Accessing External Applications from Knowledge Based Systems

Yannis Cosmadopoulos and Richard Southwick
Logic Programming Group, Imperial College, London.

Kostas Stathis
Knowledge-Based Systems Group, Numerical Algorithms Group Ltd., Oxford.

Abstract

This paper addresses the problem of interfacing knowledge-based systems (KBS) to external software packages. We describe a methodology for building tightly-coupled interfaces between logic-based KBS front-ends, and back-end application packages. These methods have been implemented in the form of a Back End Manager (BEM), a KBS component that is designed to handle the flow of information between problem solvers and back-end applications.

The system proposed here offers several advantages in KBS design. First, it allows a logical treatment of data from back-end applications. The back-end is treated as an external database — all knowledge of the operation of the back-end is kept local to the the BEM. Second, it provides a modular approach to system construction. The result of this is a sharp conceptual division between problem solving, and controlling back-end applications. Finally, it requires no modification to existing back-end applications.

We discuss techniques for controlling back-end processes, for translating front-end goals into back-end commands, and for extracting pertinent information from the back-end for use by the front-end. We give an example taken from a BEM for a statistical modelling package.

Keywords

knowledge based systems, logic-based KBS, front-end interface, back-end applications, Prolog.

1 Introduction

In this paper we discuss the problem of interfacing knowledge-based systems (KBS) to existing software packages. Such KBSs can be thought of as *front-ends* to application packages, which are known as *back-end* applications.

There are two main reasons why this activity is a desirable one. First, it is unreasonable to expect that all necessary information for a problem solving domain be supplied in advance. If there are external programs that can provide this information to the problem-solving process, this gives us a useful extension to the knowledge available for problem solving. An example of this might be a KBS that has recourse to a database management system, with a large database of information pertinent to the problem.

Second, application packages are often complex, and require a certain amount of expertise to use properly. Examples of software of this type include engineering modelling packages that require complicated patterns of input parameters to operate correctly. Often the people who use these programs are experts in their field (engineering, say), but are sporadic users of the software, and so require assistance in using the software and interpreting the results. The authors are currently

engaged in a project that addresses this problem through the design of intelligent front-ends for application packages, which provide expert advice both in the use of the software, and in the problem solving domain itself. Of course, it would be possible to rewrite these applications using KBS methodology, but the cost would be prohibitive. There is a great deal of investment made in software, and the use of an intelligent front-end to these applications can extend its useful life considerably.

In this paper, we describe a methodology for building tightly-coupled interfaces between logic-based KBS front-ends, and back-end application packages. These methods have been implemented in the form of a Back End Manager (BEM). This is a KBS component that is responsible for the control of back-end processes, and handles the flow of information between problem solvers and back-end applications in a manner transparent to the front-end.

The BEM was designed to be a general tool for the development of KBS/back-end links. Use of the BEM gives a KBS developer the basic mechanism for interaction, and a framework for specifying the logical role of back-end information. The domain knowledge base can be developed independently, requiring no knowledge of the specifics of the back-end.

Other work on interfaces to application packages include knowledge based systems that interact with database management systems, which have been described by Vassiliou et al ([13]) and Torsun and Ng ([10]). The use of a KBS as a front-end to a software package has been explored in the GLIMPSE system (Wolstenholme and O'Brien [11]), an intelligent front-end to the statistical package GLIM.

The BEM architecture proposed here offers several advantages in KBS design over these systems. First, it allows a *logical treatment of data* from back-end applications. The back-end is treated as an external database. All knowledge of the operation of the back-end — command syntax, calling procedure, output structure, etc. — is kept local to the the BEM. Back-end activities are represented in the domain knowledge base as ordinary logical predicates, and are only treated differently at the meta-level.

Second, it provides for the *modular construction* of systems. Because back-end applications are treated as databases, they are not intrinsic to the operation of the KBS. Back-end applications may be added or removed, with minimal changes to the structure of the system. The result of this is a sharp conceptual division between problem solving, and controlling back-end applications.

Finally, no modifications to existing back-end applications are required. Very often users of software packages do not have access to the source code, and cannot customise applications to work with a KBS. The framework described allows communication with applications using the standard I/O channels.

The paper is organised as follows. In Section 2, we set out the logical role of information received from back-ends. Section 3 describes the different types of applications that may be usefully front-ended. The design of the KBS front-end is described in Section 4. In Section 5, the BEM is described, and various issues of control and communication are discussed. Finally, we present an example from a specification of the BEM for a statistical modelling package.

2 Logical Treatment of External Data

It is rare for a knowledge base for a KBS to contain all the information necessary for problem solving. In many cases, information is missing because it cannot be known by a knowledge engineer at design time, or because the costs of entering and storing this data is prohibitive. Additionally, there are many tasks for which the implementation language of the KBS may be inappropriate. As a result, knowledge-based systems must be able to operate without having immediate access to all required information. The missing information may be obtained from sources external to the knowledge base. We shall treat back-end applications as such a source.

Logic-based problem solvers treat a user's query as a *goal* that is provable from the clauses that

comprise the domain theory. This can be represented by the meta-level proof predicate

$$demo(P, G),$$

which states that for a program P and goal G, $P \vdash G$ (Bowen and Kowalski [2]).

To handle information from an external application, it is convenient to regard the application as an extension to the domain theory. For a back-end application B, the provability relation becomes

$$demo(P \cup B, G),$$

which states the addition of data producible by the back-end program allows us to conclude

$$P \cup B \vdash G$$

So a domain program need not be restricted to a single database, but can be made up of information from several sources. One of these consists of logical formulae, coded in a machine-understandable format. The other is the back-end application (BE). The BEM allows a back-end to be treated as an independent, external database, and the information supplied by the back-end via the BEM is in the form of ground assertions. This approach is the same as that taken by Query-the-User (Sergot [?]), which extends the proof procedure to handle data from a user, rather than a back-end program.

A problem solver using this approach can be implemented by a meta-interpreter that requests missing information from a back-end. Predicates are designated at the meta-level to be either explicitly defined (present in P), or supplied by the back-end (obtainable from B). This is done through a series of statements of the form *backend(Goal)*. When the meta-interpreter encounters a goal G for which *backend(G)* is true, the system issues a command to the back-end application. The result is taken as a solution to that goal. If the back-end is unable to provide a solution, the goal fails. Here, in a fragment from a meta-interpreter that recognises back-end goals, the call *bem(G)* sends the goal G to the back-end manager to be solved.

```
⋮
solve( Goal ) ←
        backend( Goal ),
        bem( Goal ).
solve( Goal ) ←
        not backend( Goal ),
        clause( Goal, Body ),
        solve( Body ).
⋮
```

Because data from the back-end application is in effect being added to the domain program, we encounter the standard problem of ensuring consistency during knowledge acquisition (Kowalski [7]). If we restrict ourselves to a definite Horn clause representation, this generally poses no problem, since a program consisting of definite Horn clauses cannot itself be inconsistent.

Finally, we must consider the effect of issuing commands to an external program. We have treated external data as having a declarative meaning, but must also recognise its procedural effect. Sending a command to an application may change the state of that application, requiring the maintenance of the consistency of the *application's* state. This issue is discussed in more detail in Section 5.4.

2.1 Negated Back-end Goals

Since back-end goals are represented as ordinary predicates, they may be negated, where negation is taken to mean 'failure to prove' in the usual sense of negation as failure. Consider the goal *not file(X)*, where *file(X)* is an back-end goal.

For example, take the clause

> *newFile(File)* ←
> *not file(File),*
> *create(File).*

and the goal *newFile(foo)*. The BEM converts the back-end goal *file(foo)* to the UNIX command `ls foo`, and attempts to execute it.

If there is no such file, the back-end cannot accomplish the activity, and produces an error notification. The BEM must recognise this and fail the front-end goal. The goal *file(fred)* fails, so *not file(fred)* succeeds.

3 Classification of External Sources

In designing the interface between a KBS and back-end application, we require a classification of these applications. Such a classification is useful for providing guidelines on the design choices and implementation methods used in our systems. Our classification is based on analysing two main characteristics of an external application, its *operation mode* and its *internal state*.

According to the operation mode characteristic, applications can be categorised as *batch* or *interactive*. The batch mode of working can be typically described as *read-compute-output-stop*. Although this mode of operation has become less common with the development of new computing tools, batch applications are still used by a large community of users in a variety of domains. When front-ends are built for a batch application they tend to be fairly simple; interaction with the system contains few or no feedback loops. An interactive application, on the other hand, introduces more complex procedures that allow interaction in different stages where the output of one stage is being fed into the input of the next. In general, the batch mode of operation can be treated merely as a special case of the interactive.

External applications can also be divided into those that maintain an *internal state*, and those that do not. For example, many modelling packages keep an internal database for the creation of new instances of data structures and storing of intermediate computations. Applications that maintain their own state pose a difficulty for the KBS, which must model that state in some way. Stateless applications, on the other hand, are comparatively simple to deal with. An example of a stateless application is the set of UNIX utilities that perform processing but do not maintain an internal representation of their activity.

This classification provides us with the means for measuring the complexity of the systems that we attempt to build. The space of possible external applications can be divided along two axes: batch/interactive, and state/no state. The simplest are batch applications with no internal state, while the most complex are those interactive applications that keep an internal state. To build general KBS/back-end systems, we must solve the problems that arise in the more complex case — interactive systems having internal states — and it is these that we will concentrate on.

A final pragmatic restriction on the types of applications that we can handle concerns I/O. While systems that perform *text-based I/O* pose no problem, many applications accept input from non-textual sources (mice, etc), and produce graphical output. Applications of this type are outside the scope of this paper.

4 Design of the KBS

In this section, we discuss the architecture of the combined KBS/back-end, and explain how a knowledge base is constructed for use with the BEM.

4.1 Architecture of the System

The architecture of the complete KBS can be described as in Figure 1. A problem solver (PS) provides a means of performing inference in a problem domain. This domain is represented in a domain knowledge base (KB), consisting typically of rules about the domain. Additional information comes from one or more back-end applications (BE), which are under the control of the Back End Manager (BEM).

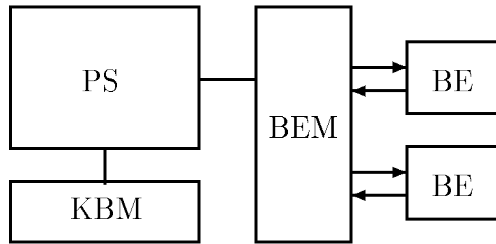

Figure 1: The Combined KBS–BEM Architecture

In our implementation of this architecture, the problem solver is a logic-based, backward reasoning shell written in Prolog, and based on a blending transformer that augments object-level programs with meta-level functionality (Cosmadopoulos and Southwick [3]). The blending transformer replaces the more standard approach of basing the shell on a Prolog meta-interpreter (e.g. APES, Hammond and Sergot [5]).

This problem solver provides some extra facilities necessary for expert system construction, such as rule-trace explanations, a Query-the-User model for user dialogue (Sergot [8]), and a reason maintenance system for maintaining a consistent set of beliefs and reducing redundant computation (Southwick [9]).

4.2 Primitive Actions

We have seen that predicates in our knowledge-base that are to be sent to the back-end are designated at the meta-level by a declaration of the form:

backend(Goal).

Let us call these back-end goals *actions*. We define an action to be a primitive activity that can be accomplished by a back-end application. An action is primitive in the sense that it cannot be broken into subactions in the knowledge representation. Of course, a single action may be converted to a series of back-end commands, but this is not important to the front-end domain representation.

This approach allows the writing of domain rules containing subgoals that are back-end actions. The result is a clean representation of domain knowledge. Actions are predicate subgoals like any other; no special representation is required. The declarative nature of the domain rules is left intact.

As an example of a domain rule that involves back-end actions, suppose we want to use an application package that computes tax payments. Consider a knowledge base that contains the following rule:

totalTax(Person, Tax) ←
    salary(Person, Salary),
    allowance(Person, Allowance),
    tax(Salary, Allowance, Tax).

If the the goal *tax* is specified as an action, it is solved by the back-end, under the control of the BEM.

Backend goals are expected to be fully and correctly specified so that the back-end is capable of executing them with no further need for communication with the front-end. To do this, we need to 'wrap up' these back-end goals with precondition statements that serve to constrain the input arguments to the back-end goal, and postconditions that check the results. Rules that invoke back-end goals, then, take the general form

> *domain_rule*←
> *preconditions,*
> *backend_goal,*
> *postconditions.*

Preconditions ensure that required arguments are instantiated, or are of the correct type. Postconditions can be used to check the results from the back end, or to record the internal state of the back-end for use in future computation. In the *tax* example, the preconditions ensure that input arguments are numbers:

> *tax(S, A, T)* ←
> *numeric(S),*
> *numeric(A),*
> *be_tax(S,A,T).*

The goal *be_tax(S,A,T)* is a action handled by the BEM, and is specified as such by the statement

> *backend(be_tax(S,A,T)).*

4.3 Building a Knowledge Base Using the BEM

The process of building a knowledge base that will interface with the BEM can be split into two parts. First, the system designer specifies the actions needed to complete the task, in a high-level way. The domain knowledge base is then written in the form of rules that use these actions. At this point, the designer needn't worry about the implementation of action definitions in the BEM — this is handled separately.

Once a set of required actions have been identified, the BEM specification of these actions may be written. This involves defining the process whereby front-end goals are translated into back-end commands, and output from the back-end is collected for use by the front-end.

This gives us a clean division between front-end and back-end definitions. Note especially that these two tasks need not be done by the same person: it may be that one person is expert in the domain, while another is conversant with the intricacies of the back-end.

5 The Back End Manager

In this section, we describe the function of the BEM, and discuss some of the particular issues that are involved in coupling KBSs and back-end applications. The most important of these concern control and communication with the back-end.

The operation of the BEM in managing communication can be summed up as follows: a front-end action is translated to a series of commands for execution by the back-end. The output containing the result of this execution is examined by the BEM, and information relevant to the front-end is extracted. This information is represented in a form that is intelligible to the front-end before being passed back.

5.1 Control of the Back End

To interface to an interactive application, the implementation platform must be capable of multi-tasking. At least two processes must be run concurrently; the KBS and the back-end. In addition communication channels are required for data transfer to and from the back-end.

The BEM must initiate the back-end processes, establish communication channels, control dialogue with the back-end, and ensure its graceful termination. In addition the BEM must be able to cope with conditions such as abnormal termination, error messages, or 'unreasonable' output.

In a UNIX environment, (the implementation platform for the BEM) the back-end is started by the BEM as a subprocess, and communication channels are implemented by means of *pipes*. More specifically, in our implementation of the BEM, the back-end's standard input channel is set as the end of one communication pipe, and its standard output channel as the end of the other. As a result, no alteration of the back-end program is required, as long as it communicates via standard input and output.

Communication between the problem solver and BEM can be done by direct subroutine call. Alternatively, greater modularity can be gained (at the expense of greater overhead costs) by establishing a *message passing* protocol for communication (Edmonds et al [4]). If all communication between KBS modules is done via messages, then individual components can be implemented as separate processes. They need not be on the same physical machine, and can communicate over a network if necessary. In fact, the different components need not be written in the same language. The implementation language for each component may be selected to reflect the requirements of that module.

5.2 Translating Back End Commands

One of the main tasks of the BEM involves the translation of a front-end goal, in the KBS language, to a corresponding sequence of commands in the syntax of the back-end. These are then issued to the back-end to produce the desired behaviour. The degree of complexity of this translation will determine the method chosen. We shall identify the characteristics of different back-end translation tasks, and outline two possible approaches to these tasks.

5.2.1 Templates

For back-end applications that have a simple and well-defined input, it may be possible to map input to output in a straightforward fashion. In such a situation a *template* approach is sufficient, mapping a command in the syntax of the front-end to a command or sequence of commands in that of the back-end. The BEM could use templates in constructing back-end commands as follows:

```
bem(Goal) :-
    template(Goal, BE_Command),
    dispatch(BE_Command),
    extract(BE_Command, Result),
    apply(Result, Goal).
```

A template is found for a back-end goal, giving a command that is dispatched to the back-end. A result to the command is extracted from the back-end, using contextual information provided by the back-end command. Finally, the result is applied to the goal, providing any variable bindings required for a solution.

In the following, let us take the operating system itself as a back-end, for explanatory purposes. A simple template for a command that lists the contents of a UNIX directory is

```
template(directory, ls).
```

Even simple, one-to-one correspondence between the front-end and back-end commands may re-

quire some additional processing to yield a command suitable for the back-end. As an example, consider a front-end goal *file_name(File)*, which succeeds if *File* is the name of a file on disk. The BEM translates this command using a template that performs variable substitution and insertion:

```
template(file_name(File), Command) :- concatenate('ls ', File, Command).
```

Additionally, problems of this sort often require some *control* over the translation. This control may be as simple as logical tests (if-then-else), or may require more sophisticated control structures (loops, etc). We may want to write something like:

```
IF input = "directory" AND num_args > 3 THEN
     {
     OUTPUT "ls ";
     FOR i=1 to num_args
        OUTPUT arg(i)
     }
```

This kind of control is needed if input actions may have arguments of varying number and type, resulting in possibly many different forms of output. This gives a one-to-many translation. Alternatively, we may allow certain arguments, if not present, to be taken as defaults, giving a many-to-one translation. These last modifications blur the distinction between filling a template and dispatching a command, since output to the back-end is done during processing.

5.2.2 Grammar-based Translation

In many situations, where the back-end has a command grammar significantly different to that of the front-end, the technique described above will not suffice. These cases are identified by the complexity of the input to be translated. An example is an arbitrary Prolog term, which may be made up of subterms of any form or depth. In these cases, some *interpretation* of the input must be done in order to understand how it is to be translated. The most appropriate way to do this interpretation is by implementing a translator that can parse the input into its components, and deal with these components in the right way.

As an example of input that requires interpretation, consider the interface of a Prolog-based problem solver to a package performing arithmetic using reverse polish notation.

```
(1+2*3(4-5))  --->  +(1,*(2,*(3,-(4,5))))
```

To construct the appropriate output, the input structure must be interpreted. Of course, it is possible to use a template approach to deal with any specific instance of input, but a new template would need to be written for *each* possible input form. This is clearly unreasonable for applications with varied input. Alternatively, one could write special-purpose code to extract certain kinds of structure from an input term. A better solution is to write a *compiler* from the input to the output language. This consists of a parser for the input language, and an interpreter which transforms the parse tree to the required output syntax. The use of grammar rules that direct a parser offers a solution that is general and complete.

If both the BEM and the front-end are written in the same language and the input is a structured term in that language, then the parsing stage may be redundant. In addition, if the input is in some form that is understood by the back-end, then this list can be treated as though it has *no internal structure*, and passed directly through the translator to the back-end. In the above example, if the arithmetic expression can be passed directly to the back-end, no parsing is required. If the expression must be 'understood' to be translated, then a parser should be used.

In order to build a parser, the first step is to define a grammar that will recognise correct forms of input, and then attach translation instructions to the rules of this grammar. This could be done in Prolog, since it is naturally suited to parse Prolog expressions. Alternatively, if a C-language based approach is desired (for reasons of speed or modularity), the best tool to use is the parser-generator `yacc` (Johnson [6]).

An advantage of a grammar-based approach is that it is independent of the language used to implement the problem solver. In addition, tools such as `yacc` and Prolog grammar rules are based on Backus Naur form and as such make the translation of a working grammar to a different implementation language a relatively simple task.

5.3 Output Extraction

Once a command has been issued to the back-end, the BEM must extract the pertinent information from the output of the back-end for use by the problem solver. Back-end applications have varied means of presenting results.

Output from a command may include a preamble, which serves to aid the user, provide header information for column output, etc. This is generally of no interest to the front-end, but may be useful to the extraction process. The components of interest in the output are often record-like; a collection of fields and separators. It should be noted that most applications do not have a well-structured form of output that would allow for a grammar-based extraction process.

The task of extraction is to filter out 'noise' such as the headers and field separators, and to recognise the required data. This is then used by the BEM to produce an *answer* to the original back-end query of a form suitable for the problem solver.

As an example consider the output of the unix command `ls -l`

```
-rwxr-xr-x  1 root     wheel       98304 Sep  7  1989 xdvi
drwxrwxrwx  1 root     wheel         512 Oct  6  1989 lib
```

A suitable Prolog representation of this is terms of the form *entry(Name,Type,Size)*:

```
entry(xdvi, exec, 98304).
entry(lib, dir, 512).
```

To construct a response of this form, the BEM requires knowledge of the form of the output, as well as what constitutes a meaningful answer for the problem solver.

Often, knowledge of the back-end application together with its current state and command history determines how a section of output is to be interpreted. This state knowledge may be used to provide a context sensitive extraction engine. As an example of the way contextual information may be required to interpret output, consider output containing the string `bad`, which may correspond to a (possibly uninteresting) component of a textual message in one situation, while signifying the hexadecimal representation of the number 2989 in another.

The extraction component of the BEM needs the following capabilities:

- It should recognise textual labels in the output of the back-end which serve as locators for data. Examples are column headers or terms of the form "`The Answer is X = 5`".
- It must understand the syntax of the output — in the above UNIX example it must be aware of the significance of each column and be able to determine that the entry for `lib` is a directory.
- It may use knowledge of the commands issued to the back-end to interpret the meaning of the output, giving expectation driven extraction.
- A back-end application may produce a stream of output, from which several pieces are to be extracted to produce an answer. The BEM must have some means of storing and manipulating these answer components.
- It must return results to the front-end in a suitable form.

In our implementation of the BEM, no contextual information is required to disambiguate output. Extraction was implemented using the UNIX utility `awk` (Aho et al [1]) to filter the application's

output, producing Prolog readable terms as a result. Awk is a pattern scanning and processing language, well suited to the extraction and formatting of data. It allows for the location of textual components using wild card pattern matching, and a 'C' like syntax for the extraction/display of particular columns, rows or otherwise specified components.

5.4 Maintaining Back End Consistency

The interface to back-end applications is done not only to get information from the back-end, but also to control the back-end application. This distinction becomes important when we consider the effect that commands may have on the back-end. Many back-end programs have their own internal state, which may be changed by commands issued by the front-end. A numerical analysis package, for example, may keep an internal representation of the state of the analysis. In normal interactive use, this state is modified by commands issued by the user.

If the back-end state can be changed, the state of the back-end must be modelled in the front-end in order to ensure the consistency of the back-end state and to prevent redundant re-computation. If the back-end state changes, and the front-end does not realise this, for example, it will no longer be able to accept back-end activity as valid.

One easy way to ensure the consistency of the back-end state is to prohibit backtracking over back-end goals. Calls to the back-end are treated deterministically — once a command has been issued and executed, it cannot be undone.

While this solves the problem, it does so in an inflexible fashion. There are cases when it is desirable to be able to redo back-end actions, as part of the normal search strategy of the problem solver. One way of implementing this ability is by storing a *snapshot* of the state of the back-end. If, on backtracking, a back-end action is to be redone, the back-end may be restored to its previous state. Some application packages have this functionality built-in. The GLIM system, for example, allows the current state of the system to be dumped and restored, which was used in the implementation of the state restoration facility in GLIMPSE (Wolstenholme [12]).

If such a facility is not available, and backtracking is required, then a model of back-end state must be maintained by the front-end. In the extreme case, restoration of state could involve having to kill the back-end process, and restart it, rebuilding the back-end state from the the front-end model of that state.

In some cases, it is possible to 'simulate' backtracking in the BEM. This can be done where the back-end returns multiple solutions for a goal. When such a back-end goal is evaluated, the BEM finds *all* possible solutions for the goal, and records them. On backtracking, successive solutions may be used. To the front-end, there is no difference in behaviour. This adds a measure of complexity to the BEM, however, which must handle all solutions correctly.

6 An Example Application

In this section we provide an example of the working of the back-end manager, by showing how some front-end actions are handled by the BEM. The back-end application used is the statistical system GLIM [?]. We will define two actions for use in the front-end knowledge base. One defines a vector, and the other performs an arithmetic calculation on that vector. Let us call these actions *vector* and *calculate* respectively.

These actions are used in rules in the knowledge base as ordinary goals, for example:

compute_vector(Name, Len, Vector, NewName, NewVector) $\leftarrow$
  vector(Name, Len, Vector),
  *calculate(NewName, (Name+2)*2, NewVectorValue).*

Suppose this rule is invoked with

compute_vector(vectorA, 5, [1,2,3,4,5], vectorB, NewVector).

When the back-end goals in the body of the rule are encountered by the problem solver, they are sent to the back-end manager for evaluation. The BEM then constructs a series of commands to be dispatched to the GLIM process.

The GLIM command that corresponds to *vector(vectorA, 5, [1,2,3,4,5])* is

```
$DATA 5 vector_name $READ 1 2 3 4 5 $
```

For *calculate(vectorB, (vectorA+2)*2, Value)* the GLIM commands are

```
$CALC vectorB = (vectorA+2)*2 $
$PRINT vectorB $
```

where the `PRINT` command is used to output the value of the calculated vector.

GLIM produces a stream of text as output. For the `$PRINT vectorB $` command above this has the following form;

```
??    6.000    8.000    10.000    12.000    14.000
```

The BEM extracts from this output the Prolog representation of the value of vectorB:

[6.0, 8.0, 10.0, 12.0, 14.0]

This example, though simple, illustrates a number of interesting points. GLIM is an application that has an internal state; the effect of the *vector* command is to set the value of an internal vector. Subsequent commands (such as *calculate*) referring to this vector will use the value corresponding to the current state. One ramification of this is that large vectors do not have to be explicitly represented in the front-end; they may be stored in the back-end database and simply referred to in the front-end.

The *compute_vector* predicate is written using explicit vector names, which are required by the back-end. This formulation is dictated by the needs of the backend, producing a specification that is not back-end independent. Alternatively, the above example could have been encoded as

*alternative_compute_vector(([1,2,3,4,5]+2)*2, VectorValue)* $\leftarrow$

Where the GLIM commands for *alternate_compute_vector* are

```
$DATA 5 tmp_vector $READ 1 2 3 4 5 $
$CALC tmp_vector = (tmp_vector+2)*2 $
$PRINT tmp_vector $
```

This implementation would require additional processing. The term $[1, 2, 3, 4, 5] + 2) * 2$ needs to be decomposed and the vector it contains must be recognised. A name must be generated for a temporary back-end variable corresponding to the vector. In general such terms can be of arbitrary complexity and would necessitate the use of a grammar-based translation method. This contrasts with the first solution, which could easily be handled by a template approach.

7 Summary and Conclusions

We have presented a method for accessing external data for use by knowledge based systems, and have discussed some of the important issues involved in this activity.

Key contributions include:

- A back-end manager that controls back-end application operation and communication.
- Modular construction of KBSs.
- All knowledge of the operation of the back-end is kept local to the the BEM.

- Back-end applications are treated as external databases, allowing the construction of declarative front-end knowledge bases.
- Techniques for translating front-end goals to back-end commands.
- Methods for extracting results from back-end output.

The back-end manager presented in this paper has been implemented, and the methodology described has been tested in a KBS that provides a front-end to GLIM. Our implementation of the BEM is written in Prolog, and is called directly by the problem solver. The extraction component of the BEM, however, is an awk process, connected between the BEM and the back-end via UNIX pipes. During the course of developing the ideas in the paper, we have experimented with implementations using yacc for grammar-based translation, and a 'C' based BEM.

Acknowledgements

Thanks are due to Chris Evans and Damian Chu, who read and commented on earlier drafts of this paper. The authors would also like to thank their colleagues from the Universitat Politecnica de Catalunya (Jesus Lores, Jose Catot and Paul Fletcher) for their many helpful discussions. The work described in this paper was funded by ESPRIT project 2620: FOCUS.

References

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk – a pattern scanning and processing language. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1977.

[2] K. Bowen and R. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[3] Y. Cosmadopoulos and R. Southwick. Using metalevel information for expert system control: A 'blending'transformer approach. In N. Shabolt, editor, *Proceedings of Expert Systems 89,Research and Development in Expert Systems VI*, pages 54–65, 1989.

[4] E. Edmonds, E. McDaid, P. F. A. Prat, and J. Lores. System architecture and KBFE/back-end interface specifications. Focus/Lutchi/Upc/5/1.3-c, Numerical Algorithm Group Ltd, Oxford, UK, 1989.

[5] P. Hammond and M. Sergot. *APES: Augmented Prolog for Expert Systems, Programmer's Manual.* Logic Based Systems Ltd, London, UK, 1987.

[6] S. C. Johnson. *Yacc: Yet another compiler-compiler.* Bell Laboratories, 1978.

[7] R. A. Kowalski. *Logic for Problem Solving.* North Holland, Amsterdam, 1979.

[8] M. Sergot. A Query–the–user Facility of Logic Programming. In P. Degano and E. Sandewall, editors, *Integrated Interactive Computer Systems*, pages 27–41. North Holland, 1983.

[9] R. W. Southwick. A Reason Maintenance System for Backward Reasoning Systems. In *Proceedings of 5th International Symposium on Methodologies for Intelligent Systems*, 1990.

[10] I. S. Torsun and Y. M. Ng. Tightly coupled expert database systems interface. In B. Kelly and A. Rector, editors, *Proceedings of Expert Systems 88*, Research and Development in Expert Systems V, pages 210 – 223. Cambridge University Press, 1988.

[11] D. E. Wolstenholme and C. M. O'Brien. GLIMPSE - a statistical adventure. *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, 1:596–599, 1987.

[12] D. E. Wolstenholme, C. M. O'Brien, and J. A. Nelder. GLIMPSE : a Knowledge–Based Front–End for statistical analysis. *Knowledge–Based Systems*, 1:173–178, 1988.

[13] J. C. Y. Vassiliou and M. Jarke. How does an expert system gets its data? CAIS Working Paper 50, New York University, 1983.