IMPLEMENTING C FUNCTION CALLS IN RULES FOR AN EXPERT SYSTEM SHELL

B.I.B. MADHAV AND NARAYAN HEGDE
Department of Computer Science and Engg.,
Indian Institute of Technology,
Madras 600 036, INDIA

ABSTRACT

In this paper, we report on the implementation of C function calls in rules for an expert system shell. The rule language supports calls to C functions in both antecedents and consequents, meta rules and queries to a relational database. The output of the compiler can be directly loaded or stored as a file. A loader and an inference engine are also provided.

1. INTRODUCTION

Rule based systems constitute the most commonly used means for coding the problem solving know-how of human experts. Rules are problem solving heuristics. Experts tend to express most of their problem solving techniques in terms of a set of situation - action rules.

A rule based system consists of an inference engine , a working memory and a rule base. Each rule consists of a set of conditions called the antecedents of the rule and a set of actions called the consequents of the rules.

The rule language for which the compiler is built supports calls to C functions, queries to a relational database and meta rules. Implementation of a meta rules and querying facility to a database is straight forward. For querying a database, a subset of SQL was used as the query language and an SQL interface to AWK was written. Implementation of calls to C functions involves tackling **homogenous coupling between the user written C functions and the inference engine library.**

2. HOW C FUNCTIONS ARE CALLED IN RULES

The rule language supports calls to C functions in both antecedents and consequents. The C functions can appear as shown

foo(x,y)
temp < foo(x,y)
foo(x,y) > = myval

Here *x* and *y* will be working memory variables and their values will be passed as parameters to the function 'foo'.
The code for all the user written C functions will be written in a single file, say *user.c*

A typical rule will appear as follows.

```
IF
 resistance < 15
AND
 voltage > = foo(frequency)
THEN
 temperature = 25
AUTHOR
 MADHAV
REMARKS
 RULE FOR TEMPERATURE
```

3. IMPLEMENTATION

A lexical analyzer passes the tokens to a recursive descent parser which groups the tokens and forms the sentences of the rule language. An initialization program opens a file called *temp.c*. The following C statements are executed.

```
fprintf(temp_ptr,"%s","func_handler(case_no)");
fprintf(temp_ptr,"%s","int case_no;");
fprintf(temp_ptr,"%s","\n{");
fprintf(temp_ptr,"%s","\n\tswitch(case_no){");
```

temp_ptr is the pointer to the file being written into.

This is equivalent to the following being written in the file *temp.c*

```
func_handler(case_no)
int case_no;
{
        switch(case_no){
```

The initialization program initializes *glob_case*, a global variable to zero.

In the rule base, there exists a declaration section, the name of the C function and the data type it returns will be given in the declaration section. A typical declaration will be as follows

```
foo : function : integer;
x : integer;
y : float;
```

Here *foo* is the function name, it is of type function and it returns an integer.
When a C function such as

```
foo(x,y)
```

is encountered in the rule base, the following C statements are executed

```
fprintf(temp_ptr,"%s","\n\t\tcase ");
fprintf(temp_ptr,"%d",glob_case);
fprintf(temp_ptr,"%s",": ");
```

The following would have been written in the file *temp.c*

```
func_handler(case_no)
int case_no;
{
        switch(case_no){
                case 0:
```

glob_case is assumed to be zero. (*foo* is the first function encountered by the compiler)

The data type of the arguments *x* and *y* are now checked. A function exists for each of the data types viz. integer, string and decimal. They will return the value of the variable from the working memory. Let's suppose that the type of the variables *x* and *y* are integer and float respectively. The following C statement is executed.

```
fprintf(temp_ptr,"%s","\n\t\tfoo(ivalue(x),fvalue(y));");
```

fprintf(temp_ptr,"%s","\n\t\tbreak");

The following would have been written in the file *temp.c*

```
func_handler(case_no)
int case_no;
{
        switch(case_no){
                case 0:
                foo (ivalue(x),fvalue(y));
                break;
```

Now *glob_case* is incremented by one. This is done for all the functions. A typical *temp.c* file will appear as follows.

```
func_handler(case_no)
int case_no;
{
        switch(case_no){
                case 0:
                foo(ivalue(x),fvalue(y));
                break;
                case 1:
                foo1(svalue(a));
                break;
        }
}
```

This file *temp.c* and the file in which the user has written his C functions *user.c* are compiled and linked with the inference engine library and when a function call is encountered the inference engine calls the C function func_handler with the appropriate case_no as an argument. This case_no is kept in the symbol table along with the function name. Thus a **homogenous coupling between user written C functions and inference engine library is** provided.

4. CONCLUSIONS

A method for homogenous interface between user written C functions and inference engine library is proposed. This sort of interface provides a *tight coupling* between user written C functions and inference engine by eliminating interaction through intermediate files. The compiler for the rule language, an inference engine and a loader have been implemented on the SUN 3/60 workstation. A few expert systems have been built using this shell.

REFERENCES

1. Kernighan B.W. and Ritchie D.M., *The C Programming Language*, Prentice Hall, 1976.

2. Aho A.V. and Ullman J.W., *Principles of Compiler Design*, Addison Wesley, 1985.

3. *Programming Utilities for the SUN workstation*, SUN Microsystems Inc. California.