A High-Level Generic Interface to External Programming Languages for ECLiPSe

Kish Shen Joachim Schimpf Stefano Novello
Josh Singer
IC-Parc, Imperial College,
London SW7 2AZ
United Kingdom

Published in *Practical Aspects of Declarative Languages, 4th International Symposium PADL 2002*, LNCS 2257, ©Springer-Verlag

Some typos in the code listings have been corrected in this version

Abstract

This paper addresses an important but rarely discussed practical aspect of programming in a declarative language: its interface to other programming languages. We present the high-level, generic interface of ECL^{i}PS^e, and discuss the reasons for our design choices. The main feature of the interface is that it cleanly separates the code for ECL^{i}PS^e from that of the external language, allowing the interface to be used for different languages. We believe that many of the concepts developed for this interface can be applied to other declarative languages, especially for other Prolog systems.

Keywords: Logic Programming, language interfaces, implementation, application development.

1 Introduction

An important practical aspect of a high-level programming language such as Prolog is its interface to other programming languages. In a large scale programming setting, it is unlikely that any one programming language will be used exclusively. In particular, the strengths of Prolog which make it ideal for expressing and solving complex problems also means it is not well suited for non-logical tasks such low-level programming and providing a graphical user interface (GUI) to applications. Thus, it is important to provide means to interface Prolog to other programming languages that are more suited to such tasks.

Many Prolog systems have interfaces to other programming languages, these generally allow Prolog code to invoke commands/procedures written in another programming language (which we will refer to as the *external language*), and/or for code in the external language to invoke Prolog predicates (if both directions are possible, the interface is said to be bi-directional). The interfaces can be broadly classified into two categories by the level of access to the internals of the Prolog system:

Low-level interfaces these provide the external language with direct access to the memory areas and low level representation of data structures of the Prolog system. The external language is thus tightly coupled to the Prolog system, and able to manipulate the Prolog data structures and machine state. In fact, some form of such an interface is normally used in the implementation of the Prolog system itself, as the interface between Prolog and the language it is implemented in, which is usually C. Because of the low-level nature of the interface, it is very system specific, and is usually restricted to interfacing to the one language.

Allowing the Prolog data structures to be directly manipulated can be powerful and efficient, but at the same time, it can be dangerous. It also requires the programmer to have a reasonable knowledge of the Prolog system.

High-level interfaces these provide the external language with a less direct access to the Prolog side. In particular, the external language is not able to directly access or manipulate the Prolog data structures.

For many purposes, such as providing a GUI, low-level access to the Prolog state is not required or even desirable. Furthermore, some programming languages (e.g. script-based languages such as Tcl/Tk or Perl) are unsuitable for manipulating the raw Prolog data structures in any case. In these cases, the data communicated between the two sides often have their own, separate, representations, and manipulating the data on one side does not directly affect the other side.

There are of course many different programming languages, and different ways they can be interfaced to Prolog. Many issues are involved in the design of such an interface, and various interfaces have been developed for various Prolog systems to various programming languages. However, there has been little or no discussion in the published literature about the reasons and issues behind the design of the interfaces. We feel that there are common issues that are worth discussing, and in this paper, we present our experience with developing a high-level interface for several external languages with ECL^{i}PS^e.

ECL^{i}PS^e is a constraint logic programming system being developed at IC-Parc. It is used at IC-Parc, and its spin-off company, Parc Technologies, as the core for developing industrial scheduling and planning applications. In the design of our external language interface, we needed to meet the commercial demands of Parc Technologies, and this strongly influenced some of our design decisions, which will be discussed in this paper.

ECL^{i}PS^e has both a low-level, bi-directional, interface to C, and a high-level, also bi-directional, interface to Java, Tcl/Tk and Visual Basic. In this paper, we concentrate on the high-level interface because it involves issues which apply to other Prolog/Logic Programming systems (and perhaps other declarative languages as well). This interface was first introduced in version 4.1 of ECL^{i}PS^e, released early in 1999, and evolved to its current form described here.

2 Motivation and Objective

Our main motivation for the development of a new external interface was to allow our ECL^{i}PS^e applications to be used within larger applications written in different languages. We also wanted to implement a robust development GUI for ECL^{i}PS^e, and provide a stable basis for a multitude of GUIs for various ECL^{i}PS^e applications in the future.

From experience with previous external language interfaces and requirements for our applications, we had several issues and objectives in mind while we were developing the interface:

Language generic We did not want an interface that was specific to a particular language. Developments in programming languages mean that a language in favour today may no longer be so in the future. Also, in commercial settings, there may be specific requirements to use a particular language, e.g. a client for one of our commercial applications explicitly required Visual Basic as the external interface language, while at the time we provided a Tcl/Tk interface.

We wanted an interface that is generic in the sense that the ECL^{i}PS^e side is independent of the languages used on the external side. Thus an agent written in a different programming language can be substituted on the external side without changing any code on the ECL^{i}PS^e side of the interface. In addition, while the syntax on the

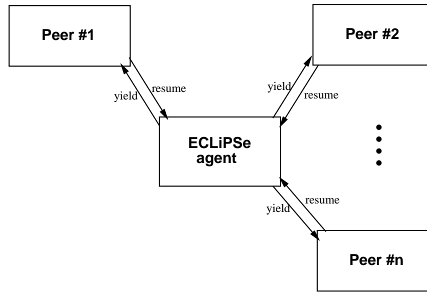

Figure 1: The Generic External Interface with Multiple Peers

external side would certainly differ from language to language, they should all implement the same concepts, so it should be easy to move from using the interface in one language to another.

Maintenance We wanted to minimise the development/maintenance overheads to support a new language via the interface. Thus, when interfacing to a new language is required, we can rapidly develop the interface required.

Control When ECL^{i}PS^e is interfaced to an external language, we often need to mesh the very different control regimes of the two languages: ECL^{i}PS^e is non-deterministic and single-threaded, and the external language tend to be deterministic and multi-threaded. We wanted to avoid complex control flow between the two languages, as this can easily lead to unmanageable nesting and interactions between the two languages.

Syntax We wanted to avoid any conflicts in syntax between ECL^{i}PS^e and the external language, which could happen if commands or data structures in the external language can occur in their native syntax within the ECL^{i}PS^e side and vice-versa. Allowing syntax of one language to appear in another is always error prone, especially for quoting of special characters (which are likely to be different in the languages), and when the command may be assembled dynamically during execution.

Uniform Usage We wanted to have a uniform interface, no matter whether ECL^{i}PS^e was used as a library (embedded into an external language host program) or as a server (in a separate process, possibly on a remote machine).

2.1 Conceptual Model

We developed a message-based, data-driven interface, as shown in Figure 1. The interface connects one agent written in ECL^{i}PS^e with one or more agents written in external languages.

The main conceptual points of the interface can be summarised by:

Multiple agents Multiple external agents can be connected to an ECL^{i}PS^e agent via the interface, and the connection itself may be by different methods. The interface provides the concept of *peers* to allow these external sides to be accessed in a uniform way. A peer is any external side (i.e. the external agent and its connecting peer queues).

Data-driven queues The two sides are connected by I/O queues with data-driven handlers, called *peer queues*. The interface provides operations to set up peer queues that connect the two sides. These are I/O queues that can send messages between the two sides in one direction. If the direction of data flow is from ECL^{i}PS^e side to the external side,

the queue is called a *from*-ECL^{i}PS^e queue; if the data flow is from the external side to ECL^{i}PS^e, it is called a *to*-ECL^{i}PS^e queue.

For each peer queue, a *handler* can be defined to handle the data transfer. These are procedures (or predicates in ECL^{i}PS^e) which are invoked to handle the transfer of data. The handler can either be a *data provider*, which supplies data to the other side when requested (when the other side reads from the queue and no data is available); or be a *data consumer*, which consumes data arriving on a queue. The execution of the handler for a queue is thus driven by the data transfer on that queue.

Structured messages To allow for platform and language independent interchange of typed data on the queues connecting the two sides, an ECL^{i}PS^e external data representation (EXDR) format was defined. EXDR allows the representation of a large subset of ECL^{i}PS^e's data types, including lists and nested structures. EXDR was inspired by Sun Microsystem's XDR format [7], however, unlike XDR, every EXDR term also includes its own type information.

Synchronous control flow Conceptually, there is a single thread of control flow between the external and ECL^{i}PS^e sides. At any time, one side has control, and only it can initiate the transfer of data on the queues (i.e. either sending data to the other side, or requesting data from the other side). On the ECL^{i}PS^e side, execution is suspended while the external side has control. Execution on the external side may or may not be suspended[1] while ECL^{i}PS^e side has control, depending on the programming language and/or the platform.

ECL^{i}PS^e remote predicate call (ERPC) The queue handlers already provide all the means for invoking actions in both directions. For convenience, and because an external side is always interfaced to an ECL^{i}PS^e agent, a form of Remote Procedure Call [1], which we call ECL^{i}PS^e Remote Predicate Call (ERPC) is always provided. It allows an external agent to conveniently invoke deterministic ECL^{i}PS^e predicates and retrieve their results.

3 Design Details

3.1 Peers

In the conceptual view of the interface, the way an external agent is connected to an ECL^{i}PS^e agent is not important, only that the two sides can communicate via data-driven peer queues. Thus, different concrete realisations of the interface are possible. We have provided two: an **embedded** variant, where the ECL^{i}PS^e agent and the external agent are in the same process (communicating through main memory); and a **remote** variant, where the ECL^{i}PS^e and remote agents are separate processes (connected by TCP/IP sockets). In the latter case, as the connections are sockets, the two agents can be located on different machines.

An ECL^{i}PS^e agent can be embedded into only one external agent. Multiple remote connections (perhaps to agents of different external languages) can be made, and any ECL^{i}PS^e agent can be connected, including one that is already embedded, i.e. an ECL^{i}PS^e agent can have at most one embedded peer, but multiple remote peers.

The differences between the embedded and remote interface variants are largely abstracted away by the unified conceptual view and the concept of peers. From the programmer's point of view, the remote and embedded variants can be largely used in the same way, and code written for one can be reused for the other. This is achieved by providing the same predicate/procedure names/methods interface calls with both variants on both the ECL^{i}PS^e and external sides.

[1] if execution is not suspended, then it cannot initiate data transfer to the ECL^{i}PS^e side.

The one main difference visible to the programmer is the process of initialising and terminating the connection with the two interfaces. With the embedding interface, the ECL^{i}PS^e agent is started from within the external agent (the agent loads ECL^{i}PS^e as a library), and the connection terminated by terminating the ECL^{i}PS^e agent. With the remote interface, the external agent has to be explicitly *attached* to the ECL^{i}PS^e agent for the connection, and detached for the termination. Attachment establishes an initial control connection between the two sides, along with some initial exchange of information. The control connection is used to co-ordinate and synchronise subsequent actions.

As an example of the use of both variants, the TkECL^{i}PS^e development tools (a set of development tools including debugger and state browsers), which were originally written to be used in an embedded setting together with the TkECL^{i}PS^e toplevel GUI, runs in the remote setting with very few modifications, even though the bulk of the code was developed before the conception of the remote interface. In terms of code sizes, there is about 4260 lines of Tcl and 1750 lines of ECL^{i}PS^e code that are shared. The specific code for starting the tools with the remote interface is about 200 lines of Tcl code and 60 lines of ECL^{i}PS^e code. About 30 lines of Tcl code and no ECL^{i}PS^e code would be needed to start the development tools with the embedding interface.

3.2 Peer Queues

The peer queues are implemented differently in the embedded and remote setting. In the embedded case the queues are shared memory buffers, while in the remote case the queues are implemented with sockets connecting the two sides.

To a user, a peer (once it is set up), whether remote or embedded, can be treated in the same way. Information is transferred between the two sides via the peer queues, which are created in a uniform way. The creation can be initiated from either the ECL^{i}PS^e or the peer side. From the ECL^{i}PS^e side, the difference between a remote peer queue and an embedded peer queue is hidden by providing the same predicate to create the queue – the user just specifies which peer the queue is for, and then the appropriate queue is created.

3.3 Typed EXDR Messages

The EXDR encoding is instrumental in providing language and architecture independence for the interface. Similar to XML [11], and unlike XDR, EXDR data includes type information. This is implemented in a very compact way by tagging each data item with a byte that identifies the particular data type. This allows the type of the data sent to be dynamically determined when the data is sent, rather than being statically fixed, and is particularly useful for a dynamically typed language like Prolog.

The data types that are available in this format are listed in Figure 2. The idea is to represent that subset of ECL^{i}PS^e types which has a meaningful mapping to many other languages' data types. Apart from the basic types, lists and nested structures are available and are mapped to meaningful types in many external languages. The main restriction is that logical variables (which have no equivalent in most other languages) are not allowed in their general form. However, singleton occurrences are allowed and useful to serve as place-holders, e.g. for result arguments in ERPCs (see section 3.7)

A small difficulty arises with a language like Tcl whose type system is too weak to distinguish between all the EXDR types: different EXDR types map to the same string in Tcl. While this is usually no problem when Tcl receives data, we have augmented the Tcl send primitive to take an additional argument which specifies the EXDR type into which a given string should be encoded.

The complete specification including the concrete physical encoding for the EXDR format is given in [6] and appendex A. As part of the external side of the interface for a particular external language, the mapping of EXDR data types into that language must be defined.

EXDR type	ECLiPSe type	Tcl type	VB type	Java type
Integer (32bit)	integer	int	Long	java.lang.Integer
Long (64bit)	integer	string	n/a	java.lang.Long
Double	float	double	Double	java.lang.Double
String	string	string	String	java.lang.String
List	./2	list	Collection of Variant	java.util.Collection
Nil	[] /0	string ""	Collection of Variant	java.util.Collection
Struct	compound	list	Array of Variant	CompoundTerm
Placeholder	anon variable	string "_"	Empty Variant	null

Figure 2: EXDR types with some language mappings

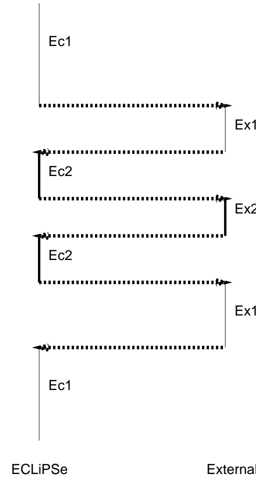

Figure 3: Nesting of Handlers

3.4 Control Flow

The control flow between the two sides is based on the synchronous yield/resume model: Control is transferred from ECL^{i}PS^e to the external side when ECL^{i}PS^e *yields* to the external side. Control is transferred from the external side back to ECL^{i}PS^e by *resuming* ECL^{i}PS^e. For example, when data is transferred from ECL^{i}PS^e to the external side on a peer queue, ECL^{i}PS^e will yield to the external side to allow the external side to process the data (via a handler for the queue). When the external side completes the processing, control is returned to the ECL^{i}PS^e by resuming ECL^{i}PS^e.

Note that handler execution on the two sides can be nested. An example of this is shown in Figure 3. In the figure, time advances down the page, and the figure shows the transfer of control between the ECL^{i}PS^e and remote sides. A vertical line shows that a particular side has control, and the horizontal arrows shows the transfer of control. Initially, the ECL^{i}PS^e handler Ec1 is executing, and ECL^{i}PS^e has control. At some point, control is transferred to the external language, and the external handler Ex1 is invoked. This transfers control back to ECL^{i}PS^e, starting a new handler Ec2 (the line is thicker to more readily distinguish it from Ec1), which in turn invokes an external handler Ex2. When Ex2 completes, control is returned to ECL^{i}PS^e and the execution of Ec2 continues until it also completes and returns to the remote side, where Ex1 continues and completes, returning control to ECL^{i}PS^e, which continues the execution of Ec1. Thus, Ex2 is nested within the execution of Ec2, which is nested within Ex1, which is nested within Ec1. This nesting allows the implementation of the equivalent of the 'call back' functionality of traditional RPCs.

The thread-based control flow limits the complexity of interactions between the two sides. As ECL^{i}PS^e has only a single thread of execution, it would not be sensible to allow

the external side to request execution of other goals while ECL^{i}PS^e side has control and is executing a goal. Note that the external side is not limited to being single threaded (and in fact neither Java or Tcl are single threaded).

The topology for transferring control with multiple peers is always a star shape with the ECL^{i}PS^e agent in the middle: control is handed over from the ECL^{i}PS^e agent to a peer, which can then only hand back control to that ECL^{i}PS^e agent. This is shown in Figure 1.

3.5 Generic invocation of action

We achieve language independence in our interface by using the generic concept of queues over which messages in the language independent EXDR format are transferred. We did not provide any built-in method to directly execute commands of the external language from within ECL^{i}PS^e. Nevertheless, ECL^{i}PS^e can cause actions to take place on the external side. The idea is that instead of making a procedure call directly, data transfers on a queue are used to invoke the handler for the queue. The data transferred specifies how the procedure should be invoked. The ECL^{i}PS^e side can thus regard the remote side as a black box, where the programming of a particular queue just involves specifying the protocol for transferring data and what the data is for. None of the details of *how* the data would be processed need to be known on the ECL^{i}PS^e side – all the code for doing this remains on the remote side. In particular, a different language can be substituted on the remote side, and as long as the handler for the queue obeys the same protocol, nothing on the ECL^{i}PS^e side is affected by the change.

3.6 Generic interface within the external language

The key abstractions of peer and of a peer queue allow natural counterpart abstractions on the external side, which can give a highly flexible underlying architecture. Abstraction is a strong element of object-oriented language such as Java, and our Java side of the interface demonstrates this. Just as a peer is a generic interface to an external side of any kind, the *EclipseConnection* interface in our Java code is implemented by different classes providing a connection to an ECL^{i}PS^e engine. For example *EmbeddedEclipse* implements *EclipseConnection* in the embedded variant of the interface; and *RemoteEclipse* class on the other hand implements *EclipseConnection* in the remote variant. Just as a peer queue allows communication between ECL^{i}PS^e and any kind of peer, so the Java classes *FromEclipseQueue* and *ToEclipseQueue* are provided by any class implementing *EclipseConnection*.

3.7 ERPC

The ERPC mechanism is provided for ease of programming. It is implemented on top of a pair of peer queues (to- and from-ECL^{i}PS^e), with the handler for the to-ECL^{i}PS^e queue reading the goal (in EXDR format), executing it, and returning the resulting goal to the external side. The handler essentially looks like:

```
erpc_handler :-
   read_exdr(rpc_in, Goal), once(Goal), write_exdr(rpc_out, Goal).
```

The actual ERPC handler code deals with failures and exceptions as well. These queues and the handler are pre-defined by the ECL^{i}PS^e-side of the interface, along with the handler.

Since we are interfacing to a variety of different external languages, none of which have concepts of logical variables, backtracking or goal suspension, the kind of ECL^{i}PS^e goals that can be called through the interface must be restricted. The abstraction of an ECL^{i}PS^e goal which the interface provides to the external language is that of a procedure with input arguments and output arguments which expect/return data of certain EXDR types.

On the ECL^{i}PS^e side that means that (i) externally callable goals are limited to return only one solution by committing to the first one, (ii) all variables in input arguments will

be singleton variables and can only be used to return results, (iii) results cannot contain shared variables. It is in the responsibility of the ECL^{i}PS^e programmer to provide callable predicates that observe these restrictions. In practice that means that e.g. difference lists need to be transformed into standard lists, and multiple solutions can be either collected and returned in a list, or alternatively returned incrementally via a dedicated, application-specific peer queue (as demonstrated in the map coloring example of section 5).

Since the external languages have different, incompatible argument passing conventions, especially for output arguments, we decided on an ERPC protocol that is at least natural and easy to implement on the ECL^{i}PS^e side: the goal is sent as a compound term, which may contain one or more singleton variables as placeholders for the output arguments. To return the result after successful execution, we send back the complete original goal, but with the former variables replaced by result values. This method avoids the need for a complex return-result protocol involving variable-handles or identifiers. It does however requires the external side to extract the results from the goal term. This protocol is suitable for most rpcs except those that have very large input arguments. In this latter case, the programmer would set up a queue for sending the input separately.

4 Discussion

4.1 Separation of ECL^{i}PS^e and external code

The interface clearly separates the code for ECL^{i}PS^e and the code for the external language. This means that the ECL^{i}PS^e code and the external code can be developed separately with just the interchanges between the two languages clearly specified. In particular, it means that

- Any problems with incompatibilities between the syntax of ECL^{i}PS^e and the remote language is avoided.
- The same ECL^{i}PS^e side of the interface can be used for different languages. There is no need to learn to use a different interface if the external language is changed.
- The development of the interface for a new external language means only a new external side of the interface has to be developed.
- The programmers on one side need not have much knowledge about the other programming language. In the case of Parc Technologies, this means that GUI and Java programmers can be hired without requiring them to either already know or undergo extensive training in ECL^{i}PS^e or Prolog.
- The converse is also true: ECL^{i}PS^e and CLP programmers do not need expertise or knowledge in GUI or Java programming.
- Each language is left to do the tasks they are most suited for. We do not need to 'enhance' ECL^{i}PS^e to provide features for tasks it is not suited for. For example, to properly support GUIs, a language needs to support some notion of multi-threading or an event-loop to cope with the inherently reactive nature of the task. As none of this management of the GUI is done on the ECL^{i}PS^e side, there is no need to introduce such features to ECL^{i}PS^e.
- Where the external language is used for providing a GUI, then the core part of the ECL^{i}PS^e code can often be easily detached from the interface and used separately. This is particularly convenient for both development and unit-testing of the ECL^{i}PS^e code.

4.2 Supporting a New External Language

The ECL^{i}PS^e side and the external side are loosely coupled, and have few dependencies on the low-level workings of either ECL^{i}PS^e or the external language.

To implement the embedded variant of the interface, the external language system must be able to load ECL^{i}PS^e as a library and to invoke a subset of the functions provided by ECL^{i}PS^e's C/C++ interface. These are the functions needed to initialise and finalise the ECL^{i}PS^e engine and to access the memory queue buffers.

For the remote variant of the interface, the remote side of the interface protocol has to be implemented, i.e. sending the appropriate message at the appropriate time, and performing the right actions on receiving messages from the ECL^{i}PS^e side. The protocol is specified in the ECL^{i}PS^e Embedding and Interfacing manual. This code should be straightforward to write and basically requires that sockets can be programmed in the language.

Both interfaces also need to provide support for the EXDR format, i.e. encoding/decoding native data into/from EXDR format. Depending on the external language, this may need to be supported at the C level (for example, in Tcl this is done in C, in Java this is done in Java).

Our experience so far is positive: the interface was initially developed mainly for use with Tcl/Tk for the development of the GUI for ECL^{i}PS^e itself. Since then, Parc Technologies have decided to standardise on using Java for all their GUI (and any other non-ECL^{i}PS^e) development, and support through this interface for Java was rapidly developed, both in the embedded and remote variants.

We could also confirm the reusability of the ECL^{i}PS^e side code with different external languages: parts of the ECL^{i}PS^e development GUI that was written in Tcl originally have been successfully replicated in Visual Basic or Java. However, with the introduction of the peer concept, this replication is now rarely necessary, as GUI components can be written in different languages.

4.3 Synchronous Control Flow

In our interface, the interaction between the external and ECL^{i}PS^e sides is synchronous. We deliberately avoided the complexity of a general message passing system, which would be difficult to combine with the already complex control flow in a constraint programming system.

As control is transferred for each exchange of data, and when the external side has control, ECL^{i}PS^e execution is suspended, there might be a problem with efficiency. However, as discussed in section 2.1, the execution of the external agent is not necessarily suspended while ECL^{i}PS^e side has control. In a multi-threaded external language like Java, the data can be read by the Java side and control returned to ECL^{i}PS^e side quickly while the Java side then processes the data concurrently.

With our current main area of application, the provision of GUIs, efficiency does not seem to be a problem. Asynchronous communications can be programmed separately, using standard sockets, if necessary.

4.4 Scope of Applicability

The high-level interface is a general interface to an external language, and is of course not limited to allowing the external language to providing GUIs for ECL^{i}PS^e applications.

The different strengths of the embedding and remote variants of the interface makes them suitable for different uses. Some of the issues to consider are efficiency, flexibility, security and fault tolerance.

4.4.1 Efficiency

The memory buffers of the embedded interface offer faster communications between the two sides than the socket connections of the remote variant. With the remote interface, data sent from one side needs to be physically transmitted (via sockets) to the other side, perhaps with buffering on both sides. TCP/IP also imposes an overhead on each transmission of data, such that it takes tens of milliseconds per transmission, regardless of the size of data transmitted, even when the two sides are on the same machine. This means that for applications where there are frequent exchanges of data, the process can be noticeably slowed by the interface. In some situations, it might be possible to reduce the number of times the control is transferred by pooling the updates and sending them in batches.

4.4.2 Flexibility

The flexibility of connecting to multiple external agents via the remote interface is quite useful. For example, Parc Technologies decided to standardise on using Java for all its non-ECL^{i}PS^e coding, while the ECL^{i}PS^e graphical development tools are written in Tcl/Tk. With the initial embedding interface, these tools were only available to programs which also used Tcl/Tk for their GUI, but the remote interface allows these tools to be used in conjunction with a Java agent, making the process of development much easier. The multiple agents approach will also allow new tools to be developed in Java, without needing to recode all the existing development tools into Java.

The remote interface allows an ECL^{i}PS^e agent to be run remotely, on any machine that can be reached via the internet. One use for this is to allow an ECL^{i}PS^e program to be debugged remotely.

A practical advantage for the remote interface that we did not initially foresee is that the memory and other resources are not shared between the ECL^{i}PS^e and external agents. In the embedding interface, our experience with programming large applications in Java, ECL^{i}PS^e and also other software systems such as an external Mixed Integer Programming solver, all doing their own memory management and all interacting as a single process, non-repeatable problems (perhaps due to some memory leak) do occur that are difficult to track down. In the remote interface, bugs caused by the interaction of different memory management are less likely to occur, and any bugs which do occur are easier to track down.

4.4.3 Security

Potentially, because the remote interface allows connections from anywhere reachable on the network, the remote side can be 'hijacked' by an imposter, and once attached, it has full access to the ECL^{i}PS^e side through ERPC, and hence to the ECL^{i}PS^e side's file space. The remote protocol implements a 'pass-term' check, where an ECL^{i}PS^e term is transmitted from the remote side to the ECL^{i}PS^e side and checked before the socket connection for the ERPC is allowed. Another method to limit access is to allow attachment on the local machine only via the loopback address. Further security can be imposed by the programmer, e.g. encryption of the data transmitted on the queues.

4.4.4 Fault-tolerance

With the remote variant, there is the possibility that the connection between the two sides may be lost unexpectedly, either because of some network problems, or because one side dies unexpectedly. In such cases, the peer queues will be disconnected, and when one side detects this, a unilateral detachment will be performed by the remote protocol, and control is returned to the programmer to deal with this unexpected situation. In the embedded variant, the two sides are in the same process, so the problem does not arise.

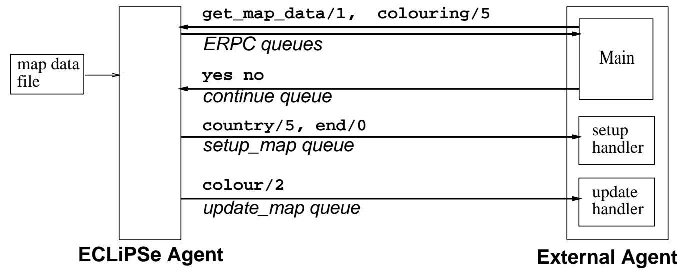

Figure 4: Structure of Map Colouring Program

5 An Example – Map Colouring

In the ECL^{i}PS^e distribution, there is an example illustrating the use of the interface. Currently Tcl/Tk is used as the external language, providing a GUI for the main ECL^{i}PS^e code, which solves the standard map colouring problem where a map of countries should be coloured with four colours such that no neighbouring countries share the same colour.

The overall structure of the program is shown in Figure 4. In brief, the ECL^{i}PS^e agent can colour a map by several different methods, using the map data specified in a map data file. The external agent provides the GUI for the user to select the map, and how many countries from the map, should be coloured; method of colouring the map, and also for displaying the map as it is being coloured by the ECL^{i}PS^e agent. Finally, when the map is successfully coloured, the external agent allows the user to ask for more solutions.

The two sides communicate via ERPCs, and three peer queues:

setup_map : this from-ECL^{i}PS^e queue transmits the shape and position information of a map to the external side, which uses the information to construct the map.

update_map : this from-ECL^{i}PS^e queue transmits the information for updating the map as it is being coloured by the ECL^{i}PS^e program.

continue : this to-ECL^{i}PS^e queue transmits the request for further solutions once the map is coloured.

The ECL^{i}PS^e code consists of two main components: the setting up of the map for the external side in `get_map_data/1`, and the colouring of the map in `colouring/5`. The abstract outline of the code relevant to the interface is as follows:

```
get_map_data(Size) :-
       ....
       write_exdr(setup_map, country(C,X1,Y1,X2,Y2)),
       ....
       write_exdr(setup_map, end)

colouring(Type, Select, Choice, Size, Time) :-
       ....
       ( write_exdr(update_map, colour(C, Colour))
       ; write_exdr(update_map, colour(C, gray)), fail
       ),
       ....
       read_exdr(continue, Continue),
       Continue == no.
```

`get_map_data/1` sends the data for the map to be coloured to the external side. The full map has been read into the ECL^{i}PS^e side earlier by another predicate (not shown here).

`Size` specifies how many countries from the full map are to be coloured: configuration information on `Size` countries are sent to the external side. This information is sent in a loop, consisting of a series of `country/5` terms, terminating in a `end` term.
`colouring/5` does the actual colouring of the map. The first four arguments specify various options, and the last argument `Time` is an output argument for returning the cpu time consumed for colouring the map. When a country `C` is set to a particular colour `Colour` during the colouring process, this information is sent to the external side via the update_map queue; a choice-point is created so that the colour can be reset (to gray) when it is backtracked over. Finally, when the map is successfully coloured, the program reads from the `continue` peer queue. This hands control over to the external side, where the user can specify via the GUI if the program should continue and return another solution or not. By clicking on the appropriate widget, either yes or no is sent via the `continue` queue to the ECL^{i}PS^e side, and `read_exdr/2` on the ECL^{i}PS^e side returns. The execution then either backtracks to get the next solution or finishes.

Both predicates are called from the external side via ERPC calls, with the ERPC invoked when the user clicks on the appropriate widget in the GUI.

This example shows the generic nature of the interface concretely: the ECL^{i}PS^e side of the code does not depend on the external side being Tcl. Another external language can be used to provide the GUI, as long as the implementation follows the protocol defined above. The actual example program in the distribution can also be run either embedded or remotely.
For illustration, we outline the Tcl code for handling the map colouring:

```
proc run {} {
     ....
     ;# calling colouring/5, followed by the type information
     ec_rpc [list colouring $solver $select $choice $mapsize _] (()()()I_)
     ....
}

proc update_map {...} {
     ....
     ;# read the colour/2 term
     set info [ec_read_exdr update_map]
     ;# extract the country and colour from the data and display it
     set country [lindex $info 1]
     set colour  [lindex $info 2]
     ....
}

proc continue_colouring {continue_queue} {
     global continue_state
     ....
     ;# wait for user to decide if more solution is wanted
     tkwait variable continue_state
     ....
     ;# send decision to ECLiPSe
     ec_write_exdr $continue_queue $continue_state ()
     ....
}
```

`run` is a procedure invoked by pressing a 'Run' button which starts the colouring process. This procedure makes an ERPC call. As discussed previously, for a weakly typed language like Tcl, type information has to be specified (the `(()()()I_)` string, see the manual [6] for more details).

While `colouring/5` is running, it sends the colour information as described above. On the Tcl side, this invokes the handler `update_map`, which reads the information from the update_map queue and displays it.

`continue_colouring` is the handler procedure for the `continue` queue. When ECL^{i}PS^e reads from the queue, this procedure is invoked on the Tcl side. The Tcl code waits for the continue_state variable to be set by the appropriate widgets (buttons that the user clicks to specify if another solution is wanted). This variable is set to either `yes` or `no`, and the information is returned to the ECL^{i}PS^e side.

6 Related Work

Many existing Prolog systems provide some form of external language interface. Most of the earliest are 'low-level' interfaces to C. Interfaces to Tcl/Tk and, more recently, to Java and Visual Basic have been developed. For script languages such as Tcl/Tk, the interface is usually high-level, because Tcl itself cannot represent or manipulate the raw representation of Prolog data structures. On the other hand, Java interfaces can be either high-level or low-level.

Examples of high-level interfaces include the old ProTcXl [5] of ECL^{i}PS^e; the Tcl/Tk and Visual Basic interfaces of SICStus [8]; the Tcl/Tk and Java interfaces of Ciao [2]; the Tcl/Tk interface of BinProlog and ProLog by BIM [9].

Most of these interfaces are not generic; for example, the Visual Basic, Tcl/Tk and Java interfaces of SICStus are very different from each other, and so are the Tcl/Tk and Java interfaces of Ciao. In fact, Java interfaces can be low-level like C interfaces, allowing the Java program to directly access the Prolog data. An example of this is Jasper, the Java interface of SICStus. Foreign language interfaces tend to be complex (this can be seen by simply looking at the amount of documentation that the manuals need to dedicate to their description). We hope that having a generic interface will significantly reduce the learning curve for the user.

The design of our interface was motivated partly by our experience with ProTcXl, an earlier interface for ECL^{i}PS^e to Tcl/Tk, which we abandoned in favour of starting afresh with the generic interface. We designed the new interface to overcome some of the problems of ProTcXl: it was Tcl specific, had a complex control scheme that inexperienced programmers often got wrong, allowed Tcl commands in Tcl-syntax to be assembled and called within the ECL^{i}PS^e code, which often lead to incorrect parsing by the Tcl interpreter. In contrast, the new interface is not Tcl specific, has a much simpler control flow, and does not provide for executing Tcl commands directly within the ECL^{i}PS^e code. In addition, there are less low-level 'glue' code, so maintenance and portability should be easier.

Our interface avoided the problem of syntax conflicts between the external language and Prolog by avoiding the specification of external procedures from within Prolog code. Other ways of avoiding this problem are:

- Wrapping the components of a command with 'type-wrappers' so that they would not be mis-identified. An example of this is Ciao's Tcl/Tk interface.
- Specifying the external command in Prolog syntax, and perform on-the-fly translation into the external language. This was the approach taken with BinProlog's Tcl/Tk interface. It offers the possibility that the command may be executed in a different external language with a different translator. However, this approach may have some problems with statically and strongly typed languages.

The Ciao Java interface has some interesting similarities to ours. The two sides are also connected via sockets, and a serialised representation of Prolog terms and Java object references is used to transport data between the two sides. Actions on both sides can be

invoked via event handlers. One main difference from our interface is that the interaction between the Prolog and Java sides appear more complex than in our scheme, and requires the Prolog side to be multi-threaded. With our more simple control scheme, we do not need threads.

An alternative to providing external interfaces directly might be to use a 'middle-ware' layer like Corba [10], which will allow RPC calls, but at the price of an additional software layer, and an unnatural match of the object oriented aspects of Corba and ECL^{i}PS^e (which currently does not have an interface to Corba). The main difference between an interface specified in Corba IDL versus one in ECL^{i}PS^e EXDR would be that the IDL typing is more rigid and does not offer such a natural match with ECL^{i}PS^e/Prolog data types. Another concern is that Corba is mainly designed for network interoperability, and having unified embedded and remote versions would require the definition of a suitable subset.

Of course, there is nothing inherent in our interface that would limit it to a Logic Programming language. It should be equally applicable to Functional Programming languages. We are not aware of any direct equivalent in Functional Programming languages: although foreign language interfaces also exist for Functional Programming languages, many such interfaces seem to be targeted to C. HaskellDirect [3] allows Haskell to be interfaced to an external language, generating the necessary code to make function calls (and be called from an external language) by specifying the 'signatures' for functions in an Interface Definition Language, which is then compiled by HaskellDirect. Although it can be used to interface to different languages, it seems to be mainly targeted for C. Unlike our interface, function calls are to be made directly in the language, instead of just passing the data. An example of a non-C foreign language interface is sml_tk [4], which is an interface to Tcl/Tk for Standard ML. This interface is quite tightly coupled to Tcl/Tk, and probably cannot be used to interface to another language.

7 Conclusion

We have presented the ECL^{i}PS^e high level external language interface. Since its initial development two years ago, this interface has been used extensively by us. The Java interface is being used for all the commercial applications that Parc Technologies is developing. The interface has also been used for a development GUI toplevel for ECL^{i}PS^e, and a set of development tools that can be accessed from the toplevel and other ECL^{i}PS^e applications that use the embedded Tcl/Tk interface, and through the remote interface, the development tools can be used with any ECL^{i}PS^e process. The Tcl/Tk interface was also used in an application that IC-Parc was developing for a customer.

We believe that the interface offers advantages over previous external language interfaces in that it cleanly separates the ECL^{i}PS^e (Prolog) and the external code. This allows the interface to be generic and eases our development and maintenance efforts.

Acknowledgements

The authors gratefully acknowledge the invaluable help that our colleagues at IC-Parc and Parc Technologies for their feedback and discussions on the development of the interface. We also thank Mark Wallace for his quick feedback and comments on this paper. We also thank the referees for their comments.

A The EXDR Format Specification

```
ExdrTerm ::= 'V' Version Term
Term     ::= (Integer|Double|String|List
              |Nil|Struct|Variable)
Integer  ::= 'I' XDR_int | 'J' XDR_long
Double   ::= 'D' XDR_double
String   ::= 'S' Length <byte>*
List     ::= '[' Term (List|Nil)
Nil      ::= ']'
Struct     ::= 'F' Arity String Term*
Variable   ::= '_'
Length     ::= XDR_int
Arity      ::= XDR_int
Version    ::= <byte>
XDR_int    ::= <4 bytes, msb first>
XDR_long   ::= <8 bytes, msb first>
XDR_double ::= <ieee double, exponent first>
```

References

[1] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. *The Ciao Prolog System Reference Manual*, 2000.

[3] S. Finne, D. Leijen, and E. Meijer. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of the International Conference on Functional Programming*. ACM Press, 1999.

[4] C. Lüth and B. Wolff. *sml_tk: Functional Programming for Graphical User Interfaces, Release 3.0*.

[5] M. Meier. *ProTcXl 2.1 User Manual*, 1996.

[6] S. Novello, J. Schimpf, J. Singer, and K. Shen. *ECLiPSe Embedding and Interfacing Manual, Release 5.2*, 2001.

[7] R. Srinivasan. *XDR: External Data Representation Standard*. Request for Comments (RFCs) 1832. The RFC Editor, Sun Microsystems, Inc., 1995.

[8] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 1995.

[9] P. Tarau and B. Demoen. Language Embedding by Dual Compilation and State Mirroring. In *Proceedings of the 6-th Workshop on Logic Programming Environments, ICLP94*, 1994.

[10] S. Vinoski. CORBA: Intergrating Diverse Applications Within Distributed Hetrogeneous Environments. *IEEE Communications*, Feb. 1997.

[11] W3C. *Extensible Markup Language (XML) 1.0 (Second Edition)*, 2000. available at url: http://www.w3.org/YR/2000/REC-xml-20001006.