XPCE-5.0 Release Notes

Jan Wielemaker
SWI
University of Amsterdam
Roetersstraat 15
1018 WB Amsterdam
E-mail: `jan@swi.psy.uva.nl`

January 1999

This document provides an overview of new functionality of XPCE-5. The highlights are single-file executables using embedded (image) resources, passing native Prolog data through arguments of Prolog-defined methods and the introduction of graphical tables.

University of Amsterdam

Dept. of Social Science Informatics (SWI)
Roetersstraat 15, 1018 WB Amsterdam
The Netherlands
Tel. (+31) 20 5256786

XPCE–5.0 Release Notes

Jan Wielemaker
jan@swi.psy.uva.nl

XPCE/Prolog is a hybrid environment integrating logic programming and object-oriented programming for Graphical User Interfaces. Applications in XPCE/Prolog are fully compatible across the supported X11 and Win32 (Windows 95, 98 and Windows NT) platforms.

Last updated January 18, 1998 for XPCE version 5.0

CONTENTS

1: OVERVIEW

XPCE-5.0 is the first major release since the introduction of user-defined classes. We have decided to name it such because the relation between the hosting Prolog system and XPCE has been changed radically, important features for the generation of stand-alone executables have been added, as well as a powerful **table** class for manipulating tabular layout of graphical objects.

The new host message-passing protocol is not only faster and uses less memory, it also allows passing native host (Prolog) data over XPCE-methods. In addition, a transparent protocol for representing native (recorded) Prolog data in XPCE instance-variables has been added. These features facilitate proper design of XPCE/Prolog programs with far less hassle. Defining graphical applications using XPCE classes defined in Prolog is the accepted preferred mechanism for structuring large GUI applications. Unfortunately however, as XPCE methods, even when defined in Prolog, only accepted XPCE data, either the native Prolog data had to be represented in XPCE data or a predicate instead of a method-invocation must be used. Chapter 2.1.1 provides an example of passing native Prolog data.

Graphical applications often require images, help-files and other data that is not easily expressed in Prolog source-code. XPCE-5.0 provides a generic mechanism for associating this data with the application. See chapter 4.

Tables are a common and well understood mechanism for defining 2-dimensional layout. XPCE-5.0 provides tables modelled after HTML-3, including row and column-spanning, defining rules and frames, spacing and manipulation mechanisms such as naming columns and rows, deleting, inserting and sorting rows, etc. See chapter 7.

2: CHANGES TO MESSAGE PASSING

2.1: Allow for Prolog-native data

In earlier versions, sending a message was realised by resolving the implementation, and activating it. In the new version a *goal* is created, after which actions on this goal resolve the implementation and its type-vector, type-check and allocate the arguments and finally execute the implementation. The advantage of this is two-fold. First of all, it allows the host-interface to check whether the implementation called is actually defined in the host-language itself. In this case the interface can decide not to route the call through XPCE, but instead build the proper argument vector an call the implementation directly. This is faster and, at least equally important, it avoids the requirement to represent all data passed over a XPCE method, even when implemented in the host to be converted into XPCE data. Second, the new implementation technique results in a much flatter and smaller C-stack. Not only this saves a considerable amount of memory in deeply nested method-invocation, but it also reduces the amount of stack-faults on processors using *register-windows* (e.g. SPARC)

To improve the Prolog integration even further, the class **prolog_term** and type **prolog** have been introduced. When invoking Prolog-defined methods, the type **prolog** indicates the interface that the Prolog argument should be passed directly to the implementation. If a XPCE-native argument is defined to be of type **prolog**, the Prolog term-reference is packed into an instance of the class **prolog_term**. If the context in which a prolog_term object was created terminates and the object has references, the interface will record the term into the Prolog database.

This approach results in natural behaviour. If a Prolog term is passed over methods that allow for it and reaches Prolog at some stage, Prolog receives the unmodified term. If the Prolog term is associated with a XPCE object, normally as data for an instance-variable, the instance-variable will contain a recorded copy of the term. If Prolog requests the value of this term, the term is copied back to the Prolog stacks (cf. **recorded/3**).

2.1.1: Example — Creating a tree from Prolog data

Suppose a tree is represented in Prolog using the term **node**(*Value, Sons*), where *Sons* itself is a tree and *Value* is an arbitrary Prolog term. We would like to represent this tree using a XPCE **tree** object. If a node is selected, the associated term is displayed.

Class parse_tree Defines the tree object itself. It defines the global visual characteristics of the tree. The hierarchy is represented by node objects. The root and all underlying nodes are created directly from the Prolog term that represents the hierarchy in Prolog.

Finally, an event-handler is associated with all nodes that invokes →**clicked** on the node after a left-mouse-button single click has been recognised.

```
:- pce_begin_class(parse_tree, tree).

initialise(T, Tree:prolog) :->
        send(T, send_super, initialise, parse_node(Tree)),
        send(T, direction, list),
        send(T, level_gap, 20),
        send(T, node_handler,
             click_gesture(left, '', single,
                           message(@event?receiver?node, clicked))).
:- pce_end_class.
```

Class parse_tree contains the magic. Unlike XPCE version 4, version 5 can built trees bottom-up, i.e. first creating the leaves of the tree and gradually relating the leaves using nodes higher in the hierarchy. Finally the whole hierarchy is associated to a tree object.

```
:- pce_begin_class(parse_node, node).
variable(value,  prolog,  get, "Associated value").
initialise(N, Tree:prolog) :->
        Tree = node(Value, Sons),
        send(N, send_super, initialise, new(text)),
        send(N, value, Value),
        (   Sons == []
        ->  send(N, collapsed, @nil)    % do not show [+] mark
        ;   forall(member(Son, Sons),
                   send(N, son, parse_node(Son)))
        ).
```

Set the associated value, as well as the label. ⟨*Value*⟩ is stored in the ←**value** instance-variable of type prolog. With the return of the →**slot** message, the scope in which ⟨*Value*⟩ is passed to XPCE ends, and ⟨*Value*⟩ is thus recorded into the Prolog database. Later destruction of the prolog_term object automatically frees the Prolog record.

Note that the label still needs to be translated to a type acceptable to XPCE. Passing the term directly forces the string to convert the argument to XPCE native data.

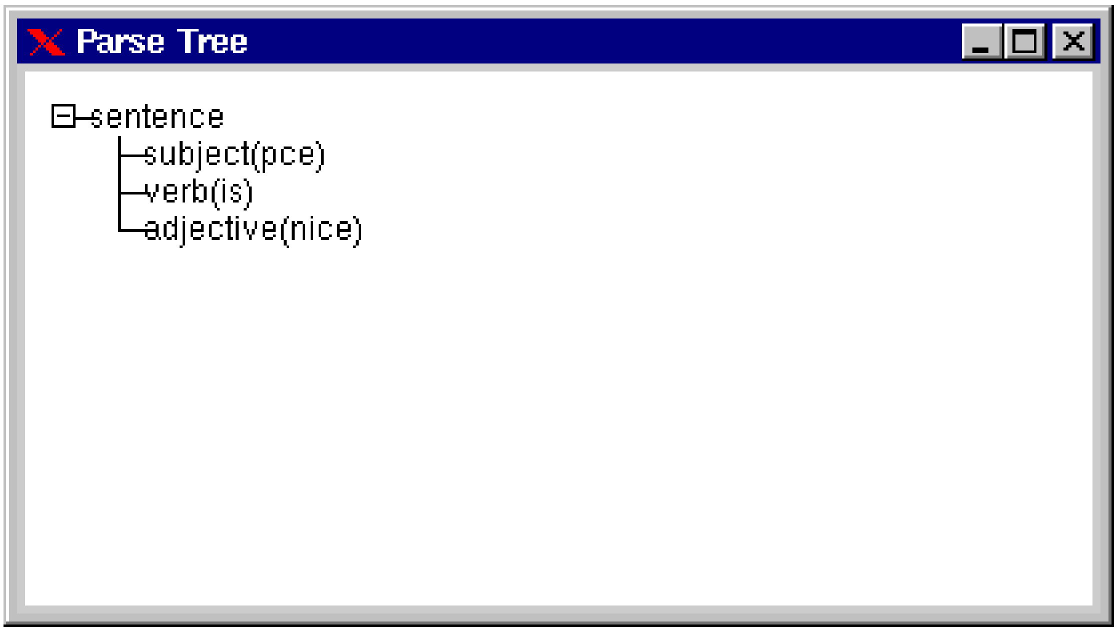
```
value(N, Value:prolog) :->
        "Set ←value and change label"::
        send(N, slot, value, Value),
        term_to_atom(Value, Label),
        send(N?image, string, Label).
```

Get the associated term, which is retrieved from the Prolog recorded database.

```
clicked(N) :->
        get(N, value, Value),
        format('User clicked "~p"~n', [Value]).
:- pce_end_class.
```

Finally, we define a Prolog tree and a the code to visualise it in a window.

```
tree(node(sentence,
          [ node(subject(pce), []),
            node(verb(is), []),
            node(adjective(nice), [])
          ])).
show_tree :-
        tree(Tree),
        new(P, picture('Parse Tree')),
        send(P, display, parse_tree(Tree), point(10,10)),
        send(P, open).
```



```
sentence
  subject(pce)
  verb(is)
  adjective(nice)
```

Figure 2.1: Resulting window for ?- showtree.

2.2: The predicates send/2 and get/3

XPCE now supports two formats for the send- and get-predicates. The principle format has been changed to:

> **send**(*Receiver, Selector(...Arg...)*)
> **get**(*Receiver, Selector(...Arg...), Answer*)

For example:

```
?- send(new(P, picture), open),
   send(P, display(box(100,100), point(10,10))).
```

Advantages of this representation are:

- No limits to the number of arguments.
- Generally easier specification of utility-predicates that wrap around the XPCE primitives as messages are represented using single Prolog term.
- Possibility to define (efficiently) other syntaxes for XPCE.[1]

As a disadvantage, XPCE produces more Prolog garbage, but using modern compilers, this is unlikely to be a problem.

The old syntax is still acceptable. When appearing as a normal subclause, the XPCE term-expansion will automatically rewrite it. Actual calls are modified at runtime.

[1]Using an infix-operator for **send/2**, we could express send using *Receiver* op *Message*. Unfortunately it is not easy to find a good operator. `->` is already reserved, `.` (dot) creates too much confusion with the final dot of a Prolog clause (a space after it would make the parser think the clause has ended). Suggestions are welcome!

2.3: Invoking behaviour on the super-class

Invoking behaviour on the super-class is implemented completely different. In the old system, send(*Object*, send_super, selector, ...) was used. In the new system, the primitive **send_class/2** is used and application programmers use **send_super/2** with the same syntax as **send/2**.

The construct **send_super/2** is translated by the XPCE term-expansion module into a proper **send_class/2** message. All variations are recognised:

send(*Receiver, send_super, Selector, ...*)
send(*Receiver, send_super(Selector, ...)*)
send_super(*Receiver, Selector, ...*)
send_super(*Receiver, Selector(...)*)

Some incompatibilities arise from this: dynamically built invocations of the old send-super construct will fail, as will calls outside the context of a class. If the message invoked from the super-class is determined dynamically, the code below should be used, i.e. the *message* should be constructed dynamically, not the *goal*.

```
...,
Msg =.. [Selector|Arguments],
send_super(Receiver, Msg),
...,
```

2.4: Variable-argument methods

XPCE methods dealing with variable-argument parameter lists use different passing rules. Instead of passing the arguments packed in a **vector** object, they are now passed as a Prolog list.

The example below redefines '**label** $\rightarrow$ **report**' to change the colour of the text according to the nature of the message:

```
:- pce_begin_class(reporter, label).

initialise(R) :->
        send_super(R, initialise, reporter, '').

report(R, Kind:name, Fmt:name, Args:any ...) :->
        colour(Kind, Colour),
        send(R, colour, Colour),
        Msg =.. [report, Kind, Fmt | Args],
        send_super(R, Msg).

colour(status, green).
colour(error,  red).
colour(_,      black).

:- pce_end_class.
```

These incompatibilities are generally easily found by scanning the source for `...`. There is no automatic rewrite feasible.

3: CLASS-VARIABLES

The XPCE-4 notion of resources, expressing user-defaults, has been replaced by the notion of class-variables, the default of which can be read from the XPCE `Defaults`-file and/or the user's `$HOME/.xpce/Defaults` file.

The notion 'resource' is now reserved for 'program resources', data required by the program that cannot (easily) be expressed in the program itself. See chapter 4.

3.1: Consequences for the application programmer

For the application-programmer, the class-expression

> **resource**(*Name, Type, String-ified-default [, Comment]*).

is replaced by

> **class_variable**(*Name, Type, Default [, Comment]*).

Please note that the default is no longer the string-ified version of the value, but the value itself.

The method '**object** ← **resource_value**' has been replaced by '**object** ← **class_variable_value**'. In addition, class variables provide normal get-behaviour, scheduled after methods and instance-variables, and the value of a class-variable can thus be retrieved using a normal get-operation.

If both a instance- and a class-variable exist with the same name, the instance-variable is initialised to **@class_default**. On first access, the value of the class-variable is used to fill the instance-variable. The code-fragment below illustrates this:

```
:- pce_begin_class(classvar_demo, object).

variable(count, int, get).
class_variable(count, int, 25).

...

show_count(CVD) :->
        get(CVD, count, Count),
        send(@display, inform, 'Count = %d', Count).
```

3.1.1: The 'compatibility/resource' library

The library(`'compatibility/resource'`) defines additional pre-processing and methods to deal with commonly used programming constructs of the old resource mechanism.

This library reports class-variable (resource) values with incompatible syntax (string-ified) and can be used to maintain source code that should run both on older versions as on XPCE-5.0. In the latter case, execute the following directive in the context of the **user** module:

```
:- (   get(@pce, version, number, Version),
       Version >= 50000
   ->  use_module(library('compatibility/resource'))
   ;   true
   ).
```

Please note that when using this library, resources as defined in chapter 4 cannot appear inside a class-definition.

3.2: Consequences for the end-user

XPCE no longer uses the X11 resource-syntax. The syntax for a class-variable default value is defined as:

⟨*class*⟩.⟨*class-variable*⟩: ⟨*value*⟩

Thus, the leading `Pce.` has been dropped and class-names are written in exact-case rather then capitalised. The value-syntax has not been changed.

The system defaults-file is located in `$PCEHOME/Defaults`, where $PCEHOME refers to the XPCE home-directory (see '@**pce** ← **home**'). The user's defaults-file is located in `$HOME/.xpce/Defaults`, which is achieved using an include-statement at the end of the system-defaults file.

4: PROGRAM RESOURCES

Resources, in the new sense of the word is data that is required by an application but cannot be expressed easily as program-code. Examples are image-files, help-files, and other files using non-Prolog syntax. Such files are declared by defining clauses for the predicate **resource/3**:

resource(*?Name, ?Class, ?PathSpec*)

Define the file specified by *PathSpec* to contain data for the resource named *Name* of resource-class *Class*.

Name refers to the logical name of the resource, which is interpreted locally for a Prolog module. Declarations in the module **user** are visible as defaults from all other modules. *Class* defines the type of object to be expected in the file. Right now, they provide an additional name-space for resources. *PathSpec* is a file specification as acceptable to **absolute_file_name/[2,3]**.

Resources can be handled both from Prolog as for XPCE. From Prolog, this is achieved using **open_resource/3**:

open_resource(*+Name, ?Class, -Stream*)

Opens the resource specified by *Name* and *Class*. If the latter is a variable, it will be unified to the class of the first resource found that has the specified *Name*. If successful, *Stream* becomes a handle to a binary input stream, providing access to the content of the resource.

The predicate **open_resource/3** first checks **resource/3**. If successful it will open the returned resource source-file. Otherwise it will look in the programs resource database. When creating a saved-state, the system saves the resource contents into the resource archive, but does not save the resource clauses.

This way, the development environment uses the files (and modifications to the **resource/3** declarations and/or files containing resource info thus immediately affect the running environment, while the runtime system quickly accesses the system resources.

From XPCE, resources are accessed using the class **resource**, which is located next to **file** below the common data-representation class **source_sink**. Many of the methods that require data accept instances of **source_sink**, making resources a suitable candidate.

Below is the preferred way to specify and use an icon.

```
resource(my_icon,       image,  image('my_icon.xpm')).

          ...,
          send(Button, label, image(resource(my_icon))),
          ...,
```

The directive **pce_image_directory/1** adds the provided directory to the search-path for images (represented in the class-variable **image.path**), as well as to the **image/1** definition of **file_search_path/2**.

Please note that MS-Windows formatted image files can currently not be loaded through **resource** objects. The Windows API only provides functions to extract these objects from a single file, or nested as Windows resources in a `.dll` or `.exe` file.

Right now, it is adviced to translate the images into `.xpm` format using the following simple command:

```
?- send(image('myicon.ico'), save, 'myicon,xpm', xpm).
```

This transformation is complete as the XPM image format covers all aspects of the Microsoft image formats.

5: ERRORS AND PROLOG EXCEPTIONS

XPCE-5.0 provides a mapping between XPCE errors and Prolog exceptions. For this reason a new '**error** ⇆ **feedback**' has been defined: `throw`. If an error is raised using '**object** → **error**' that has ←**feedback**: `throw` and the system finds a *goal* on the stack that indicates it is willing to catch errors, the error details are stored in the goal. As control is returned to Prolog, the interface maps the error details onto a Prolog exception. This exception is of the format:

error(**pce**(*ErrorId, ListOfArguments), Context*)

This error can be catched using the ISO **catch/3** construct:

```
?- catch(send(@pce, not_implemented), E, true).

E = error(pce(no_behaviour,
              [@pce/pce,  (->), not_implemented]),
          send(@pce/pce, not_implemented))
```

This error-term can be printed using **print_message/2**. The method '**error** ← **format**' may also be used to map the error into a XPCE-string:

```
error_string(ErrorId, Args, TextAsAtom) :-
        Msg =.. [format|Args],
        get(error(ErrorId), Msg, String),
        get(String, value, TextAsAtom).
```

The XPCE manual's error browser can be used to examine the defined error types.

6: MODIFIED AND DELETED FEATURES

6.1: Constraints — only automatic for graphicals

In practice, constraints are almost exclusively used to specify geometry relations between graphical objects. Until now, constraints were evaluated whenever a send-message was successfully executed. In 5.0, constraints are executed automatically if the geometry of a graphical object changes. All other case should be invoked manually using '**object** → **update_constraints**'.

If other objects are known to dependent using constraints on one or more attributes of a class, it is advised to write a wrapper that invokes →**update_constraints** whenever the relevant attributes of the object are modified.

For example, suppose a class **lamp** has been defined with a property 'is_on'. The application uses constraints to propagate the consequences of this property. The lamp should be defined according to the skeleton below:

```
:- pce_begin_class(lamp, bitmap).

variable(is_on, bool := @off, get, "Whether the lamp is on/off").

is_on(L, OnOff:bool) :->
        send(L, slot, is_on, OnOff),
        send(L, update_constraints).
```

6.2: Debugging and goal-stack examination

The class **vmi** and its instances **@vmi_send**, **@vmi_get**, **@vmi_new** and **@vmi_free** are deleted. With this, '**vmi** → **parent_goal**', a method to validate some goal is executing higher in the goal-stack has been deleted too. Applications using these constructs should take alternative measures.

The extensive debugger available in previous versions proved of little practical usage. As of version 5.0, the debugger is limited to tracing the activation of methods and the access to instance-variables, as well as breaking (like Prolog *spy*) on them. A break-point may be used to examine that status and stack-context when a method is invoked.

The interface for setting break- and trace-point is the same, using **spypce/1** and **tracepce/1**.

6.3: Edit interface

The predicate **editpce/1** has been replaced by a hook into the generic **edit/1** predicate. This hook provides the following functionality:

ClassOrObject `->` *Method*
Specifies a send-method. Either *Class* or *Method* can be a variable.

ClassOrObject `<-` *Method*
Specifies a get-method. Either *Class* or *Method* can be a variable.

Class
In addition to the predefined objects, XPCE-classes of this name are returned.

Example:

```
?- edit(_->update).
Please select item to edit:

  1 q_agenda->update            boot('inferui.pl'):455
  2 toc_rule_node->update       boot('inferui.pl'):543
  3 btext_item->update          boot('lib/bitem.pl'):41

Your choice?
```

7: LAYOUT MANAGERS

A *Layout Manager* is an object that is associated with a graphical **device** and which deals with managing the layout of the graphicals displayed on the device. Layout-managers can either manipulate the graphicals on the device directly, or it can attach a **layout_interface** object to each the graphicals managed. The **layout_interface** object contains data required by the layout manager about the layout properties of the graphical object.

In addition to managing the layout, layout_managers can provide hooks to paint the background of a device, as well as to direct events.

In the future, layout-managers will form a hierarchy with a role similar to the recogniser hierarchy. Currently the interface is only used by class **table**, using **table_cell** as a refinement of **layout_interface**.

7.1: Graphical tables

The most important graphical modification is the introduction of class **table**, defining graphical tables modelled after HTML-3 tables. The relevant classes are listed in the partial class-hierarchy below:

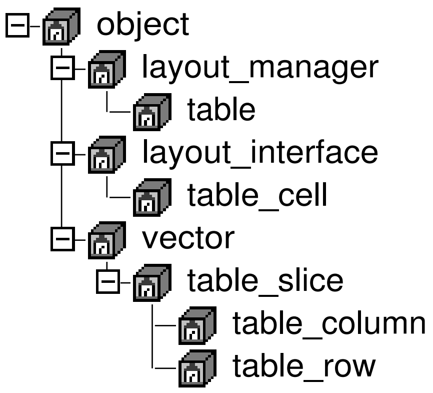


Figure 7.1: Location in the hierarchy of the table-classes

7.2: Example — Show contents of a directory in a table

```
:- module(show_directory,
          [ show_directory/1
          ]).
:- use_module(library(pce)).
```

Define images we use as resources. See chapter 4 for details.

```
resource(dir,   image,  image('16x16/closedir.xpm')).
resource(doc,   image,  image('16x16/doc.xpm')).
resource(pce,   image,  image('16x16/pce.xpm')).
```

The toplevel of this module simply creates an instance of the class dir_listing and opens it.

```
show_directory(D) :-
        send(new(dir_listing(D)), open).

:- pce_begin_class(dir_listing, picture,
                   "Show contents of a directory").

variable(directory,     directory,      get, "Current directory").
variable(sort_column,   name,           get, "Name of column to sort on").
```

Change the ordinary picture window into a table. First, the table is associated using 'device →**layout_manager**'. Then the parameters of the table are filled. Note the similarity between the HTML-3 definition of terms and those used here.

```
initialise(DL, Dir:directory) :->
        send_super(DL, initialise),
        send(DL, layout_manager, new(T, table)),
        send(T, rules, groups),
        send(T, frame, box),
        send(T, border, 1),
        (   column(I, Name, Align, Label),
            get(T, column, I, @on, Col),        % @on: create
            send(Col, halign, Align),
            send(Col, name, Name),
            send(T, append, Label),
            fail
        ;   true
        ),
        send(T, next_row, @on),                 % @on: end group
        send(DL, sort_column, name),
        send(DL, directory, Dir).
```

Definition of the columns. This is easier to read, write and maintain than long sequences of send-operation in the →**initialise** method above. The class sortable_column_label is defined further down this file.

```
%      Index    Name,   Align,  Label
column(1,      image,   center, new(graphical)).
column(2,       size,   right,  sortable_column_label(size)).
column(3,   modified,   right,  sortable_column_label(modified)).
column(4,       name,   left,   sortable_column_label(name)).

clear(DL) :->
        "Remove all entries, except for the title-row"::
        get(DL, layout_manager, Table),
        send(Table, delete_rows, 2).

sort_column(DL, Col:name) :->
        "Switch sort column and underline proper label"::
        send(DL, slot, sort_column, Col),
        get(DL, layout_manager, Table),
        new(TitleCell, ?(@arg1, cell, 1)?image),
        send(Table?columns, for_all,
             if(message(TitleCell, instance_of, sortable_column_label),
                if(TitleCell?name == Col,
                   message(TitleCell, underline, @on),
                   message(TitleCell, underline, @off)))).

sort(DL, On:[name]) :->
        (   On == @default
        ->  true
        ;   send(DL, sort_column, On)
        ),
        get(DL, layout_manager, Table),
        send(Table, sort_rows, ?(DL, compare_rows, @arg1, @arg2), 2).

compare_rows(DL, R1:table_row, R2:table_row, Result) :<-
        "Compare two rows on ←sort_column"::
        get(DL, sort_column, ColName),
        get(R1, cell, ColName, C1),
        get(R2, cell, ColName, C2),
        get(C1, image, Gr1),
        get(C2, image, Gr2),
        get(Gr1, compare, Gr2, Result).

:- pce_group(event).

:- pce_global(@direcory_listing_recogniser,
              new(click_gesture(left, '', single,
                                message(@receiver, clicked,
                                        ?(@receiver, current, @event))))).

current(DL, Event:event, Current:'file|directory') :<-
        "Return pointed file/directory"::
        get(DL, layout_manager, Table),
        get(Table, cell_from_position, Event, Cell),
        get(Cell, row, RowN),
        get(Table, row, RowN, Row),
        get(Row, attribute, client, Current).

event(DL, Ev:event) :->
        (   send_super(DL, event(Ev))
        ;   send(@direcory_listing_recogniser, event, Ev)
        ).

clicked(DL, Clicked:'file|directory') :->
        (   send(Clicked, instance_of, directory)
        ->  send(DL, directory, Clicked)
        ;   true
        ).

:- pce_group(build).
```

Build the contents of the table.

```
directory(DL, Dir:directory) :->
        send(DL, slot, directory, Dir),
        send(DL, clear),
        new(Files, chain),
        new(Dirs, chain),
        send(Dir, scan, Files, Dirs),
        send(Dirs, for_all,
             message(DL, append_directory,
                     ?(Dir, directory, @arg1))),
        send(Files, for_all,
             message(DL, append_file,
                     ?(Dir, file, @arg1))),
        send(DL, sort).

append_directory(DL, Dir:directory) :->
        "Append a directory-row"::
        get(DL, layout_manager, Table),
        get(Table, row, Table?current?y, @on, Row),
        send(Row, attribute, client, Dir),
        send(Table, append, bitmap(resource(dir))),
        send(Table, append, dir_value_text(0)),
        send(Table, append, dir_value_text(Dir?time)),
        send(Table, append, dir_value_text(Dir?name)),
        send(Table, next_row).

append_file(DL, File:file) :->
        "Append a directory-row"::
        get(DL, layout_manager, Table),
        get(File, base_name, Name),
        file_image(Name, Image),
        get(Table, row, Table?current?y, @on, Row),
        send(Row, attribute, client, File),
        send(Table, append, bitmap(Image)),
        send(Table, append, dir_value_text(File?size)),
        send(Table, append, dir_value_text(File?time)),
        send(Table, append, dir_value_text(Name)),
        send(Table, next_row).

file_type('*.pl',       pce).
file_type(*,            doc).

file_image(File, image(resource(ResName))) :-
        new(Re, regex),
        file_type(Pattern, ResName),
        send(Re, file_pattern, Pattern),
        send(Re, match, File), !.

:- pce_end_class.
```

Class sortable_column_label provides the label displayed in the first (title) row. If the lable is clicked, it will send a message to the window to sort the represented table on the named column.

```
:- pce_begin_class(sortable_column_label, text).

initialise(L, Name:name) :->
        send_super(L, initialise(Name?label_name, font := bold)),
        send(L, name, Name).

:- pce_global(@sortable_column_label_recogniser,
              new(click_gesture(left, '', single,
                                message(@receiver, clicked)))).

event(L, Ev:event) :->
        (   send(L, send_super, event, Ev)
        ;   send(@sortable_column_label_recogniser, event, Ev)
        ).

clicked(L) :->
        "Clicked. Send ->sort to the window"::
        get(L, device, Window),
        send(Window, sort, L?name).

:- pce_end_class.
```

Class dir_value_text is a simple subclass of class text, providing a generic ←**compare** method on the represented value, thus keeping the code for sorting the table on a named column generic and simple.

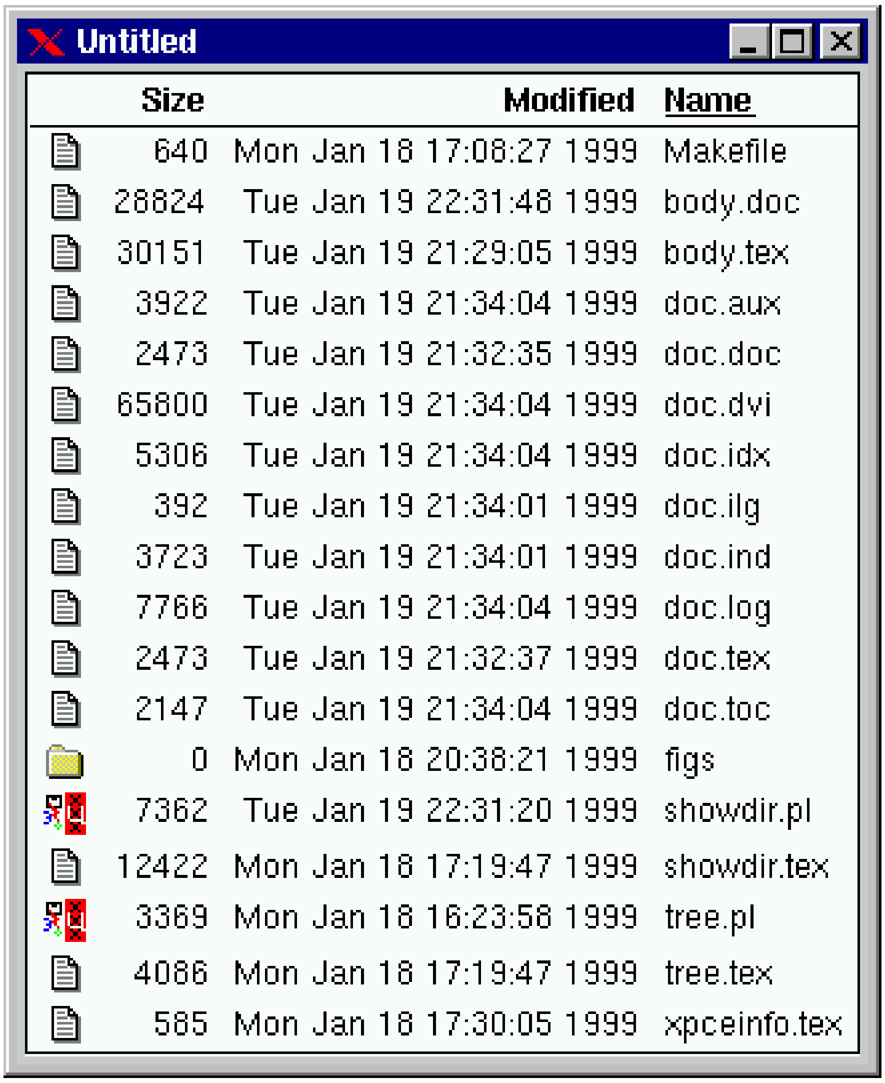
```
:- pce_begin_class(dir_value_text, text,
                   "Represent a value, providing <-compare").

variable(value, any, get, "Represented value").

initialise(C, Value:any) :->
        send(C, slot, value, Value),
        send_super(C, initialise(Value?print_name)).

compare(N1, N2:dir_value_text, Result) :<-
        get(N1?value, compare, N2?value, Result).
```

```
:- pce_end_class.
```



Untitled

Size	Modified	Name
640	Mon Jan 18 17:08:27 1999	Makefile
28824	Tue Jan 19 22:31:48 1999	body.doc
30151	Tue Jan 19 21:29:05 1999	body.tex
3922	Tue Jan 19 21:34:04 1999	doc.aux
2473	Tue Jan 19 21:32:35 1999	doc.doc
65800	Tue Jan 19 21:34:04 1999	doc.dvi
5306	Tue Jan 19 21:34:04 1999	doc.idx
392	Tue Jan 19 21:34:01 1999	doc.ilg
3723	Tue Jan 19 21:34:01 1999	doc.ind
7766	Tue Jan 19 21:34:04 1999	doc.log
2473	Tue Jan 19 21:32:37 1999	doc.tex
2147	Tue Jan 19 21:34:04 1999	doc.toc
0	Mon Jan 18 20:38:21 1999	figs
7362	Tue Jan 19 22:31:20 1999	showdir.pl
12422	Mon Jan 18 17:19:47 1999	showdir.tex
3369	Mon Jan 18 16:23:58 1999	tree.pl
4086	Mon Jan 18 17:19:47 1999	tree.tex
585	Mon Jan 18 17:30:05 1999	xpceinfo.tex

Figure 7.2: Resulting window for ?- show_directory('.').

8: REFERENCE PAGES

Below is the reference documentation for the modified XPCE/Prolog interface. This material will shortly be merged into "Programming in XPCE/Prolog".

8.1: Interface predicates

In all definitions below, *Message* is either an atom or a compound term. The functor-name (or the atom) of the message denotes the *selector* for the behaviour addressed. The arguments of a compound *Message* are used to build the argument vector for the goal.

The following steps are executed when sending a message using any of these predicates:

1. Convert the *Object* argument into a reference to a XPCE object.
2. Extract the *selector* from the *Message* and resolve the *implementation* of the behaviour denoted by the combination of the object and *selector*. This process can yield a new receiving object, either due to function-evaluation if the receiving object is a function and behaviour is not defined on the function itself, or due to *delegation*.
3. Determine a description (a vector of **type** objects) of the arguments required by the implementation.
4. Allocate and fill the actual argument vector. This process deals with named arguments (*Name, Value* := *Name, Value*, type-conversion as well as filling non-specified default arguments.

   The interface here already discriminates between XPCE implemented behaviour from Prolog implemented. In the first case the argument vector is an array of native XPCE data. If an argument accepts Prolog native data, the argument is translated into an instance of class **prolog_term**, providing a handle to the Prolog term. In the second case, it creates a vector of Prolog term references for creating the call to pce_principal:**send_implementation/3** or pce_principal:**get_implementation/4**.
5. Execute the implementation by calling a XPCE interface function if the implementation is in XPCE, or by calling the Prolog implementation directly.
6. If it concerns a get-operation, convert the return value to the proper type.
7. Discard all garbage object created in this process, except for the return-value of a get-operation.

send(*+Object, :Message*)
: Send *Message* to the indicated *Object*. In general, a send-operation is intented to modify an object or test the object for some property. This predicate either succeeds or fails if it concerns a test which fails. If an error is encountered, the error is reported to the GUI or the terminal or mapped onto a Prolog exception. See section 5 for details.

send_class(*+Object, +Class, :Message*)
: As **send/2**, but the implementation is resolved only considering methods defined at the level of the specified *Class* or higher in the inheritence hierarchy.

  This can be used in very special cases if one wants to force the usage of a particular implementation. It is dangerous as assumptions made in classes between the actual object-class and *Class* may be violated.

Normally, **send_super/2** should be used, which is mapped to a proper call of **send_class/3** by the class-compiler.

send_super(+*Object, :Message*)
This actually is not a predicate, but syntactic sugar translated by the XPCE class-compiler into an appropriate **send_class/3** call. This construct can only appear in the context of a method-definition. It is good practice that any method being refined in a sub-class invokes the implementation of its super-class at some stage, as this guarantees no assuptions made in the super-class are violated. For example, if a subclass of class **device** is defined to realise a specific graphical object, it **must** invoke **send_super**(*Dev, initialise*) in its refined →**initialise** method before it displays any **graphical** object on the device:

```
:- pce_begin_class(my_graphical, device).

initialise(MG) :->
        send_super(MG, initialise),
        send(MG, display(box(100, 50))),
        ...
```

get(+*Object, :Message, -Answer*)
Similar to **send/2**, but get-behaviour returns a value. The following cases require attention.

- *A fresh 'attribute' object is returned*
  For example, '**box** ← **size**' returns and instance of class **size**, representing the current size of the box object. This size object has no relation to its creator and may be modified freely. It need not be discarded as the incremental garbage collector will deal with it if the object is not protected using →**lock_object** or by associating it to another object.
- *An 'attribute' is returned*
  Unlike '**box** ← **size**', '**box** ← **area**' returns an instance of class **area** describing the area of the box in the coordinate system of its ←**device**, but the area object returned is a filler for the '**graphical** ← **area**' instance-variable of the box object. Modifying this area will leave the box and its ←**device** in an inconsistent state.
- *A normal fresh object is returned*
  Some get-behaviour creates a new, completely independent instance ready and intended for further manipulation. This case is from the programmers point of view no different from the return of a 'fresh' attribute object.
- *A fully functional part is returned*
  If a device is, using '**device** ← **member**', asked for the object-reference of a displayed graphical object, the returned object may be modified freely as graphicals knon on what device they are displayed and will inform this device of any relevant information, including the death of the graphical. Whether or not such a relation exists can only be found in the documentation. In general, if built-in objects are aware of each others existence, they will inform each other.

get_class(+*Object, +Class, :Message, -Answer*)
The predicate **get_class/4** is what **send_class/3** is for get-behaviour.

get_super(+*Object, :Message, -Answer*)
The predicate **get_super/3** is what **send_super/2** is for get-behaviour.

8.1.1: Alternative interface predicates

Both for compatibility reasons as because it will remain the preferred syntax for some users, the predicates **send/[3-12]** and **get/[4-13]** are defined:

send(+*Object, :Selector, ... +Arg ...*)
Equivalent to **send**(*Object, Selector(...Arg...)*). The XPCE macro-expansion translates these calls to **send/2**. The predicates are defined as well to deal with invocation through meta-predicates that is not captured by the macro-expansion. The following two calls are fully identical:

```
send(Window, display(box(100, 50), point(10, 10))),
send(Window, display, box(100, 50, point(120, 10))).
```

Note however that the existence of **send/2** makes the following valid code:

```
send_list(Window,
          [ border(1),
            background(green),
            display(box(100,100), point(10,10))
          ])
```

get(+*Object, :Selector, ... +Arg ..., -Answer*)
These predicates are handled as **send/[3-12]**.

9: STATUS, DISCUSSION AND PLANS

9.1: Prolog interface

XPCE-5.0.0 is primarily an evaluation release for the new XPCE/Prolog interface. Except for the issues noted in these release notes (especially chapter A), old code is supposed to be fully compatible with the new release. With this beta release we want to access this claim, as well as your opinion on the direction chosen with the interface.

One issue is whether to allow Prolog data (**prolog_term**) in more places by including them into type **any**. This would make the association of Prolog data with XPCE objects without user-defined classes possible, but it would harm compatibility and possibly make the intention of source-code less obvious. Consider:

```
?- new(Chain, chain(point(1, 2))).
```

If **any** accepts Prolog data, this will be a chain object holding a Prolog term. Translation into a point-object will only take place at the moment a type is requested that does not accept **prolog_term**. This implies translation may take place multiple times, resulting in multiple point instances being created from the same term.

Alternatively, it would be possible to define a class, say **term**, that is a normal subclass of class **object** and contains a single slot holding the **prolog_term** object, so a Prolog term can be stored in a chain using

```
?- new(Chain, chain(term(point(1, 2)))).
```

Conversion back to Prolog could be achieved automatically (as with the classes **real** and **prolog_term**, or by hand using a method '**term** ← **value**'. Comments?

9.2: Tables

Graphical tables have been tested quite extensively on a couple of applications now. The basic functionality is stable, but both new functionality (scrolling column/row subregions, 3D look) and utility functions will be added. Suggestions are welcome.

A: MIGRATING OLD SOURCE CODE

This section provides a brief overview of things to check when migrating existing code to XPCE-5.0.

- *Resources*
  If an application uses resources in user-defined classes, there are two options. One is to load library(`'compatibility/resource'`), the other is to do a global replace of the term `resource` into `class_variable`. Values are represented using the normal object-notation. The compatibility package will warn for possible problems. The advised route therefore it to use the compatibility package first, fix the value-notation and finally replace `resource` with `class_variable`. See chapter 4.
- *Variable-argument methods*
  Argument are passed as a Prolog list, rather then a **code_vector** instance. In general forwarding is implemented by using Prolog univ (=../2) to create a message and then calling **send/2** or **send_super/2**. Commonly, such constructs are found using `grep` for the pattern `...`., possibly `"... *) *:"` See chapter 2.4.
- *Send super*
  Send/Get super cannot be used outside the context of a class. The compiler will warn if it encounters a send-super call for which it cannot resolve the context class. This problem has to be fixed manually. See chapter 2.3.

Finally, if the generation of a runtime is desirable, it is advised to use the new-style resources to declare and access your images and other program resources.

INDEX