UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Logic programming for knowledge-intensive interactive applications

Wielemaker, J.

Publication date
2009

Link to publication

Citation for published version (APA):
Wielemaker, J. (2009). *Logic programming for knowledge-intensive interactive applications*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:07 Jun 2026

Chapter 5

An architecture for making object-oriented systems available from Prolog

About this chapter This chapter has been published at the WLPE-02 (Wielemaker and Anjewierden 2002). It examines research question 3: "How to support graphical applications in Prolog?" by describing how XPCE, an object oriented graphics library defined in C can be connected to Prolog. XPCE is used by Triple20 (chapter 2, Wielemaker et al. 2005), the MIA tools (chapter 9, Schreiber et al. 2001; Wielemaker et al. 2003a) as well as most of the development tools of SWI-Prolog (Wielemaker 2003b).

Abstract It is next to impossible to develop real-life applications in just pure Prolog. With XPCE (Wielemaker and Anjewierden 1992) we realised a mechanism for integrating Prolog with an external object-oriented system that turns this OO system into a natural extension to Prolog. We describe the design and how it can be applied to other external OO systems.

5.1 Introduction

A wealth of functionality is available in object-oriented systems and libraries. This paper addresses the issue of how such libraries can be made available from Prolog, in particular libraries for creating user interfaces.

Almost any modern Prolog system can call routines in C and be called from C. Also, most systems provide ready-to-use libraries to handle network communication. These primitives are used to build bridges between Prolog and external libraries for (graphical) user-interfacing (GUIs), connecting to databases, embedding in (web-)servers, etc. Some, especially most GUI systems, are object-oriented (OO). The rest of this paper concentrates on GUIs, though the arguments apply to other systems too.

GUIs consist of a large set of entities such as windows and controls that define a large number of operations. Many of the operations involve destructive changes to the involved

entities. The behaviour of GUI components normally involves handling spontaneous input in the form of *events*. OO techniques are well suited to handle this complexity. A concrete GUI is generally realised by sub-classing base classes from the GUI system. In —for example— Java and C++, the same language is used for both the GUI and the application. This is achieved by either defining the GUI base classes in this language or by encapsulating foreign GUI classes in classes of the target language. This situation is ideal for application development because the user can develop and debug both the GUI and application in one language.

For Prolog, the situation is more complicated. Diversity of Prolog implementations and target platforms, combined with a relatively small market share of the Prolog language make it hard to realise the ideal situation sketched above. In addition, Prolog is not the most suitable language for implementing fast and space-efficient low-level operations required in a GUI.

The main issue addressed in this paper is how Prolog programmers can tap on the functionality provided by object-oriented libraries without having to know the details of such libraries. Work in this direction started in 1985 and progresses to today. Over these years our understanding of making Prolog an allround programming environment has matured from a basic interface between Prolog and object-oriented systems (section 5.3), through introducing object-oriented programming techniques in Prolog (section 5.4), and finally to make it possible to handle Prolog data transparently (section 5.5). These ideas have been implemented in XPCE. Throughout this paper we will use examples based on XPCE and discuss how these principles can be applied to other OO systems.

5.2 Approaches

We see several solutions for making OO systems available from Prolog. One is a rigid separation of the GUI, developed in an external GUI development environment, from the application. A narrow bridge links the external system to Prolog. Various styles of 'bridges' are used, some based on TCP/IP communication and others on local in-process communication. ECLiPSe (Shen et al. 2002) defines a generic interface for exchanging messages between ECLiPSe and external languages that exploits both in-process and remote communication. Amzi! (Merritt 1995) defines a C++ class derived from a Prolog-vendor defined C++/Prolog interface class that encapsulates the Prolog application in a set of C++ methods, after which the GUI can be written as a C++ application.

Using a narrow bridge between data processing in Prolog and a GUI in some other language provides modularity and full reuse of GUI development tools. Stream-based communication is limited by communication protocol bandwidth and latency. Whether or not using streams, the final application consists of two programs, one in Prolog and one in an external language between which a proper interface needs to be defined. For each new element in the application the programmer needs to extend the Prolog program as well as the GUI program and maintain the interface consistency.

For applications that require a wide interface between the application and GUI code we

would like to be able to write the application and GUI both in Prolog. Here we see two approaches. One is to write a Prolog layer around the (possibly extended) API of an existing GUI system (Kim 1993; SICS 1998). This option is unattractive as GUI systems contain a large number of primitives and data types, which makes development and maintenance of the interface costly. An alternative is to go for a minimal interface based on the OO message passing (control) principles. For this, we consider OO systems where methods can be called from the C-language[1] based on their *name*. The ability to access methods by name is available in many (OO) GUI toolkits as part of their *reflexion* capabilities. Fully compiled languages such as traditional C++ lack this facility, but the availability of many compiler extensions and libraries to add reflexion to C++ indicate that the limited form of reflexion that we demand is widely available. In the remainder of this article we concentrate on OO systems that can call methods by name.

5.3 Basic Prolog to OO System Interface

The simplest view on an OO system is that it provides a set of methods (functions) that can be applied to objects. This uniformity of representation (objects) and functionality (methods) makes it possible to define a small interface between Prolog and an OO system. All that is required to communicate is to represent an object by a unique identifier in Prolog, translate Prolog data to types (objects) of the OO system, invoke a method and translate the result back to Prolog. This model is illustrated in figure 5.1. We first describe how objects are manipulated from Prolog ('Activate'), then how we can represent Prolog as an object ('Call back') and finally we discuss portability and limitations of this model.

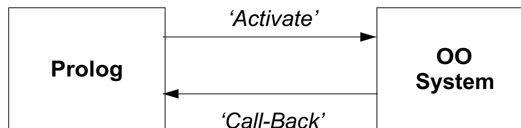

Figure 5.1: A simple view on Prolog to OO interface

Object Manipulation from Prolog We added four predicates to Prolog to manipulate objects in the OO system:

new(*?Reference, +Class(...Arg...)*)

Create an object as an instance of *Class* using the given arguments to initialise the object. Make the resulting instance known as *Reference* from Prolog. If *Arg* is atomic, convert it to a natural counterpart in XPCE (integer, float, atom). If it is compound,

[1] Or another language Prolog can communicate with. C is the de-facto interface language for many high-level languages.

create an instance using the functor as the class name and the arguments as initialising arguments. Unbound variables have no sensible counterpart in XPCE and therefore raise an exception. The example below creates a box (graphical rectangle) with specified width and height.

```
?- new(X, box(100,100)).

X = @459337
```

send(+*Reference, +Method(...Arg...)***)**
Given a reference to an object, invoke the named method with the given arguments. The arguments are translated as with `new/2`, supporting the functional notation shown in the code below. XPCE uses `send/2` for methods returning either no value or a boolean success/failure indication. The example creates a picture (graphics window) and displays a box at a given location.

```
?- new(P, picture),
   send(P, display(box(100,50), point(20,20))).
```

get(+*Reference, +Method(...Arg...), -Result***)**
Used for methods that return their result as an object or primitive data type. If the returned value is primitive, it is converted to a Prolog integer, float or atom. In all other cases an object reference is returned. The following code gets the left-edge of the visible part of a graphics window (*picture*). *Visible* is a instance of class *area* that describes the visual part and defines get-methods *x,y,w,h* to obtain the dimensions of the area object.

```
?- new(P, picture),
   get(P, visible, Visible),
   get(Visible, x, X).

P = @1072825
Visible = @957733
X = 0
```

free(+*Reference***)**
Destroy the referenced object.

Minimalists may claim that three of these four primitives are redundant. Generating an instance of a class is an operation of the class and can thus be performed by a method of the

class if classes can be manipulated as objects. In XPCE, `new/2` can be defined as `get`(*Class, instance(...Arg...), Reference*). Destroying an object is an operation on the object itself: `send`(*Reference, free*). Most object systems do not distinguish 'send' and 'get': methods that do not need to return a value return either boolean truth or the object to which the method was sent. Send and get are distinguished in the design of XPCE as well as in the interface because we perceive the distinction between *telling* an object to perform an action without interest for the result (e.g., *move* to a location) and *asking* an object to compute or create something, improves readability.

Object systems define rules that describe the lifetime of objects. These rules can be based on scoping, membership of a 'container' or garbage collection. This 'life-time' of the object is totally unrelated to the life-time of the Prolog reference and the user must be aware of the object life-time rules of the object system to avoid using references to deleted objects. XPCE defines scoping rules and provides a reference-based garbage collector. This takes care of the objects, but the programmer must be careful not to use Prolog object-references to access objects that have been deleted by XPCE. The SWI-Prolog Java interface JPL[2] represents Java objects as unique Prolog atoms and exploits a hook into the SWI-Prolog atom garbage collector to inform the interface about Java objects that are no longer referenced by Prolog. The XPCE route is lightweight but dangerous. The JPL route is safe, but atom garbage collection is a costly operation and Java objects often live much longer than needed.

Prolog as an Object The four predicates above suffice to invoke behaviour in the OO system from Prolog. Notable OO systems for GUI programming generally define *events* such as clicking, typing or resizing a window that require Prolog to take action. This can be solved by defining a class in the OO system that encapsulates Prolog and define a method *call* that takes a predicate name and a list of OO arguments as input. These arguments are translated to Prolog in the same way as the return value of `get/3` and the method is executed by calling the predicate.

In XPCE, class *prolog* has a single instance with a public reference (@prolog). GUI classes that generate events, such as a push button, can be associated with a *message* object. A message is a dormant method invocation that is activated on an event. For example, the creation arguments for a push button are the label and a message object that specifies an action when the user clicks the button. Together with the introduction of @prolog, we can now create a push button that writes `Hello World` in the Prolog console when clicked with the code below.

```
?- new(B, button(hello,
                 message(@prolog, call,
                         writeln, 'Hello World'))).
```

[2] `http://www.swi-prolog.org/packages/jpl/`

Portability The above has been defined and implemented around 1985 for the first XPCE/Prolog system. It can be realised for any OO system that provides runtime invocation of methods by name and the ability to query and construct primitive data types.

Limitations The basic OO interface is simple to implement and use. Unfortunately it also has some drawbacks as it does not take advantage of some fundamental aspects of object-oriented programming: specialisation through sub-classing and the ability to create abstractions in new classes. These drawbacks become apparent in the context of creating interactive graphical applications.

Normally, application GUI-objects are created by sub-classing a base class of the toolkit and refining methods such as *OnDraw* and *OnClick* to perform the appropriate application actions. Using the interface described above, this means we need to program the OO system, which implies The programmer has to work in Prolog and the OO language at the same time. In section 5.2 we discussed scenarios that required programming in two languages and resulted in two applications that communicated with a narrow bridge, providing good modularity. If we create a system where classes can be created transparently from Prolog we can achieve a much more productive development environment. This is especially true if the OO system does not allow for (re-)compilation in a running application, but Prolog can provide for this functionality. Unfortunately this environment no longer *forces* a clean modular separation between GUI and application, but it still *allows* for it.

5.4 Extending Object-Oriented Systems from Prolog

If we can extend the OO system from Prolog with new classes and methods that are executed in Prolog we have realised two major improvements. We gain access to functionality of the OO system that can only be realised by refining methods and, with everything running in Prolog, we can develop the entire application in Prolog and profit from the good interactive development facilities provided by Prolog. This can be realised in a portable manner using the following steps, as illustrated in figure 5.2:

- Defining a Prolog syntax for classes and methods, where the executable part of the method is expressed in Prolog.
- Create the OO system's classes from this specification and define the OO system's methods as wrappers that call the Prolog implementation.

A concrete example is given in figure 5.3, which creates a derived class *my_box* from the base class *box* (a rectangular graphical). The derived class redefines the method *event*, which is called from the window in which the box appears if the mouse-pointer hovers over the box. The method receives a single argument, an instance of class *event* that holds details of the GUI event such as type, coordinates and time. In this example we ensure the box is filled solid red if the mouse is inside the box and transparent (filled with nothing: @nil) otherwise,

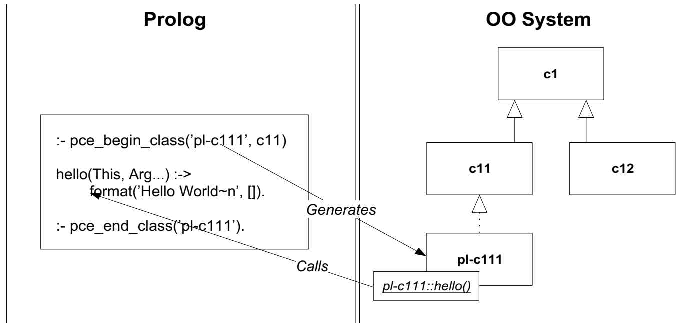

Figure 5.2: Creating sub-classes from Prolog

by redefining the method that specifies what happens if the mouse enters or leaves the box. In all other cases, the default behaviour of the super class is activated using `send_super/2`.

```
:- pce_begin_class(my_box, box).

event(Box, Event:event) :->
        (   send(Event, is_a(area_enter))
        ->  send(Box, fill_pattern(colour(red)))
        ;   send(Event, is_a(area_exit))
        ->  send(Box, fill_pattern(@nil))
        ;   send_super(Box, event(Event))
        ).

:- pce_end_class(my_box).
```

Figure 5.3: Defining a class from Prolog

XPCE is a 'soft typed' language. Method arguments may have type specifiers (`:event`) and if they do the system performs runtime checks on these. The method above requires the first argument to be an instance of the class *event*.

Implementation In XPCE, classes and methods are primary objects that can be manipulated using the API described in section 5.3. A method object consists of a name, argument type definition and a handle to the implementation. The implementation is either built into XPCE itself as a C-function, or it is provided by the XPCE/Prolog as an object (X) created by the interface. If an implementation defined by the interface is called, XPCE calls the interface

with the receiving object, type-checked arguments and provided implementation object X. The meaning of the object X is defined by the interface; XPCE just distinguishes built-in methods and methods defined by the interface.

We now describe how classes defined in Prolog are realised in XPCE. We do this in two steps. First, we describe a naive translation that creates XPCE classes and methods while the Prolog file is being compiled. Next, we describe how the actual implementation differs from this naive translation by introducing just-in-time creation of XPCE classes and methods.

The code from figure 5.3 is translated using Prolog's `term_expansion/2` based macro-facility. The `begin_class`(*my_box, box*) is expanded into a `new/2` call that creates class *my_class* as a subclass of *box*. The method definition is translated into a clause (figure 5.4) and API calls that create a method object with appropriate name, types and implementation object and associate this with the new class.

```
pce_principal:send_implementation('my_box->event', event(A), B) :-
    user:
    (   (   send(A, is_a(area_enter))
        ->  send(B, fill_pattern(colour(red)))
        ;   send(A, is_a(area_exit))
        ->  send(B, fill_pattern(@nil))
        ;   send_class(B, box, event(A))
        )
    ).
```

Figure 5.4: Clause that provides implementation for a method

Each send-method is translated into a single clause for the multifile predicate pce_principal:send_implementation/3, an example of which is shown in figure 5.4. The atom `'my_box->event'` is the unique handle to the implementation that is stored with the implementation object (X above) for the method. Prolog's first argument indexing ensures fast access, even if there are many methods.

Just-in-time creation of methods and classes Creating classes and methods while compiling the XPCE classes does not cooperate easily with Prolog's support for compiling files into object files or generating saved states. In addition, startup time is harmed by materialising all classes and methods immediately at load-time. Therefore, the actual implementation uses term expansion to compile the class declarations into the clause for pce_principal:send_implementation/3 as described and creates a number of Prolog facts that describe the classes and methods. XPCE provides a hook that is called if an undefined class or method is addressed. The Prolog implementation of this hook materialises classes and methods just-in-time. As the result of compiling an XPCE class is a set of Prolog clauses, all normal compilation and saved state infrastructure can be used without change.

Portability Our implementation is based on XPCE's ability to refine class *method*, such that the implementation can be handled by a new entity (the Prolog interface) based on a handle provided by this interface (the atom `'my_box->event'` in the example). Not all object systems have this ability, but we can achieve the same result with virtually any OO system by defining a small wrapper-method in the OO language that calls the Prolog interface. Ideally, we are able to create this method on demand through the OO system's interface. In the least attractive scenario we generate a source-file for all required wrapper classes and methods and compile the result using the target OO system's compiler. Even in this setting, we can still debug and reload the method *implementation* using the normal Prolog interactive debugging cycle, but we can only extend or modify the class and method *signatures* by running the class-compiler and restarting the application.

Experience The first version of XPCE where classes could be created from Prolog was developed around 1992. Over the years we have improved performance and the ability to generate compact and fast starting saved states. Initially the user community was reluctant, possibly due to lack of clear documentation and examples. The system proved valuable for us, making more high-level functionality available to the XPCE user using the uniform class-based framework.

We improved the usability of creating classes from Prolog by making the Prolog development environment aware of classes and methods (section 5.7). We united listing, setting spy-points, locating sources and cross-referencing. Support requests indicate that nowadays a large share of experienced XPCE users define classes. Problems are rarely related to the class definition system itself. User problems concentrate on how to find and combine classes and methods of the base system that fulfil their requirements. XPCE shares this problem with other large GUI libraries.

With the acceptance of XPCE/Prolog classes, a new style of application development emerged. In this style, classes became the dominant structuring factor for interactive applications. Persistent storage and destructive assignment make XPCE data representation a viable alternative to Prolog's poor support of these features based on assert/retract.

5.5 Transparent exchange of Prolog data

In the above scenario, control always passes through XPCE when calling a method, regardless of whether the method is built-in or implemented in Prolog. As long as the XPCE/Prolog classes only extend XPCE it is natural that data passed to the method is restricted to XPCE data. If XPCE classes are used for implementing more application-oriented code we need a way to process native Prolog data in XPCE. We distinguish two cases: *passing* Prolog terms as arguments and *storing* Prolog terms in XPCE instance variables. Below is a description of both problems, each of which is followed by a solution section, after which we present and an example in section 5.5.3.

- *Passing Prolog data to a method*

Seen from Prolog, XPCE consists of three predicates that pass arguments: `new/2`, `send/2` and `get/3`. New/2 calls the *constructor* method, called `initialise` in XPCE while `send/2` and `get/3` specify the method called. We consider the case where the method that is called is implemented in Prolog using the mechanism described in section 5.4 and the caller wishes to pass an arbitrary Prolog term to the method implementation. In this scenario the life-time of the term is limited to the execution of the method and therefore the term can be passed as a reference to the Prolog stacks.

- *Storing Prolog data in an* XPCE *instance variable*
  If methods can pass Prolog data, it also becomes useful to *store* Prolog data inside objects as the value of an instance variable. In this case the term needs to be stored on the Prolog permanent heap using a mechanism similar to `assert/1` or `recorda/3` and the instance variable needs to be filled with a handle to retrieve the stored Prolog term.

5.5.1 Passing Prolog data to a method

When a Prolog term is passed to a method, it is passed as a term handle as defined by the Prolog-to-C interface. The passed term needs not be ground and can be (further) instantiated by the implementation of the method called. Figure 5.5 provides an example, where a parse-tree is represented by a Prolog term of the form `node`(*Content, Children*) and is passed as a whole to initialise an instance of the Prolog-defined class *parse_tree*.

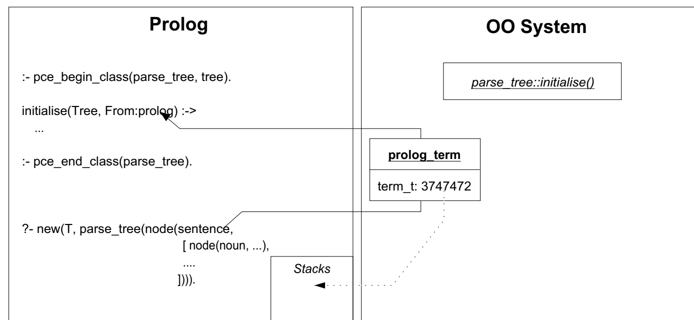

Figure 5.5: Passing Prolog terms across the OO system to other Prolog methods. The term is passed 'by reference'. It can contain unbound variables that are instantiated by the receiving method.

For the technical realisation we introduced class *host_data* providing XPCE with an opaque handle to data of the *host*, the term XPCE uses for the language(s) that are con-

nected to it. The XPCE/Prolog interface sub-classes this type to *prolog_term* (figure 5.5). Prolog data is opaque from XPCE's perspective.

When preparing an XPCE method invocation, we check whether a method argument is of type `prolog`, a type that refers to *prolog_term* as well as primitive data. If the argument is not primitive (integer, float or atom), the interface creates an instance of *prolog_term* and stores the `term_t` term reference in this object. Whenever an instance of *prolog_term* is passed from XPCE to Prolog, the interface extracts the term reference from the instance and passes it to Prolog.

5.5.2 Storing Prolog data in an XPCE instance variable

In the case where Prolog data is to be stored in instance variables, we cannot use the above described *prolog_term* with a reference to a Prolog term on the stack because the lifetime of the term needs to be linked to the object and existence of a term on the stack is not guaranteed after execution of a method completes. Instead, we need to use Prolog dynamic predicates or the recorded database and store a handle to the Prolog data in the XPCE instance variable. For the storage mechanism we opt for functions PL_record(), PL_recorded() and PL_erase() defined in the SWI-Prolog foreign interface. These functions copy terms between the stacks and permanent heap. A term on the permanent heap is represented by a handle of type `record_t`.

At the moment a method is called, the interface cannot know whether or not the term will be used to fill an instance variable and therefore a Prolog term is initially always wrapped into an instance of the XPCE class *prolog_term*. We identified two alternatives to create a persistent version of this term that is transparent to the user:

- In addtion *host_data*, Introduce another subclass of *host_data* that uses the PL_record()/PL_erase() mechanism to store the Prolog term. This can be made transparent to the Prolog user by rewriting instance variable declarations of type `prolog` to use this new class as type and define an automatic conversion between *prolog_term* and this new class.

- Introduce two internal representations for *prolog_term*: (1) the original based on `term_t` and (2) a new one based on `record_t`. Initially the object represents the Prolog term using the `term_t` reference. It is converted into the `record_t` representation by `new/2`, `send/2` or `get/3` if the *prolog_term* instance is *referenced* after completion of the method. We can detect that an object is referenced because XPCE has a reference-count-based garbage collector. Details are in the 3 steps described below.

    1. For each prolog-typed argument, create a *prolog_term* instance and keep an array of created instances.

    2. Run the method implementation.

3. Check the reference count for each created *prolog_term* instance. If zero, the instance can be destroyed. If > 0, it is referenced from some object and we transform the instance to its recorded database form. Figure 5.6 shows the situation after setting an instance variable.

We opted for the second alternative, mainly because it is easier to implement given the design of XPCE and its Prolog interface.

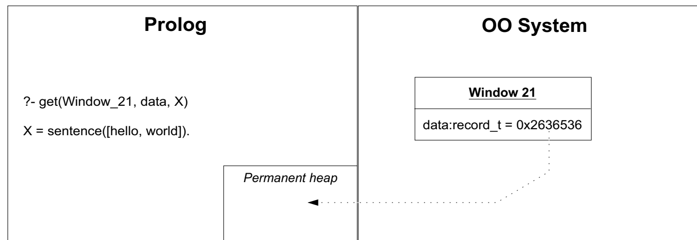

Figure 5.6: Storing Prolog data in external objects. The object contains a reference to a copy of the term maintained in the Prolog permanent heap.

5.5.3 An example: create a graphical from a Prolog tree

Figure 5.7 defines a class that creates a graphical tree where each node has a permanent payload of Prolog data associated. The hierarchy is created directly from a complex Prolog data structure. After creating a hierarchy this way, the Prolog tree as a whole is always passed *by reference* as illustrated in figure 5.5. Each node contains a permanent copy of the associated pay load as illustrated in figure 5.6.

5.5.4 Non-deterministic methods

If a Prolog-defined method is called from Prolog, it is desirable to be able to preserve possible non-determinism of the method. In the approach above, where activating a Prolog-defined method is always initiated from inside XPCE, this is not feasible. We can realise this desired feature by splitting method-invocation from Prolog into two steps. The first step resolves the methods, checks and possibly translates arguments as before. Now, instead of invoking the implementation, it returns a *continuation* (= goal). If the method is implemented in C, the implementation is called and the continuation is `true`. If the method is implemented in Prolog, the continuation is a call to the implementation for which we gave an example in figure 5.4. This implementation of `send/2` is illustrated by the code below and preserves both non-determinism and last-call optimisation.

```
:- pce_begin_class(my_node, node).

variable(data, prolog, both, "Associated data").

initialise(Node, Tree:prolog) :->
        Tree = node(Name, Data, Sons),
        send_super(Node, initialise(text(Name))),
        send(Node, data(Data)),
        forall(member(Son, Sons),
               send(Node, son(my_node(Son)))).

:- pce_end_class(my_node).
```

```
?- new(Tree, my_node(node(root, type(nounphrase),
                          [ node(dog, type(noun), []),
                            ...
                          ]))).
```

Figure 5.7: Class *my_node* creates a node and its children from a Prolog term, where each node carries a payload, also represented as a Prolog term. Bottom part shows an example invocation.

```
send(Object, Message) :-
        resolve_implementation(Object, Message, Continuation),
        call(Continuation).
```

We have not yet included this mechanism for preserving non-determinism because too much existing code relies on the implicit cut that now follows the implementation of each method. A possible solution is to add a declaration that specifies that the implementation is non-deterministic.

In this section we described passing Prolog data to methods implemented in Prolog, storing Prolog data in instance variables of the OO system and finally introduce non-determinism into methods defined in Prolog. We conclude with a brief analysis of portability and experience, where we omit non-determinism as this has not been implemented.

Portability Garbage collection in many object systems is not (only) based on reference counts and therefore the reference-count transparent change of status from term reference to a copy on the permanent heap is often not feasible. In that case one must opt for the first alternative as described in section 5.5.2. An interface as defined by PL_record()/PL_erase() is not commonly available in Prolog systems, but can be implemented easily in any system. Even without access to the source it is always possible to revert to an assert/retract-based implementation.

Experience The possibility to pass Prolog data around is used frequently, clearly simplifying and improving efficiency of methods that have to deal with application data that is already represented in Prolog, such as the parse tree generated by the SWI-Prolog SGML/XML parser (see section 7.2).

5.6 Performance evaluation

XPCE/Prolog message passing implies data conversion and foreign code invocation, slowing down execution. However, XPCE provides high-level (graphical) operations that limit the number of messages passed. Computation inside the Prolog application runs in native Prolog and is thus not harmed. For example, bottlenecks appear when manipulating bitmapped images at the pixel-level or when using Prolog-defined classes for fine-grained OO programming where good performance is a requirement.

Table 5.1 illustrates the performance on some typical method invocations through Prolog `send/2`. The first two rows call a C-defined built-in class involving no significant 'work'. We implemented class *bench* in Prolog, where the implementation of the method is a call to `true/0`. The first row represents the time to make a call to C and resolve the method implementation. The second adds checking of an integer argument while the last row adds the time to create and destroy the *prolog_term* instance. Finally, we add timing for Prolog calling Prolog and Prolog calling a C-defined built-in.

Accessing external functionality inside XPCE involves 5 times the overhead of adding C-defined predicates to Prolog. In return, we get object orientation, runtime type checking of arguments and optional arguments. Defining methods in Prolog doubles the overhead and is 10 times slower than direct Prolog-to-Prolog calls. Most XPCE methods realise significant functionality and the overhead is rarely a bottleneck.

5.7 Events and Debugging

An important advantage of the described interface is that all application-code is executed in Prolog and can therefore be debugged and developed using Prolog's native debugger and, with some restrictions described in section 5.4, Prolog's incremental compilation to update the environment while the application is running.

Goal	Class	Time (μS)
Prolog calling XPCE *a built-in method*		
`send(@426445, normalise)`	area	0.24
`send(@426445, x(1))`	area	0.31
Prolog calling XPCE *Prolog-defined method*		
`send(@426891, noarg)`	bench	0.54
`send(@426891, intarg(1))`	bench	0.69
`send(@426891, termarg(hello(world)))`	bench	0.65
Prolog calling Prolog		
`direct(@253535, hello(world))`	–	0.05
Prolog calling C		
`compound(hello(world))`	–	0.06

Table 5.1: Message passing performance, measured on an Intel X6800@2.93Ghz, SWI-Prolog 5.7.2 compiled with gcc 4.3 -O2

The Prolog debugger is faced with phenomena uncommon to the traditional Prolog world. The event-driven nature of GUI systems causes 'spontaneous' calls. Many user-interactions consist of a sequence of actions each causing their own events and Prolog call-backs. User interaction with the debugger may be difficult or impossible during such sequences. For example, call-backs resulting from dragging an object in the interface with the mouse cannot easily be debugged on the same console. The design also involves deeply nested control switches between foreign code and Prolog. The SWI-Prolog debugger is aware of the possibilities of interleaved control and provides hooks for presenting method-calls in a user-friendly fashion. Break-points in addition to the traditional spy-points make it easier to trap the debugger at interesting points during user-interaction. Figure 5.8 shows the source-level debugger in action on XPCE/Prolog code.

5.8 Related Work

To our best knowledge, there are no systems with a similar approach providing GUI to Prolog. Other approaches for accessing foreign GUI systems have been explored in section 5.2.

Started as a mechanism to provide a GUI, our approach has developed into a generic design to integrate Prolog seamlessly with an external OO programming language. The integrated system also functions as an object extension to Prolog and should therefore be compared to other approaches for representing objects in Prolog. Given the great diversity of such systems (Moura 2008), we consider this beyond the scope of this discussion.

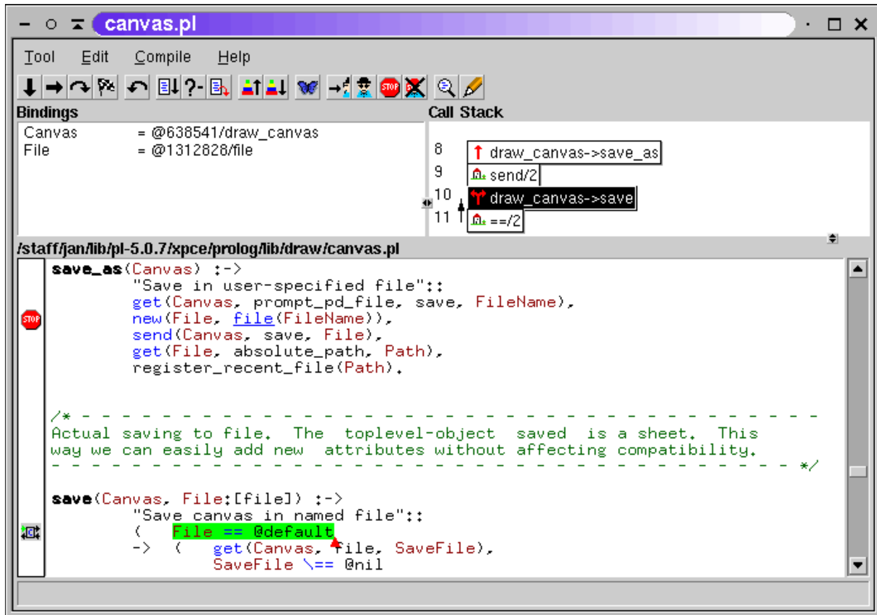

Figure 5.8: SWI-Prolog source-level debugger showing break-point and interleaved foreign/Prolog stack context.

5.9 Conclusions and discussion

We have presented an architecture for integrating an external Object Oriented system with minimal reflexive capabilities (i.e., calling a method by name) with Prolog. In most of the document we described the architecture of XPCE/SWI-Prolog and commented portability aspects when replacing XPCE with another OO system. The design allows for extending many existing OO systems naturally from Prolog. Using this interface the user can add new classes to the OO system entirely from Prolog which can be used to extend the OO system, but also for object-oriented programming in Prolog. Programming requires knowledge of the OO system's classes and methods, but requires *no* knowledge of the OO system's control primitives and syntax.

Using dynamically typed OO systems where classes and methods can be created at runtime through the interface without generating a source file, a quick and natural development cycle is achieved. If however, less of the OO system is accessible at runtime, development becomes more cumbersome because more changes to the code will require the user to restart the application.

Creating derived classes is often required to make effective use of an existing OO system, for example for refining behaviour of GUI controls. Our mechanism satisfies this requirement and allows for a tight integration between Prolog and the GUI classes. The ability to create derived classes from Prolog provides a uniform interface to the core of the OO system and extensions realised in Prolog libraries. Classes defined in Prolog form an appropriate organisation mechanism for the GUI part of applications.

The proposed solution can be less ideal for general purpose OO programming in Prolog for two reasons. First, the OO features of the OO platform dictate the OO features of our hybrid environment. For example, XPCE does not provide multiple inheritance nor polymorphism on argument types and therefore these features are lacking from XPCE/Prolog programs. Multiple inheritance can be attractive for data modelling. Second, the message passing overhead is relatively high. This is acceptable for course-grain organisation of applications and many user interface tasks, but it might be unacceptable for general purpose OO programming in Prolog. However, introducing an OO system for application programming next to interfacing to an external OO system as described in this paper is likely to confuse programmers. The discussion on supporting graphics from Prolog is continued in the overall conclusions of this thesis, section 11.3.1.

Acknowledgements

XPCE/SWI-Prolog is a Free Software project which, by nature, profits heavily from user feedback and participation. We would like to thank Mats Carlsson in particular for his contribution to designing the representation of XPCE methods in Prolog. We also acknowledge the reviewers for their extensive comments and suggestions, many of which have been used to clarify this paper.