VU University Amsterdam

De Boelelaan 1081a, 1081 HV Amsterdam
The Netherlands

University of Amsterdam

Kruislaan 419, 1098 VA Amsterdam
The Netherlands

XPCE/Prolog Course Notes

Jan Wielemaker
J.Wielemaker@cs.vu.nl

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

This is edition 2 of the XPCE training-course notes. It adds sections on types, custom-dialog design and interprocess communication to edition 1. Please E-mail comments and suggestions to J.Wielemaker@vu.nl.

Typeset using LaTeX. LaTeX files created using perl preprocessors from a LaTeX extended notation for PCE documentation and raw XPCE/Prolog code. Diagrams are psfig.sty included PostScript generated by PceDraw, the XPCE/Prolog drawing tool.

An electronic version of this manual is available using anonymous ftp to swi.psy.uva.nl (145.18.114.17), directory /pub/xpce/doc/course.

Contents

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 potentionally 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 enless 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 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] `tar cf /dev/rst0 'find . -newer STAMP -type f -print'`

1.1 Organisation of the PCE documentation

Currently, the following documents are available for learning XPCE. In addition tothese 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 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).

D = 9
```

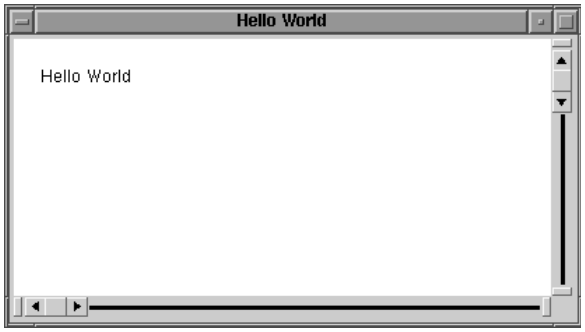

Figure 2.1: Screendump for the 'Hello World' window

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:

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

[1] It 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.

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:

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.

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

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

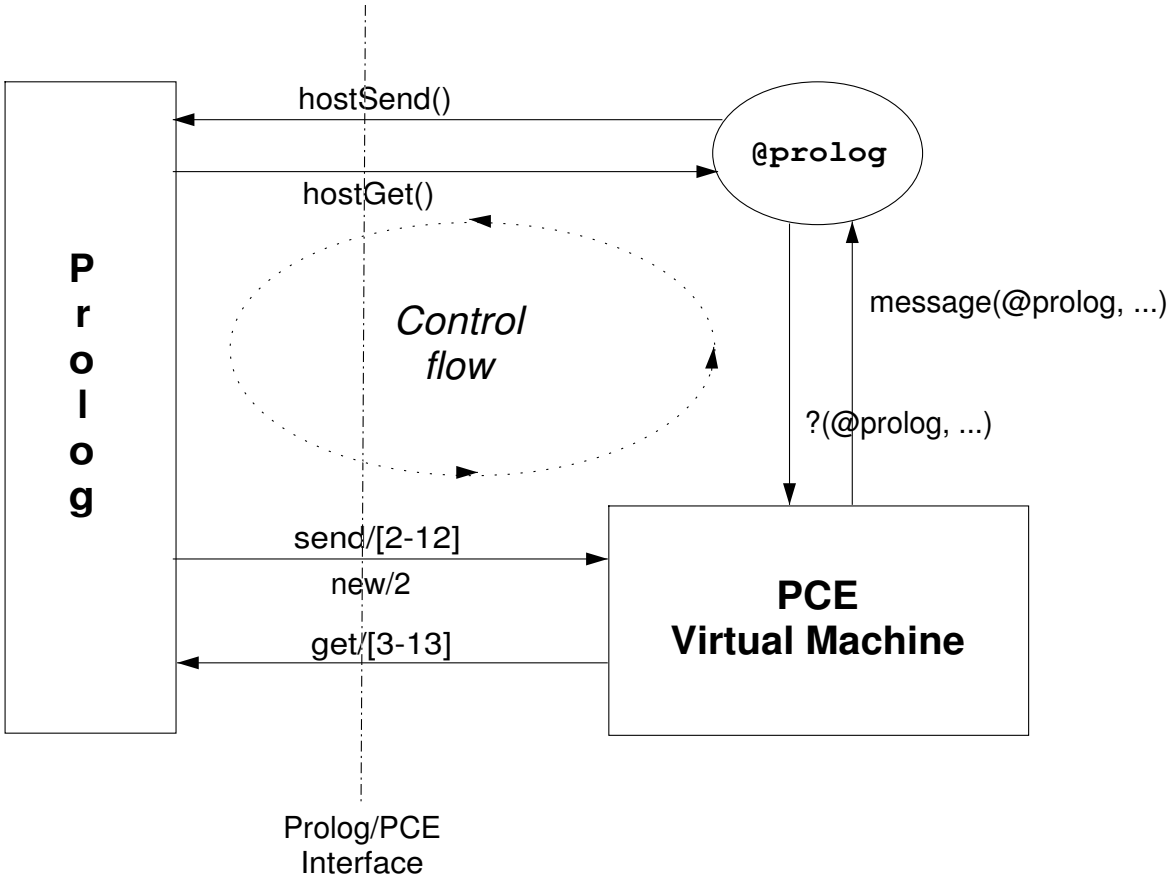

Figure 2.2: Data and Control flow in PCE/Prolog

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.

Browsers/Search Provides a WAIS search facility for the entire hypercard system.

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 ← selector`' or '`class → 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 appearance of a box?

(d) Which classes (re)define the →`report` method?

Exercise 2

Examine the structure of the inheritance path visualisation as shown in the top-right window of the ClassBrowser using the VisualHierachy 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:

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

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

Exercise 4

Class tree and class node define primitives for creating a hierarchy of graphical objects. The method '`directory ← 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.

Some methods and classes you might want to use are: class picture, class tree, class node, class directory and the methods '`node → son`', '`directory ← directories`' and '`directory ← directory`'.

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 also chapter 8.1. Unfortunately 'good' style Prolog programming often manipulates data represented on the Prolog runtime

stacks because destructive operations on the Prolog database are dangerous and loose 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-threat 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:

- Define action associated with GUI component
  This is the oldest and most common usage. A simple example is below:

[1] The variables are supposed to the 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)'.

```
?- 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 → 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`?name[2] 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:

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

[2] Equivalent to ?(`@arg1`, name): '?' is a Prolog infix operator.

?	Evaluates the result of get-operation
progn	Sequence returning value
when	Conditional function
var	Variable (returning ←_value)
***, +, -, /**	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 common[3] 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:

```
:- 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]).
```

[3] Note that object can define individual methods and variables.

[4] The 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]

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

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:

3.3.3 Examples

Defining a two-dimensional matrix

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

```
?- 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 * (W-1)),
        send(M, send_super, element, Location, Value).
:- pce_end_class.
```

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),
            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 ←`element`. See what happens if you define the return-type incorrect (for example as 'int').

Exercise 6

We would like to *visualise* a matrix using a text_table object, such that changes to the matrix are reflected on the text_table and modifications of the text_table are reflected in the matrix.

XPCE does not provide a built-in model/controller architecture such as SmallTalk. There are various ways to relate objects to each other: object-level attributes (see '`object` ⇆ `attribute`'), instance-variables and finally 'hyper-links'. See class hyper.

Try to realise the mapping using hyper objects. See class hyper and '`object` → `send_hyper`'. What are the (dis)advantages of all these techniques?

Exercise 7

Advanced usage! Suppose the matrix is used in heavy computations and modified often. This would imply passing through the mechanism described above many times. The display is not updated for each intermediate state (unless you explicitly request this using 'graphical →`flush`').

To avoid part of the computation as well, you can use the mechanism described by '`graphical` → `request_co`
and '`graphical` → `compute`'.

Try to implement this interface.

3.5 Types

XPCE is an untyped language like Prolog. This allows the system to define container classes such as class chain, vector, hash_table etc. in a comfortable fashion. Unlike Prolog however, XPCE **allows** you to declare types. Types are used for four purposes:

- Documentation
  Given the name, argument-names and argument-types for each method generally suffices to get a strong clue on what it does.

- Run-time trapping of errors
  The earlier errors are trapped, the easier it is to locate the errornous statement in your source!

- Run-time conversion
  Each type defines a validation method and a conversion method. If the first fails, the second is tried to convert the input value to the requested type. For example if the expected argument is of type 'int' and you provide the string "554" it will gracefully convert "554" to the integer 554 and pass this value to the implementation of the method.

- Ensuring some level of consistency in the data
  The predicate `checkpce/0` collects all instances and verifies that each instance-variable is consistent with the type-declaration.

3.5.1 Types are not (only) classes

Types are first-class objects that defined *validation* and *conversion*. Type objects are seldomly defined using new(X, type(...)), but normally by conversion of a type-description to a type object. Here is a brief summary of the syntax used by '`type ← convert`':

`Name=Type`	Argument named 'Name' of type 'Type'
`Type ...`	Zero or more of these arguments. Used in variable-arguments methods.
`Type1|Type2`	Disjunction (either of the types)
`Type*`	The type or the constant `@nil`
`[Type]`	The type or `@default` (optional argument)
{`Name1,Name2`}	One of these names
`int`	An integer (the only non-object datum)
`3..5`	An integer $3 \leq$ value ≤ 5
`3.2..5.6`	A real (float) in this range
`3..`	An integer ≥ 3
`..3`	An integer ≤ 3
`ClassName`	Any instance of this class (or sub-class)
`alien:char *`	Alien (C) data. In this case a char *

3.5.2 Programming conversion

The reserved get-method ←`convert` has the task to convert an incomming object to an instance of the specified class.

3.6 Exercises

Exercise 8

If an argument is specified as type 'bool', what values are acceptable as input.

Exercise 9

Advanced usage: A very common usage for conversion is a class with uniquely named reusable objects. Try to define such a class. You will have to define the methods →`initialise`, ←`lookup` and ←`convert`.

Chapter 4

Tracing and Debugging

This chapter provides a brief overview of the tracer. Besides the tracer, the VisualHierachy 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
PCE:  2 exit: V @892203/point <->x: 20
```

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

```
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—you finally discovered 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 10

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

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 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 ← confirm`'.

This method behaves as '`frame → open`' and next starts an event-dispatching subloop until '`frame → 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).
```

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 →`append`: menu_item. Class menu_item defines a type-conversion converting any value to a new menu_item using the value as '`menu_item ⇆ value`'. The visible labels of the menu_items are generated automatically (see 'menu_item ←`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.

5.3 Editing Attributes of Existing Entities

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

Class dialog defines the methods →restore and →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 →report and the get-method ←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)
        ).
```

The first argument of →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 →`alert` mechanism (→`bell`, →`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 →`report` or ←`report_to` can improve error reporting.

5.5 Exercises

Exercise 11

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.

5.6 Custom Dialog-Items

As from XPCE version 4.7 (ProWindow 1.1), graphics and dialog_items are completely integrated. This implies that custom dialog-items can be constructed using general graphics as well as other dialog-items.

Dialog-items should respond in a well-defined manner to a number of messages and have standard initialisation arguments. These methods are:

item → initialise: Name:name, Default:[any], Message:[code]*
Items are normally created from their *Name*, which is processed by '`char_array ← label_name`' to produce a capitalised visible label, the *Default* value and a message to execute on →`apply` or operation of the item.

item → selection: Value:any
Sets the selection. After this, ←`modified` is supposed to return `@off`.

item ← selection: Value:any → R
eturn the current selection.

item → default: Value:any
Sets the *default* value and the →`selection`. If →`restore` is called after the user modified the item and before it is →`apply`'d, the ←`selection` should be restored to the ←`default`. Normally called by →`initialise`. Normally implemented by setting the -default slot and calling →`restore`.

item → restore
Normally simply invokes →`selection` with ←`default`.

item → apply: Force:[bool]
If ←`modified` returns `@on` or Force equals `@on`, forward the ←`selection` over `@arg1` of the ←`message`.

The XPCE library ('`@pce ← home`'/prolog/library) contain various examples of custom dialog items. See for example pce_font_item.pl (consists of three cycle-menus for specifying a PCE font), pce_style_item.pl (enter a style (font, colour, underline, etc.) for an editor-fragment). The dialog editor contains various examples as well.

5.7 Exercises

Exercise 12
Study the pce_font_item.pl file. Use the font_item to control the font of a simple text object. Analyse the structure of the application using the Visual Hierarchy tool.

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,   'size(100, 50)',        "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).
```

```
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(pce_prompter)).

make_graph_editor(E) :-
        new(E, picture('Graph Editor')),
        send(E, popup, new(P, popup(options))),
        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) :-
        prompter('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 → recogniser`' which makes the gesture work for a single graphical object. The alternative is to define an →`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),
                              end_group := @on),
                    menu_item(delete,
                              message(TB, free))
                  ]).
```

[1] In 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.

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


rename(TB) :->
        "Prompt for new name"::
        prompter('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(@text_box_recogniser, event, Ev)
        ;   send(@link_recogniser, event, Ev)
        ).

:- 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,
                        @arg1?to?string?value))).

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

6.5 Exercises

Exercise 13

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

Exercise 14

The example in section 6.3 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 15

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 16

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

Chapter 7

Multiprocess Applications

In this chapter we will discuss the usage of XPCE-4 for applications spread over multiple Unix processes, possibly running on multiple systems. We distinguish the following designs:

- Calling non-interactive data-processing applications
  With 'non-interactive data-processing applications' we refer to applications that take an input specification from a file and dump the result of there activities on another file without user interaction.
- Front-end for terminal-based interactive applications
  Traditional examples from the Unix/X11 world are xdbx/xgdb as a graphical front-end for dbx/gdb and various graphical mailtools acting as front-ends for their terminal based counterparts.
- Client-Server applications
  In these applications, XPCE/Prolog will generally be used as a graphical front-end for one or more applications to which it communicates over the network.

7.1 XPCE building blocks

XPCE provides the application programmer with the following building-blocks for multi-process and/or multi-user applications:

- Class process
  Class process defines the interaction between XPCE and both synchronously and asynchronously running Unix processes. Communication can be provided using Unix *pipes* as well as using a Unix *pseudo-tty*.

  The process controlled this way can only be started as a child process of XPCE/Prolog. Class *socket* discussed below allows for non-child-process communication.
- Class socket
  The XPCE class socket makes Unix sockets available to the XPCE programmer. It supports both Unix-domain sockets (sockets that make an *end-point* in the Unix file-structure and can only be used to communicate between processes on the same machine) and *inet* domain sockets (sockets that make an *internet* end-point and can be used to communicate with other processes running anywhere on the internet).

- Class display and class display_manager
  XPCE can communicate with multiple X11-displays simultaneously.[1] Thus, cooperative applications (*CSCW*) may be implemented as client-server applications as well as using a single XPCE/Prolog application managing windows for multiple users.

7.2 Calling non-interactive data-processing applications

There are various options for calling such applications:

- Using class process
  The communication between the child-process and XPCE is arranged using Unix pipes, and optionally using a Unix pseudo-tty. Data may be sent and received incrementally. XPCE can assign a call-back to be executed for newly available data and/or termination of the process or wait for either of these events.

- Using Prolog's `unix/1`
  Generally only allows for the schema "prepare input-file, call process, parse output-file". On most Prolog systems, XPCE event-dispatching will not ocur during execution of the external application.

The proper choice from the above depends on the characteristics of the application:

- The application can read/write to Unix standard I/O
  This will allow you to handle the output incrementally as it is produced by the external process, simutaneously handling user-events.

- Execution time of the application
  Applications with short execution-times may be called using Prolog's `unix/1` predicate as well as using XPCE process objects with XPCE waiting for termination of the application.

- Amount of data produced, nature and destination of it
  Can output of the application be parsed in separate records?

7.3 Examples

The examples below generally take trivial standard Unix applications. Many of the manipulations could be established inside Prolog or XPCE.

7.3.1 Calling simple Unix output-only utilities

In this example, we will define the Prolog predicate `call_unix/3`, which takes the following arguments:

- Name of the utility

[1] Provided the application is purely event-driven

- List of arguments to it
- Output: Prolog list of ASCII values

```
call_unix(Utl, Args, Output) :-
        NewTerm =.. [process, Utl | Args],
        new(P, NewTerm),
        send(P, use_tty, @off),
        send(P, open),
        new(OS, string),
        repeat,
            (   get(P, read_line, Line)
            ->  send(OS, append, Line),
                fail
            ;   !
            ),
        pce_string_to_list(OS, Output).

pce_string_to_list(S, L) :-
        pce_string_to_list(S, 0, L).

pce_string_to_list(S, I, [C|T]) :-
        get(S, character, I, C), !,
        NI is I + 1,
        pce_string_to_list(S, NI, T).
pce_string_to_list(_, _, []).
```

This version of the call will block and not dispatch X11 events during the execution of `call_unix/3`. Below we use another definition that processes normal user events during the execution. We borrow the definition of `pce_string_to_list/2` from the previous example.

The call '`process → wait`' runs the XPCE event-loop until all data from the process has been handled.

```
call_unix(Utl, Args, Output) :-
        NewTerm =.. [process, Utl | Args],
        new(P, NewTerm),
        send(P, use_tty, @off),
        new(OS, string),
        send(P, record_separator, @nil),
        send(P, input_message, message(OS, append, @arg1)),
        send(P, open),
        send(P, wait),
        pce_string_to_list(OS, Output),
        send(P, done),
        send(OS, done).
```

7.4 Exercises

Exercise 17

What is the function of '`process` → `record_separator`' and why is it set to `@nil` in the example above?

Exercise 18

Write a predicate call_unix(*+Utility, +Args, +Input, -Output*) that pipes the data from the Prolog string *Input* through the command and unifies *Output* with a Prolog string representing the output of the process. Use `call_unix/3` above as a starting point.

7.5 Front-end for terminal-based interactive applications

In the example for this type of application we will define a simple front-end for the Unix ftp(1) program. Below is a screen-dump of this demo-application:

```
1  /*  Part of XPCE
2      This example code is in the public domain
3  */
4  :- module(ftp,
5            [ ftp/0,                      % just start it
6              ftp/1,                      % ... and connect as 'ftp'
7              ftp/2                       % ... and connect as user
8            ]).
9  :- use_module(library(pce)).
10 :- require([ atomic_list_concat/2
11             , ignore/1
12             , maplist/3
13             , reverse/2
14             , send_list/3
15             ]).
16 ftp :-
17         new(_, ftp_frame).
18 ftp(Address) :-
19         new(_, ftp_frame(Address)).
20 ftp(Address, Login) :-
21         new(_, ftp_frame(Address, Login)).
```

Like most tools, the tool as a whole is represented by a subclass of class frame. The major advantage of this approach is that all visual parts are in a natural way 'part' of the tool and each of them can easily find the overall tool using ←`frame`, which is defined on most visual classes.

```
22 :- pce_begin_class(ftp_frame, frame, "Ftp interaction").
23 variable(home,  name,           get, "The home directory").
24 initialise(F, Address:[name], Login:[name]) :->
25         "Create frame [and connect]"::
26         send(F, send_super, initialise, 'FTP Tool'),
27         send(F, append, new(DT, dialog)),
```

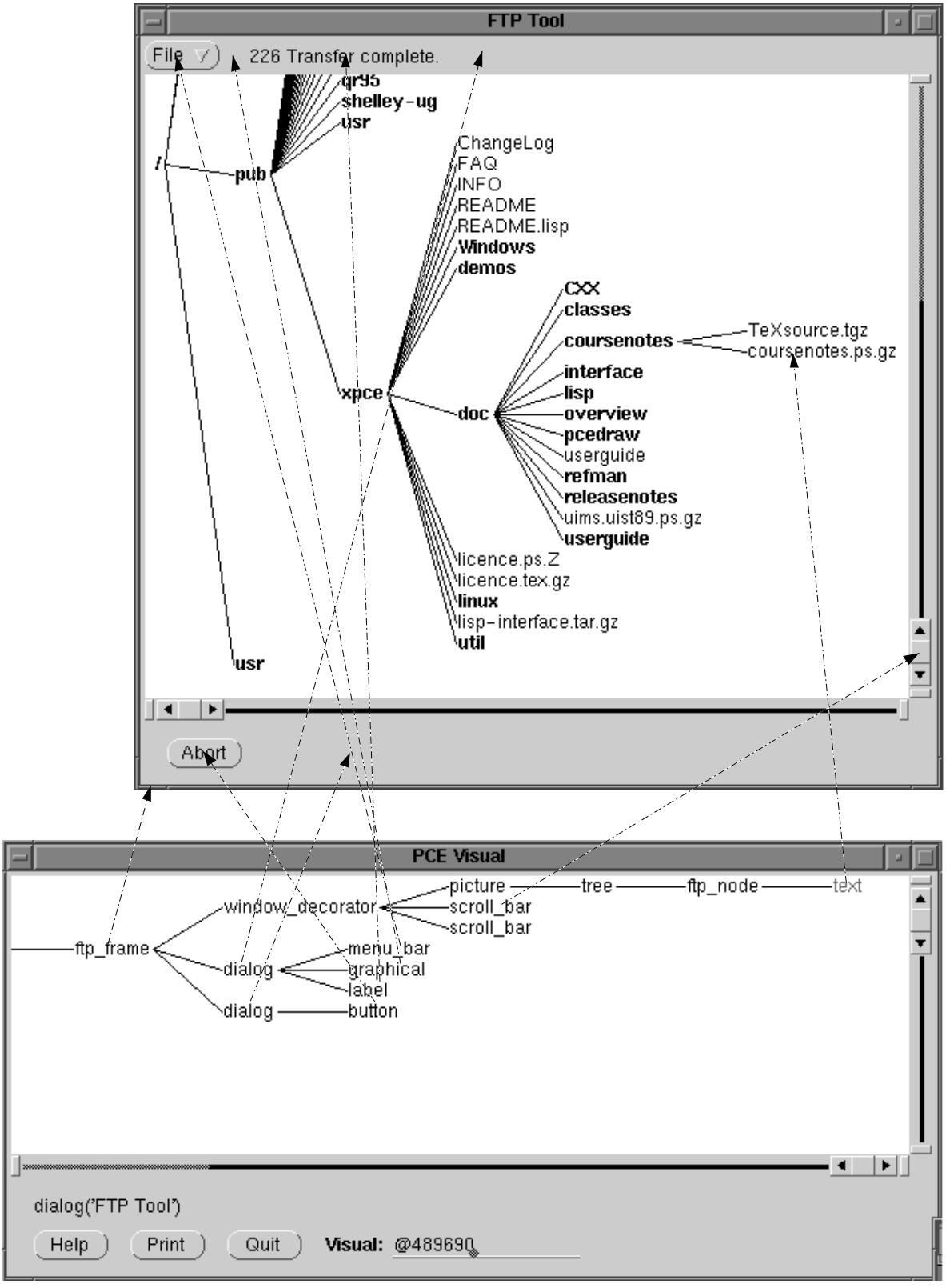

Figure 7.1: Screendump for the ‘FTP tool’

```
28          send(new(P, picture), below, DT),
29          send(new(DB, dialog), below, P),
30          fill_top_dialog(DT),
31          fill_picture(P),
32          fill_bottom_dialog(DB),
33          send(F, open),
34          (   Address == @default
35          ->  true
36          ;   send(F, connect, Address, Login)
37          ).
```

`fill_top_dialog/1` creates the top menu-bar. There is only one item in this incomplete tool. The →`pen` and →`gap` of the dialog window are zeroed to make the menu-bar appear nicely above the application window. A label is put right to the menu-bar. A 'reporter' label is used by the general →`report` mechanism and will display all errors and warnings produced by the tool.

```
38  fill_top_dialog(D) :-
39          get(D, frame, Tool),
40          send(D, pen, 0),
41          send(D, gap, size(0,0)),
42          send(D, append, new(MB, menu_bar)),
43          send(D, append, graphical(width := 20), right), % add space
44          send(D, append, label(reporter), right),
45          send(MB, append, new(File, popup(file))),
46          send_list(File, append,
47                    [ menu_item(quit,
48                                message(Tool, destroy))
49                    ]).
```

The application window is a graphics window itself (picture) consists of a tree holding 'ftp_node' objects. We turned this into a class because much of the basic functionality of the tool (expanding, collapsing, viewing, etc. are naturally defined as operations on the nodes of the tree.

```
50  fill_picture(P) :-
51          send(P, display, new(T, tree(ftp_node(directory, /)))),
52          send(T, node_handler, @ftp_node_recogniser).
```

The bottom-dialog window is used to place some commonly used push-buttons.

```
53  fill_bottom_dialog(D) :-
54          get(D, frame, Tool),
55          send(D, append, button(abort, message(Tool, abort))).
```

The →`unlink` method is called when the frame (= tool) is destroyed. It ensures that the ftp-connection is terminated before removing the frame. →`unlink` *cannot* stop the unlinking process: it must succeed and it must call →`send_super`: unlink.

```
56  unlink(F) :->
57          "Make sure to kill the process"::
58          ignore(send(F, disconnect)),
59          send(F, send_super, unlink).
```

Part of the tool may be found using the ←member method. It is good programming practice to make methods for finding commonly used parts so that the structure of the tool can remain hidden for the outside world.

```
tree(F, Tree:tree) :<-
        "Find the tree part of the interface"::
        get(F, member, picture, P),
        get(P, member, tree, Tree).

home(F, Home:name) :->
        "Set the home directory"::
        send(F?tree?root, string, Home),
        send(F, slot, home, Home).
```

The ftp-process is connected using a 'hyper' link. Alternatives are the use of attributes or instance-variables. The advantage of using hyper-links is that they are bi-directional, safe with regard to destruction of either end-point and the semantics of destructions may be defined on the link rather than on the each of the connected objects. See 'hyper →unlink_to' 'hyper →unlink_from'.

```
ftp(F, P:ftp_process*) :->
        (   P \== @nil
        ->  new(_, hyper(F, P, ftp, tool))
        ;   get(F, find_hyper, ftp, Hyper),
            free(Hyper)
        ).
ftp(F, P:ftp_process) :<-
        get(F, hypered, ftp, P).
```

Connect simply creates an 'ftp_process' object, which automatically will login on the remote machine. After the connection is established, it will obtain the root-directory using the pwd command. See the method 'ftp_process →pwd' below. The message argument is the message that should be called when the ftp's pwd command completes. See description on ftp_process class below for the communication details.

```
connect(F, Address:name, Login:[name]) :->
        "Connect to address"::
        send(F, disconnect),
        send(F, ftp, new(P, ftp_process(Address, Login))),
        send(P, pwd, message(F, home, @arg1)).

disconnect(F) :->
        "Close possible open connection"::
        (   get(F, ftp, Process)
        ->  send(Process, kill),
            send(F, ftp, @nil)
        ;   true
        ).

abort(F) :->
        "Send Control-C to the ftp process"::
        get(F, ftp, Process),
        send(F, report, status, 'Sending SIGINT to ftp'),
```

```
92          send(Process, kill, int).
93  :- pce_end_class.
```

Node in the ftp-directory/file hierarhy. Directory nodes are printed bold, file nodes in normal roman font. This class defines handling of a selection (inverted item), attaching a popup menu and the basic operations (expand, collapse, view, etc.).

```
94  :- pce_global(@ftp_node_recogniser, make_ftp_node_recogniser).
95  make_ftp_node_recogniser(R) :-
96          Node = @receiver,
97          new(S1, click_gesture(left, '', single,
98                                and(message(Node?device, for_all,
99                                            message(@arg1, inverted, @off)),
100                                   message(Node, inverted, @on)))),
101         new(S2, click_gesture(left, s, single,
102                               message(Node, inverted,
103                                       Node?inverted?negate))),
104         new(E1, click_gesture(left, '', double,
105                               and(message(Node, expand),
106                                   message(Node, inverted, @off)))),
107         PopNode = @arg1,
108         new(P, popup_gesture(new(Pop, popup(options)))),
109         send_list(Pop, append,
110                   [ menu_item(expand,
111                               message(PopNode, expand),
112                               condition := PopNode?type == directory),
113                     menu_item(collapse,
114                               message(PopNode, collapse),
115                               condition := not(message(PopNode?sons, empty)))
116                   ]),
117         new(R, handler_group(S1, S2, E1, P)).
118 :- pce_begin_class(ftp_node, node, "Node in the ftp-file-tree").
119 variable(type,  {file,directory},       get,    "Type of the node").
120 initialise(N, Type:{file,directory}, Name:name, Size:[int]) :->
121         "Create from type, name and size" : :
122         (   Type == directory
123         ->  Font = font(helvetica, bold, 12)
124         ;   Font = font(helvetica, roman, 12)
125         ),
126         new(T, text(Name, left, Font)),
127         send(N, send_super, initialise, T),
128         send(N, slot, type, Type),
129         (   Size == @default
130         ->  true
131         ;   send(N, attribute, attribute(file_size, Size))
132         ).
```

Deduce the path of the node by appending the components upto the root.

```
path(N, Path:name) :<-
        "Get path-name"::
        node_path(N, L0),
        reverse(L0, L1),
        insert_separator(L1, /, L2),
        atomic_list_concat(L2, Path).

node_path(N, [Me|Above]) :-
        get(N?image?string, value, Me),
        (   get(N?parents, head, Parent)
        ->  node_path(Parent, Above)
        ;   Above = []
        ).

insert_separator([], _, []).
insert_separator([H], _, [H]) :- !.
insert_separator([H|T0], C, [H, C | T]) :-
        insert_separator(T0, C, T).
```

Expand means: If it is a directory, cd the ftp-server to the directory of this node and issue an 'ls' command. Parse the output line-by line using the given message.

If the node is a file, 'view' it: get the file in a local tmp file and start an XPCE view on it.

```
expand(N) :->
        "Get the sub-nodes"::
        (   get(N, type, file)
        ->  send(N, view)
        ;   get(N, path, Path),
            get(N?frame, ftp, Process),
            send(Process, cd, Path),
            send(N, collapse),
            send(Process, ls,
                 message(N, son, create(ftp_node, @arg1, @arg2, @arg3)))
        ).
```

The method →`son` is redefined to make sure the new node is visible. The method 'window →`normalise`' ensures the visibility of a graphical, set of graphicals or area of the window.

```
son(N, Node:ftp_node) :->
        "Add a son and normalise"::
        send(N, send_super, son, Node),
        send(N?window, normalise, Node?image).

collapse(N) :->
        "Destroy all sons"::
        send(N?sons, for_all, message(@arg1, delete_tree)).

view(N) :->
        "Read the file"::
        get(N, path, Path),
        get(N?frame, ftp, Process),
        send(Process, view, Path).

:- pce_end_class.
```

The class ftp_process defines the interaction with the ftp process. It is written such that it can easily be connected in another setting.

Communication is completely asynchronous. This approach is desirable as FTP servers sometimes have very long response-times and there is no reason to block the interface in the mean-while.

In general, when an ftp-command is issued, it will:

1. Wait for the ftp-process to get to the prompt. It dispatches events (keeping other applications happy) while waiting.
2. Set the 'state' variable indicating the command and the 'action_message' indicating what to do with the output and finally send the command to the server.

Subsequent data from the ftp-process will be handled by the →input method, which parses the data according to ←state and invokes the ←action_message on some patterns.

```
:- pce_begin_class(ftp_process, process, "Wrapper around ftp").

variable(state,          name,   get,    "Current command state").
variable(login,          name,   get,    "Login name").
variable(action_message,[code], get,    "Message to handle output").
```

Initialise the ftp_process. Noteworthy are the →record_separator, which will combine and/or break the data-records as received from the process into lines. It knows about the two prompts and will consider those to be a separate record. Finally it preperes the login command.

```
initialise(P, Host:name, Login:[name]) :->
        "Create ftp process and connect"::
        send(P, send_super, initialise, ftp, Host),
        send(P, record_separator, string('^ftp> \\|^Password:\\|\n')),
        send(P, input_message, message(P, input, @arg1)),
        send(P, slot, action_message, @default),
        send(P, open),
        default(Login, ftp, TheLogin),
        send(P, slot, login, TheLogin),
        send(P, slot, state, login),
        send(P, format, '%s\n', TheLogin).
```

Password stuff. As a little hack, the password entry-field is unreadable by using a font that produces no readable output. This is the simplest solution. A more elegant solution would be to use two text-entry-fields: one that is visible and one that is not visible. The visible one could pass all editing commands to the invisible one to do the real work. XPCE is not designed for this kind of things ...

```
give_pass(P) :->
        "Answer the password prompt"::
        get(P, login, Login),
        getpass(Login, Passwd),
        send(P, format, '%s\n', Passwd).

getpass(ftp, EMail) :- !,
        email(EMail).
getpass(anonymous, EMail) :- !,
        email(EMail).
```

```
getpass(User, Passwd) :-
        new(D, dialog('Enter Password')),
        send(D, append, new(T, text_item(User, ''))),
        send(T, value_font,
             font(screen, roman, 2,
                  '-*-terminal-medium-r-normal-*-2-*-*-*-*-*-iso8859-*')),
        send(D, append, button(ok, message(D, return, T?selection))),
        send(D, append, button(cancel, message(D, return, @nil))),
        send(D, default_button, ok),
        get(D, confirm_centered, RVal),
        send(D, destroy),
        Passwd = RVal.

email(EMail) :-
        get(@pce, user, User),
        get(@pce, hostname, Host),
        new(EMail, string('%s@%s', User, Host)).
```

This is a case where we have to help the reporting system a bit. By default, reports for non-visual objects are forwarded to the visual object that handles the current event. On call-backs from the process input however, there is no current event and messages will go to the terminal. The method ←`report_to` ensures they are forwarded to the associated tool. See also 'ftp_frame →`ftp`'.

```
report_to(P, Tool:frame) :<-
        "→report to the associated tool"::
        get(P, hypered, tool, Tool).
```

Handling input from the process (ftp) is the central and most tricky bit of this application. The method →`input` will scan through the patterns for one matching the input (which is broken in lines). When a match is found, the action part is translated into a message:

- The functor is the selector on this class
- The arguments are arguments to the message. Arguments of the form digit:type are replaced by the n-th register of the regular expression converted to the indicated type.

Thus, action(1:name) will invoked 'ftp_process →`action`' with the first register of the regular expression converted to a name.

```
input(P, Input:string) :->
        get(P, state, State),
        pattern(State, Pattern, Action),
        to_regex(Pattern, Regex),
        send(Regex, match, Input), !,
        Action =.. [Selector|Args],
        maplist(map_pattern_arg(Regex, Input), Args, NArgs),
        Message =.. [send, P, Selector | NArgs],
        Message.

pattern(_,
        '^ftp> $',
        prompt).
pattern(ls,
```

```
229         '[-l].* \\(\\sd+\\) [A-Z][a-z][a-z].* \\([^ \n]+\\)$',
230         action(file, 2:name, 1:int)).
231 pattern(ls,
232         'd.* \\([^ \n]+\\)$',
233         action(directory, 1:name)).
234 pattern(ls,
235         '^total \\sd+$\\|^\\sd+ bytes received\\|remote: -l',
236         succeed).
237 pattern(pwd,
238         '257[^"]*"\\([^"]+\\)',
239         action(1:name)).
240 pattern(login,
241         '^Password:',
242         give_pass).
243 pattern(view,
244         '226 Transfer complete.',
245         action).
246 pattern(_,
247         '\\sd+.*\\.$',
248         report(status, 0:string)).
249 pattern(_,
250         '2[35]0-\\(.*\\)',
251         message(1:string)).
252 pattern(_State,
253         '.*',
254         message(0:string)).
255 %       report(warning, 'Unrecognised (%s): %s', State, 0:string)).

256 :- dynamic mapped_to_regex/2.

257 to_regex(Pattern, Regex) :-
258         mapped_to_regex(Pattern, Regex), !.
259 to_regex(Pattern, Regex) :-
260         new(Regex, regex(string(Pattern))),
261         send(Regex, lock_object, @on),
262         asserta(mapped_to_regex(Pattern, Regex)).

263 map_pattern_arg(Regex, Input, Reg:Type, Value) :- !,
264         get(Regex, register_value, Input, Reg, Type, Value).
265 map_pattern_arg(_, _, Value, Value).
```

The method →`action` is called from →`input` to have data handled by the caller. Doing

```
send(P, pwd, message(Tool, home, @arg1))
```

will, when the pwd-state pattern mathes, call 'ftp_process →`action`: PWD', which in turn will execute the given message using the PWD argument.

```
266 action(P, Args:any...) :->
267         "Perform action"::
268         get(P, action_message, Msg),
269         (   Msg == @default
270         ->  send(@pce, send_vector, write_ln, Args)
271         ;   send(Msg, forward_vector, Args)
```

```
272         ).

273 succeed(_P) :->
274         "Just succeed"::
275         true.
```

If we get back to the prompt, all input is handled and we will clear the action_message.

```
276 prompt(P) :->
277         "Has reverted to the prompt"::
278         send(P, slot, state, prompt),
279         send(P, slot, action_message, @default).
```

Informative messages from the ftp process are processed by this method. It pops up a view for displaying the messages. Note once more the usage of a hyper, which ensures the user can discard the view without introducing inconsistencies.

```
280 message(P, Message) :->
281         "Handle informative messages"::
282         (   get(P, hypered, message_view, V)
283         ->  View = V
284         ;   new(_, hyper(P, new(View, view('FTP message')),
285                          message_view, ftp)),
286             send(View, confirm_done, @off),
287             send(new(D, dialog), below, View),
288             send(D, append, button(quit, message(View, destroy))),
289             send(D, append, button(clear, message(View, clear))),
290             send(View, open)
291         ),
292         send(View, appendf, '%s\n', Message).
```

The method →wait_for_prompt runs the main XPCE event-loop using 'display →dispatch' until the ftp-process gets in the 'prompt' ←state.

```
293 wait_for_prompt(P) :->
294         "Process input till prompt"::
295         send(P, report, progress, 'Waiting for ftp prompt'),
296         repeat,
297         (   get(P, state, prompt)
298         ->  !,
299             send(P, report, done)
300         ;   send(@display, dispatch),
301             fail
302         ).
```

The basic commands. Given all the infra-structure above, these are quite simple now!

```
303 ls(P, Msg:[code]) :->
304         send(P, wait_for_prompt),
305         send(P, slot, state, ls),
306         send(P, slot, action_message, Msg),
307         send(P, format, 'ls -l\n').
```

```
pwd(P, Msg:[code]) :->
        send(P, wait_for_prompt),
        send(P, slot, state, pwd),
        send(P, slot, action_message, Msg),
        send(P, format, 'pwd\n').

cd(P, Dir:char_array) :->
        send(P, wait_for_prompt),
        send(P, slot, state, cd),
        send(P, format, 'cd %s\n', Dir).

view(P, Path:char_array) :->
        "Start view on contents of file"::
        send(P, wait_for_prompt),
        new(Tmp, string('/tmp/xpce-ftp-%d', @pce?pid)),
        send(P, slot, action_message, message(P, make_view, Path, Tmp)),
        send(P, slot, state, view),
        send(P, format, 'get %s %s\n', Path, Tmp).

make_view(_P, RemoteFile:name, LocalFile:file) :->
        "Make a view for displaying file"::
        new(V, view(string('FTP: %s', RemoteFile))),
        send(V, load, LocalFile),
        send(LocalFile, remove),
        send(V, confirm_done, @off),
        send(V, open).

:- pce_end_class.
```

7.6 Exercises

Exercise 19

Add a facility to download a file to your current directory.

Chapter 8

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.

8.1 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:

- chain
  A chain is a single-linked list. Class chain defines various set-oriented operations and iteration primitives.

[1] Using 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.

- 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.

- 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.

8.2 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").

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"::
```

[2] Unfortunately 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).

```
        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).

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.
```

8.3 Exercises

Exercise 20

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 21

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 22

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 →`unlink` method.

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.