return WHILE_TK;}
[Dd][Oo]
        {return DO_TK;}
[Ii][Ff]
        {return IF_TK;}
[Ee][Ll][Ss][Ee]
        {return ELSE_TK;}

{comment}
        {return COMMENT_TK;}

[Aa][Nn][Dd]
        {return META_AND_TK;}
[Oo][Rr]
        {return META_OR_TK;}
[Nn][Oo][Tt]
        {return META_NOT_TK;}

^\.\n
        {return RESET_DOT_TK;}

{real}                  |
{real}{exp}
        {return REAL_TK;}
{pointer}
        {return POINTER_TK;}
{integer}
        {return INTEGER_TK;}
{qstring}
        {return QSTRING_TK;}

||              {return OR_TK;}
&               {return AND_TK;}
~               {return NOT_TK;}

{lvar}          {return LOGICAL_VAR_TK;}
{pvar}          {return PROGRAM_VAR_TK;}
{gpvar}         {return PROGRAM_VAR_TK;}

=>      |
[Cc][Oo][Nn][Cc][Ll][Uu][Dd][Ee]
        {return TEMP_CONCLUDE_TK;}
~>      |
[Rr][Ee][Tt][Rr][Aa][Cc][Tt]
        {return RETRACT_TK;}
!       |
```

```
[Aa][Cc][Hh][Ii][Ee][Vv][Ee]
        {return ACHIEVE_TK;}
\?          |
[Tt][Ee][Ss][Tt]
        {return TEST_TK;}
#           |
[Pp][Rr][Ee][Ss][Ee][Rr][Vv][Ee]
        {return PRESERVE_TK;}
\^          |
[Ww][Aa][Ii][Tt]
        {return WAIT_TK;}
\%          |
[Mm][Aa][Ii][Nn][Tt][Aa][Ii][Nn]
        {return MAINTAIN_TK;}

{oplisp}    {return OP_LISP_TK;}
{cplisp}    {return CP_LISP_TK;}
{opar}      {return OP_ARRAY_TK;}
{cpar}      {return CP_ARRAY_TK;}

{id}        {return SYMBOL_TK;}
{idbar}     {return SYMBOL_TK;}

{op}        {return OP_TK;}
{cp}        {return CP_TK;}
```

# Appendix L

# Xt/Motif Widgets Hierarchy and Resources

Motif and Xt allow the user to set most widget resource values using the resource file (see the X11 [Sta91b] and Xt [Sta91a] documentation for more on this topic). X-OPRS and the OP Editor takes advantage of this mechanism as much as possible. In fact, as few as possible resource values are "hard wired" in the code, and most of them are set in the *'XOprs'* and *'Op-editor'* resources files. To take full advantage of this mechanism, the user needs to know the widget hierarchy of the application. The versions of the X-OPRS and the OP Editor which are based on X11R5/Xt/Motif 1.2 can even take advantage of the new resource protocol which allows the user to interactively set resource values. The user can thus make a lot of customization using resource setting.

Since version 1.3.1, OPRS offers support for different language (French and English). Thus the proper selection of the application default file is very important. A priori, if no application file is found, the default language resources value are stored in the kernel.

For each X application (X-OPRS and the OP Editor) , the *'app-defaults'* directory contains 3 files. For example, for X-OPRS, it contains *'XOprs-fr'*, *'XOprs-en'* and *'XOprs'* (which is a link on one of the two first file). A priori, you need to tell X where are the application defaults file are located. This can be done using the variable `XFILESEARCHPATH`.

```
setenv OPRSDIR /usr/local/oprs
if ( $?XFILESEARCHPATH ) then
     setenv XFILESEARCHPATH "${XFILESEARCHPATH}:${OPRSDIR}/%T/%N%C"
else
     setenv XFILESEARCHPATH "${OPRSDIR}/%T/%N%C"
endif
```

The %C (for customization) is supported in X11 since release 5. Basically, if you have, for an application `app_name` with `app_name_class` as application

| Option String | Resource Name | Argument Kind | Resource Value |
|---|---|---|---|
| -background | *background | SepArg | next argument |
| -bd | *borderColor | SepArg | next argument |
| -bg | *background | SepArg | next argument |
| -borderwidth | .borderWidth | SepArg | next argument |
| -bordercolor | *borderColor | SepArg | next argument |
| -bw | .borderWidth | SepArg | next argument |
| -display | .display | SepArg | next argument |
| -fg | *foreground | SepArg | next argument |
| -fn | *font | SepArg | next argument |
| -font | *font | SepArg | next argument |
| -foreground | *foreground | SepArg | next argument |
| -geometry | .geometry | SepArg | next argument |
| -iconic | .iconic | NoArg | "true" |
| -name | .name | SepArg | next argument |
| -reverse | .reverseVideo | NoArg | "on" |
| -rv | .reverseVideo | NoArg | "on" |
| +rv | .reverseVideo | NoArg | "off" |
| -selectionTimeout | .selectionTimeout | SepArg | next argument |
| -synchronous | .synchronous | NoArg | "on" |
| +synchronous | .synchronous | NoArg | "off" |
| -title | .title | SepArg | next argument |
| -xnllanguage | .xnlLanguage | SepArg | next argument |
| -xrm | next argument | ResArg | next argument |

Table L.1: Xt Application Default Line Arguments and Resources

class, a `app_name_class.customization:  res-value` resource defined in your *'.Xresources'* or *'.Xdefault'* files (i.e. already defined in the resource manager of your X server), then when you launch application `app-name`, it will look for an application default file with the name %N%C, i.e. `app_name_class` concatenated with `res-value`.

So if you define `XOprs.customization:  -en`, it will load *'XOprs-en'*, while if you define `XOprs.customization:  -fr`, it will load *'XOprs-fr'*. If none are defined, it will load *'XOprs'*. This mechanism is the same for the `Op-editor`. This mechanism will allow you to automatically select the language of the interface.

Note that each application file contains a `*.language` resource which is used by the application to then select the proper string when these are not defined a X resources.

# L.1    Xt Command Line Arguments

There are a number of arguments and resources which are defined by default for any Xt based application. These are indeed available for the X-OPRS application and the OP Editor application. Table L.1 presents these arguments. See [Sta91a] for more information on this subject.

Example:

```
op-editor -iconic
xoprs -bg blue
```

# L.2    X-OPRS Motif Widgets Hierarchy and Resources

## L.2.1    How to Connect your Own Widget in X-OPRS

You can if you want connect you own widget or widget tree to the X-OPRS one. They will be treated by the X-OPRS Kernel as any other widget. You can then manage them or unmanage them from your OPs. However, you should not make active wait (by starting recursively a Xt Main Loop...) or you will most likely block the X-OPRS Kernel.

**x_oprs_top_level_widget**                              **Kernel Variable**

> **extern Widget x_oprs_top_level_widget** see [Important Variables], §G.1.2, page 338

```
void start_kernel_user_hook()
{
 intention_scheduler = &intention_scheduler_time_sharing;
 create_my_widget_tree(x_oprs_top_level_widget);
}
```

## L.2.2    X-OPRS Resources

Specific X-OPRS resources can be set by the user:

```
XOprs.fontList: variable=variable_cs,\
fixed=default_cs,\
variable=text_title_cs,\
-adobe-helvetica-bold-r-normal-*-24-*-*-*-*-*-*-*=title_cs,\
fixed=text_cs,\
-adobe-helvetica-bold-r-normal-*-*-100-*-*-*-*-*-*=node_cs,\
6x12=edge_cs
XOprs.defaultFontList: variable=variable_cs,\
fixed=fixed_cs
XOprs.ipX:              10
```

```
XOprs.ipY:               50
XOprs.ipWidth:           80
XOprs.ctxtX:             10
XOprs.ctxtY:             150
XOprs.ctxtWidth:         60
XOprs.setX:              10
XOprs.setY:              250
XOprs.setWidth:          60
XOprs.effX:              10
XOprs.effY:              350
XOprs.effWidth:          60
XOprs.propX:             10
XOprs.propY:             400
XOprs.propWidth:         60
XOprs.docX:              10
XOprs.docY:              450
XOprs.docWidth:          60
XOprs.actX:              310
XOprs.actY:              50
XOprs.actWidth:          60
XOprs.bdX:               310
XOprs.bdY:               50
XOprs.bdWidth:           120
XOprs.edgeWidth:         40
```

The `XOprs.fontList` resource defines the fonts which are used in the following character set:

**title_cs** is the character set used to name the OP which appear in the top left corner.

**text_title_cs** is the character set used to name text fields like `INVOCATION, CONTEXT, etc.`

**text_cs** is the character set used for the text contained in the text field.

**node_cs** is the character set used for the name of the node.

**edge_cs** is the character set used for the edge text.

## L.2.3   X-OPRS Motif Widgets Hierarchy

The widget hierarchy for the X-OPRS program can now be obtained by issuing the command:

```
xoprs -pwt
```

Each widget is given with its path from the root and with its class name in parenthesis.

# L.3 OP Editor Motif Widgets Hierarchy and Resources

## L.3.1 OP Editor Resources

The OP Editor has a number of resource which can be set by the user:

```
Op-editor.fontList: variable=variable_cs,\
fixed=default_cs,\
-adobe-helvetica-bold-r-normal-*-24-*-*-*-*-*-*-*=title_cs,\
variable=text_title_cs,\
fixed=text_cs,\
-adobe-helvetica-bold-r-normal-*-*-100-*-*-*-*-*-*=node_cs,\
6x12=edge_cs
Op-editor.defaultFontList: variable=variable_cs,\
fixed=fixed_cs
!
! Default value for the size of the drawing area
!
Op-editor.workWidth:    2000
Op-editor.workHeight:   1500
!
! Default value for the x, y and with
! of the various OP fields. These values
! are the one used for the Text OP.
!
Op-editor.ipX:          10
Op-editor.ipY:          50
Op-editor.ipWidth:      80
Op-editor.ctxtX:        10
Op-editor.ctxtY:        150
Op-editor.ctxtWidth:    60
Op-editor.setX:         10
Op-editor.setY:         250
Op-editor.setWidth:     60
Op-editor.effX:         10
Op-editor.effY:         350
Op-editor.effWidth:     60
Op-editor.propX:        10
Op-editor.propY:        400
Op-editor.propWidth:    60
Op-editor.docX:         10
Op-editor.docY:         450
Op-editor.docWidth:     60
Op-editor.actX:         310
Op-editor.actY:         50
```

```
Op-editor.actWidth:      60
Op-editor.bdX:           310
Op-editor.bdY:           50
Op-editor.bdWidth:       120
Op-editor.edgeWidth:     40
!
! Default print command
!
! This one is for A4 if you have the most recent pbm pacopge.
Op-editor.printCommand: xwdtopnm < \%s |  pnmtops -r -w 8 -h 11.25 | lpr
! For us letter
!Op-editor.printCommand:         xwdtopnm < %s |  pnmtops -r | lpr
```

The `Op-editor.fontList` resource defines the fonts which are used in the following character set:

**title_cs** is the character set used for the name of the OP which appear in the top left corner.

**text_title_cs** is the character set used for the name of text fields like `INVOCATION`, `CONTEXT, etc`.

**text_cs** is the character set used for the text contained in the text field.

**node_cs** is the character set used for the name of the node.

**edge_cs** is the character set used for the edge text.

The `Op-editor.printCommand` resource defines the shell command used by the various print commands of the OP Editor.

The `Op-editor.workWidth` and `Op-editor.workHeight` resource defines the width and the height of the drawing canvas. These values can be changed with the "Change Drawing Size" command, See [Change Drawing Size], §16.1.4, page 232.

### L.3.2  OP Editor Motif Widgets Hierarchy

The widget hierarchy for the OP Editor program can now be obtained by issuing the command:

```
op-editor -pwt
```

Each widget is given with its path from the root and with its class name in parenthesis.

# Appendix M

# Known Problems and Things to Avoid

Although our goal is to constantly improve the OPRS Development Environment, few problems have been reported by users or found by our development team and, remain unsolved or unresolved. Most of them originate in mechanisms, tools or systems which are not directly under our control (such as X11, Motif or Unix).

In any case, bugs and problems should be reported to the following electronic mail address: `oprs-bug@ingenia.fr`. If your license agreement include software support, your problem will be taken care of as specified in the License Agreement.

## M.1    Known Problems

1. There is a terrible memory leak in X11R4 which may make the X-OPRS Kernel grows indefinitely. Here is the purify trace of this leak:

```
     16 bytes (3874 times).     Last memory leak at 0x779040
Purify (mlk): 61984 total bytes lost, allocated from:
     malloc        [p6.o, pc=0x46d4]
     XtMalloc      [Alloc.o, pc=0x1c8aec]
     XtAppAddWorkProc [NextEvent.o, pc=0x1e8898]
     register_main_loop_from_other_events [line 133, xp-main.c, pc=0x2e280]
     register_main_loop [line 140, xp-main.c, pc=0x2e2b8]
     DoOtherSources [NextEvent.o, pc=0x1e97a0]
```

2. All the modules with a graphic interface are written under X11/Motif. Both X11R5/Motif1.2 and X11R4/Motif1.1.X versions are available. But because of some weirdness with Sun OpenWindows 3.0 X server, the

X11R5 version does not work. So you should not try to use it. (A patch to Sun OpenWindows 3.0 exists which fixes this problem).

3. Due to some weird interaction between OpenLook Window Manager (olwm) and Motif application, a number of things may not work as expected. It is strongly recommended to use mwm (Motif Window Manager) as window manager, or if not possible twm (the window manager distributed with X11).

4. If Motif is not properly installed on your host, the application may not works has expected. A very common problem is the lack of file *'/usr/lib/X11/XKeysymDB'*. The symptom are very easy to identify. Whenever you start a Motif application, you get a number of message stating that some Key Sym are undefined. This problem is fixed in Motif 1.2 (there is now a variable XKEYSYMDB for this purpose).

5. Using the OP Editor under twm, the Information Dialog which pops up when OP files are loaded may not properly disappear after the load has completed.

## M.2    Things to Avoid

1. Remember that the word `test` is reserved (as are achieve, preserve, maintain, wait, conclude and retract), and cannot be used as a predicate name, nor a symbol or a function.

2. Recursive data structures are currently forbidden for many different reasons (printing problems in particular). So do not build structures which point on themselves or you will most likely break the kernel. For example, if `foo` is not an evaluable function, the following code will badly break the kernel (or make it loop...) `(!  (= @x (foo @x)))`, while `(!  (= @x (foo (val @x))))` will work, as it will use the value of `@x` instead of `@x`.

# Appendix N

# Glossary

Here is an alphabetical list of some words and concepts used in this manual and which may have a particular meaning in OPRS.

**Action:** Actions are the basic operations in OPRS. They are the means by which a OPRS Kernel or a X-OPRS Kernel modifies or acts on the external world. Actions can be as simple as printing objects, or sending messages, or complex as RPC or call to linked code. The user can define his own actions.

**Command mode:** . A OPRS Kernel is usually in Run Mode. When the OPRS-Server connects it, it switches to Command Mode (see also Run Mode).

**Conditions:** . A condition is a particular statement the OPRS Kernel is periodically checking. It can be a waiting condition (corresponding to a wait goal in a OP) , or a preserve condition (corresponding to a passive or active preserve in a OP).

**OPRS Development Environment:** The OPRS Development Environment is the set of programs provided to develop an application using procedural reasoning.

**OPRS Application Environment:** The OPRS Application Environment is the set of programs provided to run an application using procedural reasoning.

**OPRS Kernel:** See Kernel.

**Database:** The database is where OPRS stores all the static information representing the state of the world.

**Binding Environment:** Binding environments are OPRS structures which hold the binding of a set of variables in a particular context.

**Edge:** The edges of the OPs are the links between the nodes of a OP. They are labeled with goals (one goal per edge) which must be satisfied to allow the transition between the two nodes linked by the edge.

**Evaluable Function:** Evaluable functions are functions which correspond to machine executable code and are evaluated when needed. The main difference between actions and evaluable functions is that evaluable functions appear as arguments of predicates, and can be evaluated a number of times (depending of the Current/Quote setting). As a result, they should not have any side effect which repetition could lead to unexpected results. The user can define his own evaluable functions.

**Evaluable Predicate:** Evaluable predicates are predicates which are not stored explicitly in the database but are evaluated executing some machine executable code. The user can define his own evaluable predicates.

**Fact:** A fact is an expression which represents a true statement in the current world state.

**Frame:** A Frame is a synonym for binding environment (see also Binding Environment).

**Goal:** A goal is a statement and a temporal operator specifying how this statement must be treated.

**Include File:** Include files are files containing commands to be executed by a OPRS Kernel or a X-OPRS Kernel.

**Intention:** An intention or a task is a stack of procedures (calling each other) and working towards the achievement of a goal or responding to the occurrence of an event. This goal or this event are responsible for the execution of the first procedure in the stack.

**Join Node:** Join nodes are node corresponding to a synchronization point in multi threads execution. The execution can proceed from a join node, only when a number of threads equal to the number of incoming edges, reached the join node.

**OP:** OP is more of an historical name. It means *Knowledge Area*, and was used at the beginning of OPRS research probably to remind people of other *Knowledge* constructions such as *Knowledge Source* used in Black Board systems. See also *Procedure*.

**OP Editor:** The OP Editor is the program which allows the user to build his own procedures, which can then be executed by a X-OPRS Kernel or OPRS Kernel.

**OP File:** A OP file is a file containing a set of OPs. There are no particular reasons to put some OPs and other together in the same file. However, it is usually a good practice to break down the set of OPs used in a particular application into various files.

**OP Instance:** A OP instance is a OP which is "applicable", i.e. a OP for which there is a binding environment currently satisfied. The same OP can have more than one instances applicable at the same time (in which case, the binding environments are different).

**OP Library:** The OP library is the set of OP loaded in a particular OPRS Kernel or X-OPRS Kernel.

**OP Predicate:** A OP predicate is a predicate which can only be satisfied by calling OPs. It is usually a good idea to declare OP predicates as it tends to speed up the kernels (OP predicate are not checked in the database as the system knows they can only be satisfied calling OPs).

**Kernel:** The OPRS Kernel or the X-OPRS Kernel are the kernels of the OPRS Development Environment. However, the kernel part of both programs are identical. This is the reason why we refer sometimes to them as the Kernel. The Kernel is in fact the "brain" of the OPRS Development Environment. In the OPRS Kernel environment, it displays information and results in a standard tty interface. In the X-OPRS Kernel environment, it displays graphical information and results in a X11/Motif environment. Besides, this X11/Motif environment enables the user to interact with the kernel in an easier way.

**Message:** A message is a fact sent to or receives by a OPRS Kernel (in fact by any module of a OPRS application). Messages are identical to facts, except that they come from outside, i.e. they are not concluded by the OPRS Kernel itself but are received from outside.

**Motif:** Motif is a commercial product sold by Open Software Foundation. It is a popular widget set (see Xt) which can be used to build high level user interfaces with a consistent look and a uniform interaction mechanism [Fou].

**Node:** The nodes are the building blocks of OPs. They are meant to present states. When you have achieved a particular goal labeling an edge, you end up on a new node representing the new state you have reached.

**Plan:** In this documentation, we use interchangeably the word *OP*, *procedure* and even sometimes *plan*. See OP.

**Procedure:** In this documentation, we use interchangeably the word *OP*, *procedure* and even sometimes *plan*. See OP.

**OPRS-Server:** The OPRS-Server is used to start/kill OPRS Kernels and X-OPRS Kernels and to interact with the OPRS Kernels.

**Relevant OP:** Relevant OPs are OPs which "mention" a particular fact or goal in their invocation part.

**Relocatable:** A relocatable is a file containing linked code which can be re-located by the linker. The OPRS Kernel and the X-OPRS Kernel are distributed as pure executable but also as relocatable in which the final user can link his own actions, evaluable functions and predicates and set up a number of other important variables. The OPRS Development Environment comes with two kinds of relocatable, one to link C code, and the other to link C++ functions (in the later the main is called oprs_main).

**Run mode:** A OPRS Kernel is in run mode when its main loop is executing. See Command mode.

**Split Node:** A split node is the forking point in multi threads execution. When execution reaches a split node, it forks as many threads as there are out-going edges from this node.

**Thread:** Thread corresponds to an execution sequence in an intention. An intention has always at least one thread. However, due to parallel split operations in OPs, more than one thread can be active in an intention.

**Unix:** Unix is an operating system family with many members and branches.

**Variable:** A variable is a specific symbol which can be bound or not. When it is, it is bound to a particular value which depends of the current binding environment.

**X Toolkit Intrinsics:** The X Intrinsics is a library which provides high level mechanism to build user interface. However, the X Intrinsics library does not preclude any particular look and feel, or any particular manipulated object. Moreover, the X Intrinsics by itself is not enough to build a user interface, one needs to associate a widget set (such as the Athena widget set, the Motif widget, or the Olit widget set, etc) [Sta91a].

**X11:** X11 is a popular window system available on many platforms [Sta91b].

**X-OPRS Kernel:** See Kernel.

**XInfo:** XInfo is an *info* format displaying widget. It is used in the X application of the OPRS Development Environment to display the manuals and the documentation of the OPRS Development Environment.

**Xt:** See X Toolkit Intrinsics.

# General Index

# General Index

398

# Command Index

# User Commands Index

# Evaluable Function and
Action Index

# Evaluable Functions and Actions Index

# Evaluable Predicate Index

# Predicates Index

# Kernel Function Index

# Kernel Functions Index

# Variable Index

# Variables Index

# Bibliography

[Aba93] A. Abassi. Outil d'aide au diagnostic pour la supervision du systeme CAUTRA. Mémoire de Stage Ecole Nationale de l'Aviation Civile ???, ENAC, Toulouse, France, June 1993. In french.

[BG83] U. Bonollo and M. P. Georgeff. Peritus: A Knowledge Engineering Tool For The Development Of Procedural Expert Systems. Technical Report 39, Monash University, Melbourne, Australia, 1983.

[Cha93] R. Chatila. Autonomous navigation in natural environment. In *3rd International Symposium on Experimental Robotics*. ISER, Kyoto (Japan), October 1993.

[CM84] W. F. Clocksin and C. S. Mellish. Programming in prolog. Springer-Verlag, Berlin, 1984.

[Fou] Open Software Foundation. *Motif Programmer's Manual*. Open Software Foundation, Englewoods Cliffs, New Jersey.

[Fra92] Francois Felix Ingrand. *OPRS Development Environment Manual*. Francois Felix Ingrand, 20, Chemin Michoun, 31500 Toulouse France, 1992.

[FSF93] FSF. *GNU Make Manual*. Free Software Foundation, 1993.

[GB83] M. P. Georgeff and U. Bonollo. Procedural Expert Systems. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Oprlsruhe, Germany, 1983.

[Geo82] M. P. Georgeff. Procedural Control in Production Systems. *Artificial Intelligence*, 18:175–201, 1982.

[Geo84] M. P. Georgeff. An expert system for representing procedural knowledge. In J. J. Richardson, editor, *Proceedings of the Joint Services Workshop on Artificial Intelligence in Maintenance*, pages 153–170, Air Force Systems Command, Human Resources Laboratory, Brooks AFB, Texas, 1984.

[Geo85]  M. P. Georgeff. Reasoning about Procedural Knowledge. In *Proceedings of the AIAA/ ACM/NASA/IEEE Computers in Aerospace Conference*, Long Beach, California, U.S.A, 1985.

[Geo88]  M. P. Georgeff. An Embedded Real-Time Reasoning System. In *Proceedings of the 12th IMACS World Congress*, Paris, France, 1988.

[GI88]  M. P. Georgeff and F. F. Ingrand. Research on Procedural Reasoning Systems. Final Report, Phase 1, for NASA Ames Research Center, Moffet Field, California, U.S.A, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, October 1988.

[GI89a]  M. P. Georgeff and F. F. Ingrand. Decision-Making in an Embedded Reasoning System. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 972–978, Detroit, Michigan, U.S.A, 1989.

[GI89b]  M. P. Georgeff and F. F. Ingrand. Monitoring and Control of Spacecraft Systems Using Procedural Reasoning. In *Proceedings of the Proceedings of the Space Operations-Automation and Robotics Workshop*, Houston, Texas, 1989.

[GI90a]  M. P. Georgeff and F. F. Ingrand. Real-Time Reasoning: The Monitoring and Control of Spacecraft Systems. In *Proceedings of the Sixth IEEE Conference on Artificial Intelligence Applications*, Santa Barbara, California, U.S.A, March 1990.

[GI90b]  M. P. Georgeff and F. F. Ingrand. Research on Procedural Reasoning Systems. Final Report, Phase 2, for NASA Ames Research Center, Moffet Field, California, U.S.A, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, June 1990.

[GL85]  M. P. Georgeff and A. L. Lansky. Development of an Expert System for Representing Procedural Knowledge. Final Report, for NASA Ames Research Center, Moffet Field, California, U.S.A, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, December 1985.

[GL86a]  M. P. Georgeff and A. L. Lansky. A System for Reasoning in Dynamic Domains: Fault Diagnosis on the Space Shuttle. Technical Note 375, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1986.

[GL86b]  M. P. Georgeff and A. L. Lansky. Procedural Knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation*, 74:1383–1398, 1986.

[GL87] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning: An Experiment with a Mobile Robot. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, Washington, 1987.

[GLB85] M. P. Georgeff, A. L. Lansky, and P. Bessiere. A Procedural Logic. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California, U.S.A, 1985.

[GLS87] M. P. Georgeff, A. L. Lansky, and M. Schoppers. Reasoning and planning in dynamic domains: an experiment with a mobile robot. Technical note 380, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1987.

[GLS89] M. P. Georgeff, A. L. Lansky, and M. J. Schoppers. Reasoning and Planning in Dynamic Domains: An Experiment with a Mobile Robot. Technical Report 380, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1989.

[IC93a] F. F. Ingrand and V. Coutance. Procedural Reasoning versus Blackboard Architecture for Real-Time Reasoning. In *Proceedings of the 13th International Workshop on Artificial Intelligence*, Avignon, France, 1993.

[IC93b] F. F. Ingrand and V. Coutance. Real-Time Reasoning using Procedural Reasoning. Technical Report 93-104, LAAS/CNRS, Toulouse, France, 1993.

[IG90] F. F. Ingrand and M. P. Georgeff. Managing Deliberation and Reasoning in Real-Time AI Systems. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning*, Santa Diego, California, U.S.A, November 1990.

[IGL89] F. F. Ingrand, J. Goldberg, and J. D. Lee. SRI/Grumman Crew Members' Associate Program: Development of an Authority Manager. Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1989.

[IGR92] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert, Knowledge-Based Diagnosis in Process Engineering*, 7(6):34–44, December 1992. Also available as LAAS Technical Report 92-521.

[Pos89] Jef Posopnzer. *PBM Pacopge - Man page.* None, 1989.

[Rev92] F. Revillod. Une Architecture Décisionelle pour le Contrôle d'un Robot Autonome. Rapport de Stage Ecole Supérieure d'Aéronautique et de l'Espace ???, LAAS/CNRS, Toulouse, France, September 1992. In french.

[RG90]  A. S. Rao and M. P. Georgeff.  Intelligent Real-Time Network Management. In *Avignon: Expert Systems and their Applications*, Avignon, France, 1990.

[Sta91a]  MIT X Consortium Standard. *X Toolkit Intrinsics - C Language Interface*. X11 Consortium, x version 11, release 5 edition, August 1991.

[Sta91b]  MIT X Consortium Standard. *Xlib - C Language X Interface*. X11 Consortium, x version 11, release 5 edition, August 1991.

[Sti87]  Mark Stieckel.  Term Indexing Database.  Technical Report 87-5, SRI International, Menlo Park, California, U.S.A, 1987.

[Wal93]  Larry Wall. *PERL*. ORA, 1993.

[WI91]  L. Wesley and F. F. Ingrand.  Application of OPRS to Network Management System.  Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1991.

[WL93]  L. Wesley and J. Lee. Evaluation of OPRS: Final Report. Final Report, Artificial Intelligence Center, SRI International, Menlo Park, California, U.S.A, 1993.