

Bibliography

- [Anjewierden, 1992] A. Anjewierden. *PCE/Lisp: PCE Common Lisp Interface*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: anjo@swi.psy.uva.nl.
- [Wielemaker & Anjewierden, 1992a] J. Wielemaker and A. Anjewierden. *PCE-4 Functional Overview*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.
- [Wielemaker & Anjewierden, 1992b] J. Wielemaker and A. Anjewierden. *Programming in PCE/Prolog*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.
- [Wielemaker, 1992a] J. Wielemaker. *PCE-4 User Defined Classes Manual*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.
- [Wielemaker, 1992b] J. Wielemaker. *PceDraw: An example of using PCE-4*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.

```

deliver(F, M:male, Name:name, Date:name, Sex:{male,female}, Child:person) :-
    "Deliver a child"::
    (   Sex == male
    -> new(Child, male(Name, Date)),
        new(_, hyper(F, Child, son, mother)),
        new(_, hyper(M, Child, son, father))
    ;   new(Child, female(Name, Date)),
        new(_, hyper(F, Child, daughter, mother)),
        new(_, hyper(M, Child, daughter, father))
    ).

:- pce_end_class.

:- pce_begin_class(male, person, "Male person").

mary(M, F:female) :->
    "Marry with me"::
    send(F, mary, M).

wife(M, Female:female) :-<
    "To whom am I married?"::
    get(M, hypered, woman, Female).

:- pce_end_class.

```

7.3 Exercises

Exercise 14

Load the file family.pl containing the example above and enter a simple database by hand. Inspect the data-representation using the inspector and online manual tools.

Exercise 15

Design and implement a simple graphical editor for entering a database. What visualisation will you choose? What UI technique will you use for marriage and born children?

Exercise 16

The current implementation does not define an object that reflects the entire database. Define and maintain a hash-table that maps names onto person entries. You can redefine destruction of an object by redefining the \rightarrow *unlink* method.

```

initialise(P, Name:name, BornAt:name) :->
    send(P, send_super, initialise),
    send(P, name, Name),
    send(P, date_of_birth, BornAt).

father(P, M:male) :-<-
    "Get my father"::
    get(P, hypered, father, M).

mother(P, M:female) :-<-
    "Get my mother"::
    get(P, hypered, mother, M).

sons(P, Sons:chain) :-<-
    "Get my sons"::
    get(P, all_hypers, Hypers),
    new(Sons, chain),
    send(Hypers, for_all,
        if(@arg1?forward_name == son,
            message(Sons, append, @arg1?to))).

daughters(P, Daughters:chain) :-<-
    "Get my daughters"::
    get(P, all_hypers, Hypers),
    new(Daughters, chain),
    send(Hypers, for_all,
        if(@arg1?forward_name == daughter,
            message(Daughters, append, @arg1?to))).

:- pce_end_class.

:- pce_begin_class(female, person, "Female person").

mary(F, Man:male) :->
    "Marry with me"::
    ( get(F, husband, Man)
    -> send(F, report, error, '%N is already married to %N', F, Man),
      fail
    ; new(_, hyper(F, Man, man, woman))
    ).

husband(F, Man:male) :-<-
    "To whom am I married"::
    get(F, hypered, man, Man).

```

- *chain*
A chain is a single-linked list. Class chain defines various set-oriented operations and iteration primitives.
- *vector*
A vector is a dynamically expanding one-dimensional array. Multi-dimensional arrays may be represented as vectors holding vectors or using a large one-dimensional array and redefine the access methods. See also section 3.3.3.1.
- *hash_table*
A hash-table maps an arbitrary key on an arbitrary value. Class hash_table defines various methods for finding and iteration over the associations stored in the table. Class chain_table may be considered to associate a single key with multiple values.

For the representation of frames and relations, XPCE offers the following building-blocks:

- *sheet*
A sheet is a dynamic attribute-value set. Cf. a property list in Lisp.
- *Refining class object*
A common way to define storage primitives is by modeling them as subclasses of class object. This way of modeling data allows you to exploit the typing primitives of XPCE. Modeling as classes rather than using dynamic sheets also minimises storage overhead.
- *hyper*
A hyper is a relation between two objects. The classes hyper and objects provide methods to define the semantics of hypers and to manipulate and exploit them in various ways.

A hyper is a simple and safe way to represent a relation between two objects than cannot rely on their mutual existence.

7.2 A Simple Database

In this section we will define a simple family database.² This example used hyper-objects to express marriage and child relations. The advantage of hyper-objects is that they will automatically be destroyed if one of the related objects is destroyed and they may be created and queried from both sides.

```
:- pce_begin_class(person, object, "Person super-class").

variable(name,          name,    both,    "Name of the person").
variable(date_of_birth, name,    both,    "Textual description of date").
```

²Unfortunately we cannot use class data for representing dates as the current implementation only ranges from the Unix epoch (Jan 1 1970 to somewhere in the 22-nth century).

Chapter 7

Representation and Storing Application Data

There are various alternatives for representing application data in XPCE/Prolog. The most obvious choice is to use Prolog. There are some cases where XPCE is an alternative worth considering:¹

- *Store data that is difficult to represent in Prolog*
The most typical example are hyper-text and graphics. XPCE has natural data-types for representing this as well as a save and load facilities to communicate with files.
- *Manipulations that are hard in Prolog*
XPCE has a completely different architecture than Prolog: its basic control-structure is message passing and it's data-elements are global and use destructive assignment. These properties can make it a good alternative for storing data in the recorded or clause database of Prolog.
- *You want to write a pure OO system*
Not only can you model your interface, but also the *application* as a set of XPCE classes. This provides you with a purely object oriented architecture where XPCE is responsible for storage and message passing and Prolog is responsible for implementing the methods.

7.1 Data Representation Building Blocks

In this section we will discuss the building-blocks for data-representation using XPCE. We start with the data *organising* classes:

¹Using XPCE for storage of application-data has been used in three Esprit-funded projects for the development of knowledge acquisition tools: 'Shelley' (storage only), The 'Common Kads WorkBench' (using XPCE for OO control structure) and 'KEW' (Common Lips, Using XPCE for storage only). In the 'Games' project XPCE user-defined classes are only used for specialising the UI library. CLOS is used for storing application data.

Representing 'knowledge' in XPCE to be used for reasoning in Prolog or Lisp is difficult due to the lack of 'natural' access to the knowledge base. Representing text and drawings in XPCE works well.

```
:- pce_end_class.
```

6.4 Reading the Diagram

XPCE has been designed to be able to create drawings that can be interpreted. This design is exemplified by the use of connections. If the drawing had been built from just lines, boxes and text it would have been difficult to convert the drawing into a Prolog representation of a graph. Connections explicitly relate *graphical object* instead of positions.

Below is a Prolog fragment that generates a Prolog fact-list from the graph.

```
assert_graph(E, Predicate) :-
    functor(Term, Predicate, 2),
    retractall(Term),
    send(E?graphicals, for_all,
        if(message(@arg1, instance_of, connection),
            message(@prolog, assert_link,
                    Predicate,
                    @arg1?from?string?value,
                    @arg2?from?string?value))).

assert_link(Predicate, From, To) :-
    Term =.. [Predicate, From, To],
    assert(Term).
```

6.5 Exercises

Exercise 10

Extend the graph-editor's frame with a dialog window that allows for saving the graph to the Prolog database.

Exercise 11

The example in section ?? uses a generic connect gesture and a generic connection. Make a refinement of class connect_gesture that creates instances of a class arc_connection. Define class arc_connection as a subclass of class connection that has a popup attached which allows the user to delete the connection.

Exercise 12

Write a predicate to create a graph from a list of Prolog facts defining the graph. Add a possibility to load a graph to the dialog window.

Exercise 13

Experiment with the 'graphical \rightarrow layout' method to create some layout for the graph-elements created in the above exercise

```

                end_group := @on),
            menu_item(delete,
                message(TB, free))
        ]).

event(TB, Ev:event) :->
    ( send(TB, send_super, event, Ev)
      -> true
      ; send(@move_resize_recogniser, event, Ev)
      ).

rename(TB) :->
    "Prompt for new name":
    prompt('Rename text box',
        [ name:name = Name/TB?string
          ]),
    send(TB, string, Name).

:- pce_end_class.

```

6.3 Graphs: Using Connections

A connection is a line connecting two graphical objects. A connection has typed end-points that can attach to a *handle* of the same type. If multiple such handles exist, the system will attach to the ‘best’ one using some simple heuristics.

Below is the code fragment that allows you to link up graphicals by dragging from the one to the other with the left-button held down.

```

:- pce_extend_class(text_box).

handle(0, h/2, link, west).
handle(w, h/2, link, east).
handle(w/2, 0, link, north).
handle(w/2, h, link, south).

:- pce_global(@link_recogniser, new(connect_gesture)).

event(TB, Ev:event) :->
    ( send(TB, send_super, event, Ev)
      ; send(@move_resize_recogniser, event, Ev)
      ; send(@link_recogniser, event, Ev)
      ).

```

```

send_list(P, append,
          [ menu_item(add_new_box,
                      message(@prolog, add_new_box,
                              E, E?focus_event?position)),
            menu_item(clear,
                      and(message(@display, confirm,
                                  'Clear entire drawing?'),
                          message(E, clear)))
          ],
send(E, open).

```

```

add_new_box(E, Pos) :-
    prompt('Name of box',
          [ name:name = Name
            ]),
    send(E, display, text_box(Name), Pos).

```

6.2.2 Using gestures

A gesture is an object that is designed to parse mouse-button event sequences into meaningful actions. XPCE predefines gestures for clicking, moving, resizing, linking and drag-drop of graphical objects.

All gesture classes are designed to be subclassed for more application specific handling of button-events.

There are two ways to connect a gesture to a graphical object. The first is ‘graphical \rightarrow recogniser’ which makes the gesture work for a single graphical object. The alternative is to define an \rightarrow event method for the graphical. Below we will use this to extend our text-box class with the possibility to move and resize the boxes.¹

```

:- pce_extend_class(text_box).

:- pce_global(@text_box_recogniser,
             make_text_box_recogniser).

make_text_box_recogniser(R) :-
    new(R, handler_group(new(resize_gesture),
                        new(move_gesture),
                        popup_gesture(new(P, popup(options))))),
    TB = @arg1,
    send_list(P, append,
          [ menu_item(rename,
                      message(TB, rename),

```

¹In this example we define the class text_box in small parts that can be loaded on top of each other. We use :- pce_extend_class(+Class). to continue the class-definition.

```

geometry(TB, X:[int], Y:[int], W:[int], H:[int]) :->
    "Handle geometry-changes"::
    get(TB, member, box, Box),
    send(Box, set, 0, 0, W, H),
    send(TB, recenter),
    send(TB, send_super, geometry, X, Y).

fill_pattern(TB, Pattern:image*) :->
    "Specify fill-pattern of the box"::
    get(TB, member, box, Box),
    send(Box, fill_pattern, Pattern).
fill_pattern(TB, Pattern:image*) :<-
    "Read fill-pattern of the box"::
    get(TB, member, box, Box),
    get(Box, fill_pattern, Pattern).

string(TB, String:char_array) :->
    "Specify string of the text"::
    get(TB, member, text, Text),
    send(Text, string, String).
string(TB, String:char_array) :<-
    "Read string of the text"::
    get(TB, member, text, Text),
    get(Text, string, String).

:- pce_end_class.

```

6.2 Making Graphicals Sensitive

6.2.1 Adding popup menus to graphicals

In this section we will use the `text_box` defined in the previous section to define a graphical editor that allows you to create a graph of with text-boxes as nodes. In the first step we will define window that allows us to add new text-boxes to it using a background popup. We will prompt for the text to be put in the box.

```

:- use_module(library(prompt)).

make_graph_editor(E) :-
    new(E, picture('Graph Editor')),
    send(E, popup, new(P, popup(options))),

```

Chapter 6

Graphics

In this chapter we will build a little direct-manipulation graphical editor. The application consists of a graphical editor that allows the use to define entities and their interrelations involved.

6.1 Graphical Devices

A graphical device (class device) defines a compound graphical object with it's own local coordinate system. Graphical devices may be nested.

Graphical devices are usually subclassed to define application-specific visual representations. As a first step towards our graphical application, we will define a box with centered text and a possibly grey background.

```
:- pce_begin_class(text_box, device, "Box with centered text").

resource(size,          size,    '100x50',          "Default size").
resource(label_font,    font,    '@helvetica_bold_12' "Font for text").

initialise(TB, Label:[name]) :->
    "Create a box with centered editable text"::
    send(TB, send_super, initialise),
    get(TB, resource_value, size, size(W, H)),
    get(TB, resource_value, label_font, Font),
    send(TB, display, box(W, H)),
    send(TB, display, new(T, text(Label, center, Font))),
    send(T, transparent, @off),
    send(TB, recenter).

recenter(TB) :->
    "Center text in box"::
    get(TB, member, box, Box),
    get(TB, member, text, Text),
    send(Text, center, Box?center).
```

The first argument of $\rightarrow report$ is the type of report. The reporting mechanism uses this information to decide what to do with the report. The types are: *status*, *warning*, *error*, *inform*, *progress* and *done*.

5.4.2 Handling reports

A default report handling mechanism is defined that will try to report in a *label* object called *reporter* in a dialog window in the frame from where the command was issued. If no such label can be found important (error, inform) reports are reported using a confirmer on the display. Less serious (warning) are reported using the $\rightarrow alert$ mechanism ($\rightarrow bell$, $\rightarrow flash$) and others are ignored.

Most applications therefore can just handle all reports by ensuring the frame has a dialog window with a label called *reporter* in it. In specific cases, redefining $\rightarrow report$ or $\leftarrow report_to$ can improve error reporting.

5.5 Exercises

Exercise 9

Define a dialog based application that allows you to inspect the properties of Prolog predicates. Use `predicate_property/2` and `source_file/2`. The dialog should have appropriate fields for the file the predicate is defined in, whether it is dynamic or not, multiple, built-in, etc.

Use a browser to display all available predicates. Double-clicking on an predicate in the browser should show the predicate in the dialog.

5.3 Editing Attributes of Existing Entities

All dialog items that may be used to edit attributes (i.e. represent some value) define a \Leftarrow *default* value and the methods \rightarrow *restore* and \rightarrow *apply*. \Leftarrow *default* specifies an XPCE *function* object or plain value. On \rightarrow *restore*, the \Leftarrow *selection* is restored to the \Leftarrow *default* or the result of evaluating the \Leftarrow *default* function. \rightarrow *apply* will execute \Leftarrow *message* with the current \Leftarrow *selection* if the \Leftarrow *selection* has been changed by the user.

Class *dialog* defines the methods \rightarrow *restore* and \rightarrow *apply*, which will invoke these methods on all items in the dialog that understand them. This schema is intended to define *attribute* editors comfortably.

Below we define a dialog-window to edit some appearance values of a box object.

```
edit_box(Box) :-
    new(D, dialog(string('Edit box %N', Box))),
    send(D, append,
        text_item(name, Box?name, message(Box, name, @arg1))),
    send(D, append,
        new(S, slider(pen, 0, 10, Box?pen, message(Box, pen, @arg1)))),
    send(S, width, 50),
    send(D, append, button(apply)),
    send(D, append, button(restore)),
    send(D, append, button(cancel, message(D, destroy))),
    send(D, default_button, apply),
    send(D, open).
```

5.4 Status, Progress and Error Reporting

XPCE defines a standard protocol for reporting progress, errors, status, etc. This protocol is implemented by two methods: the send-method \rightarrow *report* and the get-method \Leftarrow *report_to*. The aim is to provide the application programmer with a mechanism that allows for reporting things to the user without having to know the context of the objects involved.

5.4.1 Generating reports

For example, you define a predicate *my_move* that moves a graphical to some location, but not all locations are valid. You can simply define this using:

```
my_move(Gr, Point) :-
    ( valid_position(Gr, Point)
    -> send(Gr, move, Point)
    ; send(Gr, report, warning,
        'Cannot move %N to %N', Gr, Point)
    ).
```

5.2 Entering Values

Operations often require multiple values and dialog windows are a common way to specify the values of an operation. Actually executing the operation is normally associated with a button. Below is an example of a dialog for this type of operations.

```
create_window :-
    new(D, dialog('Create a new window')),
    send(D, append, new(Label, text_item(label, 'Untitled'))),
    send(D, append, new(Class, menu(class, choice))),
    send_list(Class, append,
        [ class(window),
          class(picture),
          class(dialog),
          class(browser),
          class(view)
        ]),
    send(D, append,
        new(Width, slider(width, 100, 1000, 400))),
    send(D, append,
        new(Height, slider(height, 100, 1000, 200))),
    send(D, append,
        button(create,
            message(@prolog, create_window,
                    Class?selection,
                    Label?selection,
                    Width?selection,
                    Height?selection))),
    send(D, append,
        button(cancel, message(D, destroy))),
    send(D, open).
```

```
create_window(Class, Label, Width, Height) :-
    get(Class, instance, Label, Window),
    send(Window?frame, set, @default, @default, Width, Height),
    send(Window, open).
```

Note that a menu accepts the method \rightarrow *append*: menu_item. Class menu_item defines a type-conversion converting any value to a new menu_item using the value as 'menu_item \Leftarrow value'. The visible labels of the menu_items are generated automatically (see 'menu_item \Leftarrow default_label').

The action associated with the button is a message invoking the Prolog predicate create_window/4. Four obtainers collect the values to be passed.

Chapter 5

Dialogue Windows in Depth

5.1 Modal Dialogue Windows (Prompters)

A common use is to query or inform the user during an ongoing task. The task is supposed to be blocked until the user presses some button. For example, the user selected ‘Quit’ and you want to give the user the opportunity to save before doing so or cancel the quit all the way.

Model dialog windows are the answer to the problem. In XPCE, Windows by themselves are not *modal*, but any window may be operated in a *modal* fashion using the method ‘frame \leftarrow confirm’.

This method behaves as ‘frame \rightarrow open’ and next starts an event-dispatching subloop until ‘frame \rightarrow return’ is activated. So, our quit operation will look like this:

```
quit :-
    new(D, dialog('Quit my lovely application?')),
    ( application_is_modified
      -> send(D, append,
            button(save_and_quit,
                  message(D, return, save_and_quit)))
      ; true
    ),
    send(D, append, button(quit, message(D, return, quit))),
    send(D, append, button(cancel, message(D, return, cancel))),

    get(D, confirm, Action),
    send(D, destroy),
    ( Action == save_and_quit
      -> save_application
      ; Action == quit
      -> true
    ),
    send(@pce, die).
```

```
PCE: 2 exit: V @892203/point <->x: 20
```

```
P = @892203/point
```

Both `tracepce/1` and `breakpce/1` accept a specification of the form `<Class> ->`, `<-` or `-<Selector>`.

```
?- tracepce((box->device)).
```

```
?- send(new(@p, picture), open).
```

```
?- send(@p, display, new(@b, box(20,20))).
```

4.1.1 Trapping situations: conditional breaks

Suppose a graphical is at some point erased from the screen while you do not expect this to happen. How do you find what is causing the graphical to disappear and from where this action is called? If your program is small, or at least you can easily narrow down the possible area in your code where the undesired behaviour happens, just using the Prolog tracer will generally suffice to find the offending call.

What if your program is big, you don't know the program to well and you have no clue? You will have to work with conditional trace- and breakpoints to get a clue. If you know the reference of the offending object (lets assume `@284737`), you can spy all send-operations invoked using:

```
?- send(@vmi_send, trace, @on, full, @receiver == @284737).
```

If —for example— if you finally discovery that the object is sent the message `→device:nil`, you can trap the tracer using:

```
?- send(@vmi_send, break, @on, call, and(@receiver == @284737,  
                                         @selector == device)).
```

4.2 Exercises

Exercise 8

Consider `PceDraw`. Use the `XPCE` tracer to find out what happens to an object if the `Cut` operation is activated.

Chapter 4

Tracing and Debugging

This chapter provides a brief overview of the tracer. Besides the tracer, the VisualHierarchy and Inspector tools are very useful tools for locating problems with your application. See section 2.4.

4.1 Tracing PCE Methods and Executable Objects

XPCE offers tracing capabilities that are similar to those found in many Prolog and Lisp environments. The XPCE tracer is defined at the level of class `program_object`. Executable objects (messages, etc.), and methods are the most important subclasses of class `program_object`. Execution of `program_objects` can be monitored at three ‘ports’:

- *The call port*
Entry point of executing the object.
- *The exit port*
The object executed successfully.
- *The fail port*
Execution of the object failed.

On each port, the user can specify a **trace-** point or a **break-** point. A trace-point will cause a message printed when the port is ‘passed’. A break-point will do the same and start the interactive tracer (similar to a *spy-* point in Prolog).

The Prolog predicates `(no)tracepce/1` and `(no)breakpce/1`¹ provide a friendly interface for setting trace- and break-points on methods. Example:

```
?- tracepce((point->x)).  
Trace variable: point <->x  
  
?- new(P, point), send(P, x, 20).  
PCE: 2 enter: V @892203/point <->x: 20
```

¹The XPCE debugging predicates are defined in the library `pce_debug`. For Prolog systems lacking `autoload` you need to do `?- [library(pce_debug)]`.

```

    between(1, H, Y),
    GX is (X-1) * (CW-Pen),
    GY is (Y-1) * (CH-Pen),
    send(T, display, new(B, box(CW, CH)), point(GX, GY)),
    send(B, pen, Pen),
    new(Txt, text('', center)),
    xy_name(X, Y, XYName),
    send(Txt, name, XYName),
    send(Txt, center, point(GX + CW/2, GY + CH/2)),
    send(T, display, Txt),
fail ; true.

```

```

element(T, X:int, Y:int, Value:char_array) :->
    "Set text of cell at X-Y":
    xy_name(X, Y, XYName),
    get(T, member, XYName, Text),
    send(Text, string, Value).

```

```

xy_name(X, Y, Name) :-
    get(string('%d,%d', X, Y), value, Name).
:- pce_end_class.

```

3.4 Exercises

Exercise 5

Extend class `matrix` with a `get`-method \leftarrow *element*. Check what happens if you define the return-type incorrect (for example as `'string'`).

Exercise 6

Define a subclass `tic_tac_toe` of class `matrix` that defines the following methods:
 \rightarrow *won_position*: $\{x,o\}$, \rightarrow *play_x* and \rightarrow *play_o*.

Exercise 7

Use the `text_table` class to create a visualisation of the `tic_tac_toe` game object.

```
?- new(@matrix, matrix(3,3)).
?- send(@matrix, element, 2, 2, x).
?- send(@matrix, element, 1, 2, o).
```

We will implement the two-dimensional matrix as a subclass of the one dimensional vector:

```
:- pce_begin_class(matrix(width, height), vector, "Two-dimensional array").

variable(width,          int,    get, "Width of the matrix").
variable(height,        int,    get, "Height of the matrix").

initialise(M, Width:int, Height:int) :->
    "Create matrix fom width and and height"::
    send(M, send_super, initialise),
    send(M, slot, width, Width),
    send(M, slot, height, Height),
    Size is Width * Height,
    send(M, fill, @nil, 1, Size).

element(M, X:int, Y:int, Value:any) :->
    "Set element at X-Y to Value"::
    get(M, width, W),
    get(M, height, H),
    between(1, W, X),
    between(1, H, Y),
    Location is X * Y,
    send(M, send_super, element, Location, Value).
:- pce_end_class.
```

3.3.3.2 Defining a graphical matrix (table)

In the second example, we will define a graphical matrix of textual values and similar access functions to set/get the (string) values for the cells. We will use class device as a starting point. Class device defines a compound graphical object.

```
:- pce_begin_class(text_table(width, height), device,
    "Two-dimensional table").

initialise(T, W:int, H:int, CellWidth:[int], CellHeight:[int]) :->
    "Create from Width, Height and cell-size"::
    send(T, send_super, initialise),
    default(CellWidth, 80, CW),
    default(CellHeight, 20, CH),
    Pen = 1,
    between(1, W, X),
```

```

:- pce_begin_class(ClassName, SuperClassName, [Comment]).

<variable and method definitions>

:- pce_end_class.

```

These two predicates manipulate Prolog's macro processing and Prolog's operator table, which changes the basic syntax.

An (additional) instance variable for the class is defined using

```
variable(Name, Type, Access, [Comment]).
```

Where 'Name' is the name of the instance variable, 'Type' defines the type and 'Access' the implicit side-effect-free methods: {none,get,send,both}.

The definition of a method is very similar to the definition of an ordinary Prolog clause:

```

name(Receiver, Arg1:Type1, ...) :->
    "Comment"::
    Normal Prolog Code.

```

Defines a send method for the current class. 'Receiver', 'Arg1', ... are Prolog variables that are at runtime bound to the corresponding XPCE objects. The body of the method is executed as any normal Prolog clause body. The definition of a get method is very similar:

```

name(Receiver, Arg1:Type1, ..., ReturnValue:ReturnType) :->
    "Comment"::
    Normal Prolog Code.

```

The body is supposed to bind 'ReturnValue' to the return value of the method or fail.

3.3.2 Called methods

Some methods are called by the infra-structure of XPCE. These methods may be redefined to modify certain aspects of the class. The most commonly redefined methods are:

initialise	Called when an instance of the class is created
unlink	Called when an instance is destroyed
event	Called when an event arrives on the (graphical) object
geometry	Called when the size/position of a (graphical) object is changed

3.3.3 Examples

3.3.3.1 Defining a two-dimensional matrix

The first example defines a class two-dimensional matrix of fixed size. We want to be able to say:

message	Invoke a send-operation
assign	Variable assignment (see also 'var')
if	Conditional branch
and	Sequence and logical connective
or	logical connective
not	logical connective
while	loop
==	Equivalence test
\==	Non-Equivalence test
>, =, =<, >, >=	Arithmetic tests

and defines the following functions

?	Evaluates the result of get-operation
progn	Sequence returning value
when	Conditional function
var	Variable (returning \leftarrow <i>value</i>)
*, +, -, /	Arithmetic operations

A function is evaluated iff

1. It is the receiver of a get- or send-operation **and** the method is not defined on class function or a relevant subclass. These classes define very few methods for general object manipulation, which normally start with an '_' (underscore).
2. It is passed as an argument to a method and the argument's type-test does not allow for functions. This is true for most arguments except for behaviour that requires a function.
3. It appears as part of another code object.

3.3 Creating PCE Classes

A PCE class defines common³ storage details and behaviour of its instances (objects). A PCE class is an instance of class 'class'. Behaviour, variables and other important properties of classes are all represented using normal PCE objects. These properties allow the Prolog programmer to query and manipulate PCE's class world using the same primitives as for normal object manipulation: send(), get(), new() and free().

3.3.1 The Prolog Front End

The XPCE/Prolog interface defines a dedicated syntax for defining XPCE classes that have their method implemented in Prolog.⁴

The definition of an XPCE class from Prolog is bracketed by:

³Note that object can define individual methods and variables.

⁴The XPCE/CommonLisp interface defines a dedicated syntax for defining XPCE classes and methods where the implementation is realised by PCE executable objects. See [Anjewierden, 1992]

- *Define action associated with GUI component*

This is the oldest and most common usage. A simple example is below:

```
?- new(D, dialog('Hello')),
   send(D, append,
        button(hello, message(@prolog, format, 'Hello~n', []))),
   send(D, append,
        button(quit, message(D, destroy))),
   send(D, open).
```

- *Code fragment argument as method parameter*

The behaviour of various methods may be refined using a code object that is passed as an (optional) parameter. For example, the method ‘chain \rightarrow sort’ is defined to sort (by default) a chain of names alphabetically. When the chain contains other objects than names or sorting has to be done on another criterium, a code object may be used to compare pairs of objects. Suppose ‘Chain’ is a chain of class objects which have to be sorted by name:

```
...
send(Chain, sort, ?(@arg1?name, compare, @arg1?name)),
...
```

‘?’ is the class-name of a PCE ‘obtainer’. When executed, an obtainer evaluates to the result of a PCE get-operation: `@arg1?name2` evaluates to the return value of ‘get(@arg1, name, Name)’.

- *Implement a method*

A method represents the mapping of a ‘selector’ to an ‘action’. The action is normally represented using a PCE executable object. Define methods is discussed further in section 3.3.

3.2.1 Procedures and Functions

Like send- and get-operations, PCE defines both procedures and functions. Procedures return success/failure after completion and functions return a value (a PCE object or integer) or failure. PCE defines the following procedures:

²Equivalent to `?(@arg1, name)`: ‘?’ is a Prolog infix operator.

also chapter 7.1. Unfortunately ‘good’ style Prolog programming often manipulates data represented on the Prolog runtime stacks because destructive operations on the Prolog database are dangerous and lose two important features of Prolog: logical variables and backtracking.

The event-driven approach is commonly used by applications that present and/or edit some external (persistent) database. As long as name conflicts in the Prolog program and the global PCE name spaces (classes and named object references (e.g. `@prolog`, `@main_window`)) are avoided, multiple applications may run simultaneously in the same XPCE/Prolog process. For example the PCE manual runs together with the various PCE demo applications.

3.1.2 Asking Questions: Modal windows

Prolog applications that want to manipulate data represented on the Prolog runtime stacks cannot use the event-driven approach presented above. Unlike with event-driven applications, a single-thread Prolog system can only run one task. The overall structure of such an application is:

```
application :-
    initialise(Heap),
    create_GUI_components(Heap),
    process_events_until_computation_may_proceed,
    proceed(Heap),
    ...
```

The ‘`process_events_until_computation_may_proceed`’ is generally realised using the methods ‘`frame ← confirm`’ combined with ‘`frame → return`’. Suppose we have a diagnose system that presents a graphics representation of the system to be diagnosed. The application wants to the the user’s hypothesis for the faulty component. The window has implemented a selection mechanism, an ok button and a label to ask the question. The relevant program fragment could read:¹

```
....
send(Label, format, 'Select hypothesis and press "OK"'),
send(OkButton, message,
    message(Frame, return, Diagram?selection)),
get(Frame, confirm, Hypothesis),
....
```

3.2 Executable Objects

In the previous sections we have seen some simple examples of the general notion of ‘PCE Executable Objects’. Executable objects in PCE are used for three purposes:

¹The variables are supposed to be instantiated to the proper UI components. The construct “`Diagram?selection`” denotes a PCE ‘obtainer’. When the button is pressed, PCE will send a message `→return` to the frame. The argument to this message the return value of ‘`get(Diagram, selection, Selection)`’.

Chapter 3

Programming Techniques

3.1 Control Structure of Graphical Applications

Interactive terminal operated applications often are implemented as simple ‘Question-Answer’ dialogues: the program provides a form, the user answers the question(s) and the program continues its execution. Graphical interfaces often allow the user to do much more than just answering the latest question presented by the program. This flexibility complicates the design and implementation of graphical interfaces. Most interface libraries use an ‘event-driven’ control-structure.

3.1.1 Event Driven Applications

Using the event-driven control structure, the program creates the UI objects and instructs the graphical library to start some operation when the user operates the UI object. The following example illustrates this:

```
?- send(button(hello, message(@prolog, format, 'Hi There~n')), open).
```

Yes

This query creates a button object and opens the button on your screen. The button is instructed to call the Prolog predicate `format/1` with the given argument when it is pressed. The query itself just returns (to the Prolog toplevel).

Using this formalism, the overall structure of your application will be:

```
initialise :-
    initialise_database,
    create_GUI_components.

call_back_when_GUI_xyz_is_operated(Context ...) :-
    change_application_database(Context ...),
    update_and_create_GUI_components.
```

This type of control structure is suitable for applications that define operations on a database represented using the Prolog database or some external persistent database (see

Browsers/Group Overview provides an overview of functionally related methods. Typing in the window searches.

Tools/Visual Hierarchy provides an overview of the consists of hierarchy of GUI components.

Tools/Inspector to analyse the structure of objects.

History allows you to go back to previously visited cards.

Manpce/1 accepts a classname, in which case it switches the ClassBrowser to this class. It also accepts a term of the form ‘class \leftarrow selector’ or ‘class \rightarrow selector’, in which case it will pop up the documentation card of the indicated card. Try

```
?- manpce((display ->inform)).
```

2.5 Exercises

Exercise 1

Start the PCE manual tools and find answers to the following questions:

- (a) What is the super-class of class ‘device’?
- (b) Find the description of class ‘tree’. Which different layouts are provided by class tree?
- (c) Which attributes describe the ‘look’ of a button?
- (d) Which classes (re)define the \rightarrow report method?

Exercise 2

Examine the structure of the inheritance path visualisation as shown in the top-right window of the ClassBrowser using the VisualHierarchy tool.

Exercise 3

Class device defines compound (‘nested’) graphical objects. Define a predicate `make_text_box(?Ref, +Width, +Height, +String)`, which creates a box with a centered text object. Test your predicate using:

```
1     ?- send(new(P, picture), open),
2       make_text_box(B, 100, 50, 'Hi There'),
3       send(P, display, B, point(50, 20)).
```

Exercise 4

Class tree and class node define primitives for creating a hierarchy of graphical objects. The method ‘directory \leftarrow directories’ allow you to query the Unix directory structure. Write a predicate `show_directory_hierarchy(+DirName)` with displays the Unix directory hierarchy from the given root.

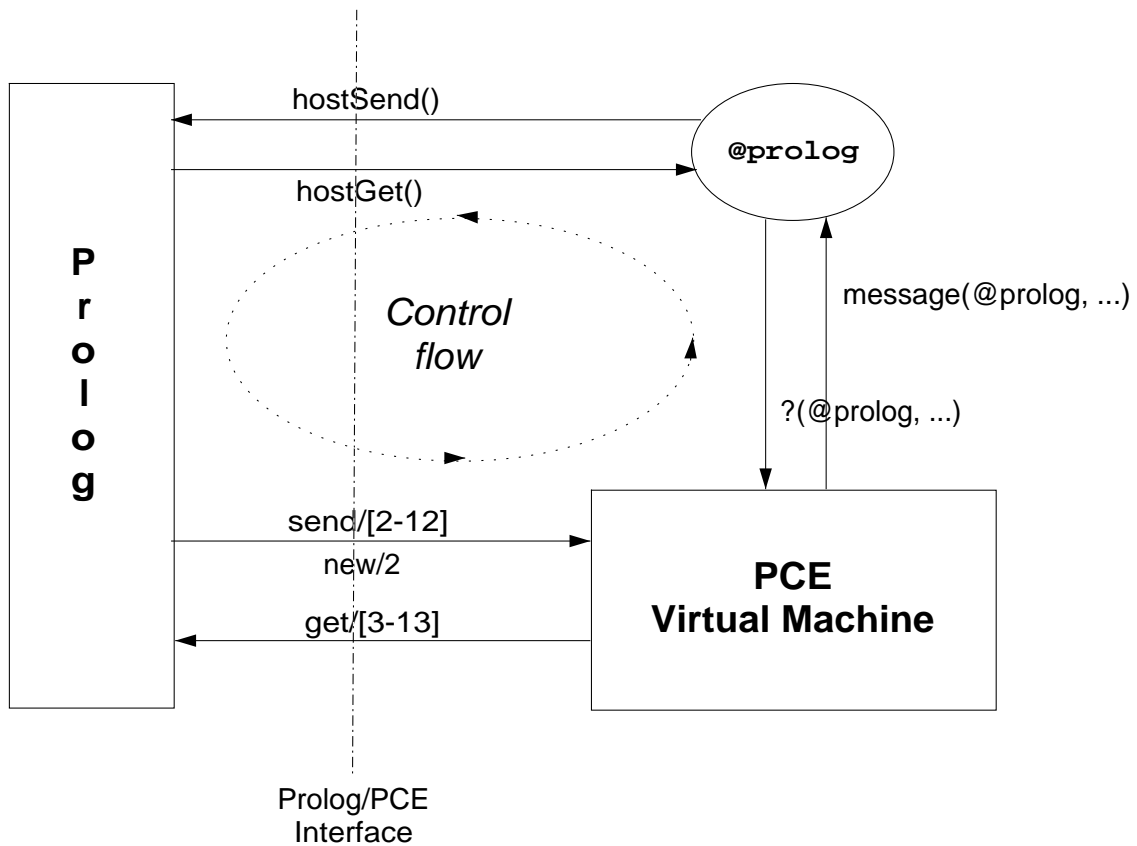


Figure 2.2: Data and Control flow in PCE/Prolog

File/Demo Programs starts a demo-starter which also provides access to the sources of the demo's.

Browsers/Class Hierarchy examines the inheritance hierarchy of PCE classes. Most classes are right below class object. Useful sub-trees are program_object (PCE's class en executable world), recogniser (event handling) and visual (anything that can appear on the screen).

Browsers/Class Browser is the most commonly used tool. It displays attributes of a class. Various searching options allow for finding candidate methods or classes. The **Scope** option needs explanation. By default it is set to 'own', making the tool search only for things (re)defined on this class. Using scope 'Super' will make the browser show inherited functionality too. Using scope 'sub' is for searching the entire class hierarchy below the current class.⁴

Browsers/Global Objects visualises all predefined (named) objects: @pce, @prolog, @display, @arg1, @event, etc.

Keywords shows a list of keywords and related information. Note that typing in the keyword list searches!

⁴The current implementation does not show *delegated* behaviour.

2.3 Architecture

XPCE is a C-library written in an object-oriented fashion.¹ This library is completely dynamically typed: each data element is passed the same way and the actual type may be queried at runtime if necessary. Also written in C is a meta-layer that describes the functions implementing the methods and the C-structures realising the storage. The meta-layer is written in the same formalism as the rest of the system and describes itself. C-functions allow for invoking methods (functions) in this library through the meta-layer. This sub-system is known as the message passing system or PCE virtual machine. This machine implements the 4 basic operations of XPCE: `new()`, `send()`, `get()` and `free()`.

The ‘host’ language (Prolog in this document) drives the PCE virtual machine instructions through the Prolog to C interface.

PCE predefines class ‘host’ and the instance `@prolog`. Messages send to this object will call predicates in the Prolog environment:

```
?- send(@prolog, format, 'Hello World~n', []).  
Hello World
```

makes Prolog call PCE `send()` through its foreign interface. In turn, the message to the `@prolog` is mapped on the predicate `format`. Note however that both systems have their own data representation. Try

```
?- send(@prolog, member, A, [hello, world]).
```

PCE has nothing that resembles a logical variable and thus will complain that ‘A’ is an illegal argument. Neither the following will work as Prolog lists have no counterpart in XPCE:²

```
?- send(@prolog, member, hello, [hello, world]).
```

And finally, Calls to PCE cannot be resatisfied:

```
?- send(@prolog, repeat), fail.  
No
```

Figure 2.2 summarises the data and control flow in the XPCE/Prolog system.

2.4 The Manual Tools

The manual tools are started using the Prolog predicate `manpce/[0,1]`³ Just typing `manpce` creates a small window on your screen with a menu bar. The most useful options from this menu are:

¹It was developed before C++ was in focus. We are considering C++ but due to the totally different viewpoints taken by XPCE (symbolic, dynamically typed) and C++ (strong static typing) it is unclear whether any significant advantage is to be expected.

²The empty list `[]` is an atom, and thus maybe passed!

³Quintus: `user_help/0`. To use `manpce/1`, load the library(`pce_manual`).

```
D = 9
```

Get *requests* the object to return (compute) a value. In this case this is the (rounded) distance to the origin of the coordinate system. The arguments to `get/[3-13]` are the same as for `send/[2-12]`, but `get/[3-13]` requires one additional argument for the return value.

Finally, the following call destroys the created object.

```
?- free(@791307).
```

```
Yes
```

2.2 Creating Windows is More Fun

In the previous section we have seen the basic operations on objects. In this section we will give some more examples to clarify object manipulation. In this section we will use windows and graphics.

```
?- new(P, picture('Hello World')),  
    send(P, display, text('Hello World'), point(20, 20)),  
    send(P, open).
```

```
P = @682375
```

First created a picture (=graphics) window labeled 'Hello World', displays a text (graphical object representing text) on the picture at location (20,20) from the top-left corner of the window and finally opens this window on the display:

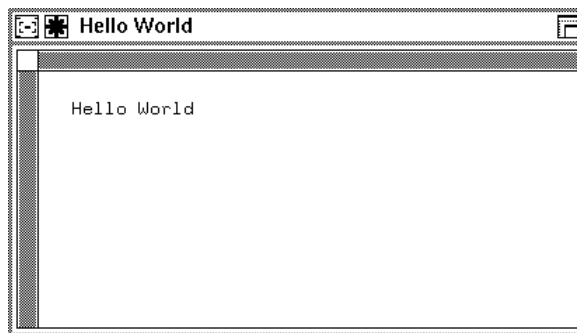


Figure 2.1: Screenshot for the 'Hello World' window

Chapter 2

Getting the Dialogue

2.1 Creating and Manipulating Objects

XPCE is an object-oriented system and defines four predicates that allows you to create, manipulate, query and destroy objects from Prolog: `new/2`, `send/[2-12]`, `get/[3-13]` and `free/1`:

```
?- new(X, point(5, 6)).
```

```
X = @791307/point
```

Created an instance of class ‘point’ at location (5,6). In general, The first argument of `new/2` provides or is unified to the *object reference*, which is a Prolog term of the functor `@/1`. The second argument is a ground Prolog term. The function describes the class from which an instance is to be created, while the arguments provide initialisation arguments.

`New/2` may also be used to create objects with a predefined object reference:

```
?- new(@s, size(100, 5)).
```

```
Yes
```

Created an instance of class `size` with width 100 and height 5. We call `@s` a ‘named’ or ‘global’ object reference.

```
?- send(@791307, x, 7).
```

```
Yes
```

Send *Manipulates* objects. In the example above, the X-coordinate of the point is changed to 7. The first argument of `send` is the object-reference. The second is the *selector* and the remaining arguments (up to 10) provide arguments to the operation.

```
?- get(@791307, distance, point(0,0), D).
```

objects that allow for elegant connections between GUI components and the application. Handling the manual and debugging tools is another subject in this course.

Besides knowing the theory, practice and looking at examples are obligatory ingredients for learning a language.

1.1 Organisation of the PCE documentation

Currently, the following documents are available for learning XPCE. In addition to these documents, the demo programs, libraries and the sources of the online manual may be examined as a source of examples.

- *Programming in PCE/Prolog*
[Wielemaker & Anjewierden, 1992b] Explains the basics of programming the PCE/Prolog environment. It should be the first document to read.
- *PCE-4 Functional Overview [Wielemaker & Anjewierden, 1992a]*
This document provides an overview of the functionality provided by PCE. It may be used to find relevant PCE material to satisfy a particular functionality in your program.
- *PCE-4 User Defined Classes Manual [Wielemaker, 1992a]*
This document describes the definition of PCE classes from Prolog.
- *PceDraw: An example of using PCE-4 [Wielemaker, 1992b]*
This document contains the annotated sources of the drawing tool PceDraw. It illustrates the (graphical) functionality of PCE. Useful as a source of examples.
- *The online PCE Reference Manual*
The paper documents are intended to provide an overview of the functionality and architecture of PCE. The online manual provides detailed descriptions of classes, methods, etc. which may be accessed from various viewpoints.

Chapter 1

Introduction

This document provides course-notes and exercises for programming in XPCE/Prolog. Some basic knowledge of Prolog is assumed. XPCE provides the following subsystems.

- *Primitives for building a GUI*
Graphical user interfaces are complicated systems. They potentially consist of a large number of different (visual) entities, possibly with complicated interrelations. XPCE provides high-level building blocks for building dialog windows and interactive graphical representations as well as the dynamics thereof.
- *The Object System*
The object system of XPCE allows the user to create, manipulate, query and destroy objects. XPCE objects represent, in addition to GUI components, various data representation primitives, classes (for defining and extending XPCE) and executable objects for relating GUI components to each other and the application.
- *The Unix process and filesystem*
These building blocks allow the user to communicate to the rest of the computing environment.
- *Debugging and statistics*
Tools allow the user to analyse the GUI as well as tracing the dynamic behaviour of GUI's.

XPCE is both a very large library and a programming environment in its own right. XPCE has similar problems as other big systems such as SmallTalk, CommonLisp, GNU-Emacs and the Unix Tool Kit. All of these environments are hard to master, just because they provide a virtually endless collection of built-in behaviour together with an elegant mechanism to combine behaviour. Novice users have little problems typing 'ls' to get a listing of the current directory. Putting all files modified after STAMP onto a tape is harder.¹

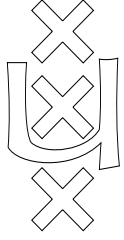
This course only deals with the basic building blocks of the XPCE environment. In addition to the building blocks we will learn the 'glue' of XPCE: XPCE's executable

¹`tar cf /dev/rst0 'find . -newer STAMP -type f -print'`

5.5	Exercises	22
6	Graphics	23
6.1	Graphical Devices	23
6.2	Making Graphicals Sensitive	24
6.2.1	Adding popup menus to graphicals	24
6.2.2	Using gestures	25
6.3	Graphs: Using Connections	26
6.4	Reading the Diagram	27
6.5	Exercises	27
7	Representation and Storing Application Data	28
7.1	Data Representation Building Blocks	28
7.2	A Simple Database	29
7.3	Exercises	31

Contents

1	Introduction	3
1.1	Organisation of the PCE documentation	4
2	Getting the Dialogue	5
2.1	Creating and Manipulating Objects	5
2.2	Creating Windows is More Fun	6
2.3	Architecture	7
2.4	The Manual Tools	7
2.5	Exercises	9
3	Programming Techniques	10
3.1	Control Structure of Graphical Applications	10
3.1.1	Event Driven Applications	10
3.1.2	Asking Questions: Modal windows	11
3.2	Executable Objects	11
3.2.1	Procedures and Functions	12
3.3	Creating PCE Classes	13
3.3.1	The Prolog Front End	13
3.3.2	Called methods	14
3.3.3	Examples	14
3.3.3.1	Defining a two-dimensional matrix	14
3.3.3.2	Defining a graphical matrix (table)	15
3.4	Exercises	16
4	Tracing and Debugging	17
4.1	Tracing PCE Methods and Executable Objects	17
4.1.1	Trapping situations: conditional breaks	18
4.2	Exercises	18
5	Dialogue Windows in Depth	19
5.1	Modal Dialogue Windows (Prompters)	19
5.2	Entering Values	20
5.3	Editing Attributes of Existing Entities	21
5.4	Status, Progress and Error Reporting	21
5.4.1	Generating reports	21
5.4.2	Handling reports	22



University of Amsterdam

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

XPCE/Prolog Course Notes

Jan Wielemaker
jan@swi.psy.uva.nl

This document provides background reading material and exercises for a course in programming XPCE/Prolog.

Copyright © 1994 University of Amsterdam