McESE - McMaster EXPERT SYSTEM ENVIRONMENT

F. Franek* and I. Bruha†

Dept. of Computer Science and Systems
McMaster University
Hamilton, Ont.
Canada L8S 4L7

Abstract

McESE is an expert system environment (a software tool) designed to help create problem-specific shells with incomplete and uncertain knowledge, fast and compact expert system applications in a particular programming language. Specialized software of McESE is written in C and facilitates handling of all aspects of dealing with rule-based knowledge bases. Practical and theoretical aspects of McESE are discussed. For more details see [FB].

1. INTRODUCTION

McESE (McMaster Expert System Environment) is a software tool to build problem-specific shells and create expert system applications. It is designed to satisfy the goals listed below (not in the order of their significance):

- allow the user to deal with imprecise and incomplete knowledge in McESE knowledge bases with a declarative formalism that has a satisfactory degree of expressive power;
- allow the user to customize the shell as so it handles uncertainty in the way of his preference;
- allow the user to create expert system applications in a particular programming language (C, FranzLISP, and SCHEME are available at the moment), with a point of reference being the application rather than the knowledge base (so the creation of such an application resembles ordinary programming as much as possible);

*Research supported by SERB 5-26397 and NSERC OGP0025112 research grants.
†Research supported by NSERC A20037 research grant.

- allow the user a natural (hierarchical) connection of different knowledge bases in an application;
- allow rapid prototyping;
- allow fast inferring.

2. HOW THOSE GOALS ARE ATTAINED

(2.1)

In McESE the user can encode the domain knowledge in rules of the following form:

$$TERM1 \ \& \ TERM2 \ \& \ ... \& \ TERMn \ == CVPF \ ==> \ TERM$$

where *cvpf* abbreviates "certainty value propagation function".

The *meaning* of a simple rule $TERM1 \ \& \ TERM2 \ == F \ ==> \ TERM3$ is: if we are certain with value $v1$ that $TERM1$ is true, and if we are certain with value $v2$ that $TERM2$ is true, then we are certain with value $F(v1, v2)$ that the left hand side (LHS for short) holds, and so we are certain with that value that $TERM3$ holds.

An (meaningless) example of a McESE rule:

$$R1: \ .8 * P1(x, y)[>= .3] \ \& \ -P2(z) \ == F2 \ ==> \ P3(x, y, z)[< .5]$$

where $R1$ is the rule's id, $P1$, $P2$, and $P3$ are predicates, $F2$ is a cvpf, x, y, and z are predicate variables, "-" stands for negation, .8 preceding $P1$ is the weight of the first term (must be a real number between 0 and 1 inclusive; if omitted, it is assumed to be 1), [>= .3], [< .5] are threshold directives (>= and > in [ ] are threshold operators, and .3 and .5 in [ ] are threshold values, must be real values between 0 and 1 inclusive).

A predicate, possibly preceded by a weight, possibly preceded by "-" or " " (denoting negation), and possibly followed by a threshold directive, is called a term.

The *firing* of the above mentioned rule consists of: first, for the rule to be *fired*, all predicate variables in the rule must be bound to some data structures, called objects. Let x be bound to the object X, let y be bound to the object Y and let z be bound to the object Z. Second, the certainty values (real values between 0 and 1 inclusive) of all LHS terms must be known. Then the certainty value of the right hand side (RHS for short) predicate can be computed as: Let $v1$ be the value of the first term of the LHS of the rule $R1$ (i.e. the term $.8 * P1(X, Y)[>= .3]$), let $v2$ be the value of the second LHS term of the rule $R1$ (i.e. the term $P2(Z)$). Then the value of the LHS is $F2(v1, v2)$. ($F2$ must be a function of two real arguments returning a real value between 0 and 1 inclusive, or -1.) From this the value of the RHS predicate $P3(X, Y, Z)$ is determined by

the threshold directive. In this case, if the value of the LHS is strictly less than .5, the value of $P3(X, Y, Z)$ will be set to 1, otherwise it will be set to 0.

The value of a LHS term is computed from the value of the term's predicate according to the weight and the threshold directive. E.g. the value of $P1(X, Y)[>= .3]$ will be 1 if the value of $P1(X, Y)$ is greater or equal to .3, otherwise it will be 0. If the weight is not specified, it is assumed to be 1, and so the final value of the term is completed at this point. On the other hand when the weight is specified, as in this case, the final value of the term is obtained by multiplying by the weight.

The value of $P2(Z)$ will be (1-value of $P2(Z)$).

If the cvpf $F2$ returns -1, then the rule is considered not *fired*.

Rules in this form allow to capture an imprecise, uncertain and incomplete knowledge, since the rules are guaranteed to *fire* for any values of LHS terms (except some situations when the firing is prevented by the cvpf), and only the resulting value of the RHS predicate is affected by the values of LHS terms. Thus, we can formulate our rules in vague terms, as in this example from an expert system to play a card game Canasta:

$$opponent_collect(x)\ \&\ used_stck_high\ = F => \quad discard(x)$$

where we can never be sure if the opponent really collects x, and when the used stack is high. But we can build into the knowledge base enough information to estimate these facts numerically (based on current input data) and these numbers project via cvpf F into the value of $discard(x)$. Even in the case of complete lack of information, say if the value of $opponent_collect(x)$ is 0 we may want to associate the value of .25 with $discard(x)$ (since there are 4 possible types the opponent may be collecting and so in the absence of any relevant information a good guess is that there is .25 chance of the opponent collecting x) and that's what cvpf F can do.

(2.2)

If no cvp function in a rule is stipulated, the default one is used. If unchanged by the user, it is so-called weighted cumulative evidence computed according the following formula: let

$$w = w1 + w2 + w3 + .. + wn$$

where $w1$ is the weight of $term1$, $w2$ the weight of $term2$, ... , wn is the of $termn$. Let $v1$ be the value of LHS $TERM1$, $v2$ the value of LHS $TERM2$, ..., vn the value of LHS $TERMn$. Then $(v1 + v2 + ... + vn)/w$ is the value of the LHS.

As any cvpf can be defined as the default choice, one can pre-determine that all rules in the knowledge base will be handled uniformly, in essence fixing a particular method of the treatment of uncertainty in the whole knowledge base.

(2.3)

Most of expert systems shells are either presented with the knowledge representation language as the main language of the application, and hence the application is *centered* around the *model* (knowledge base), and the procedural parts are connected to it by different means (in the case of OPS languages and PROLOG it is the only language), or they themselves are written in the language of application (for example KEE in LISP). We tried to give the user a possibility to write an application in the usual way, at least the procedural parts, and in the programing language of his choice, but still preserve the possibility of having access to a declarative knowledge base when needed. This is achieved by *extending* a particular programming language by McESE commands to facilitate all required communication between the application and the knowledge bases. The software to performer the communication is written in C, but is transparent to the user. Thus, a particular application is completely built using a single programming language and the language of McESE rules. At this point, McESE extensions of C, FranzLISP, and SCHEME are available. Note that this shift in emphasis changes the focal point from knowledge base to the application in an effort to allow for ordinary programming techniques, methods, and experience to be utilized.

(2.4)

Predicates which never occur on RHS of any rule correspond to facts and observations; we shall call them level 0 predicates for they will be on level 0 of the knowledge tree (see 3.2). They represent data input nodes of the knowledge tree. Their values are not derived (inferred) using rules, they must be obtained from so-called predicate service procedures. These may be ordinary procedures to supply the facts and/or observations, or they may in fact be other expert systems. This mechanism allows for convenient partitioning of the domain knowledge into a hierarchy of knowledge bases (or more precisely expert systems), see Fig. 1.

(2.5)

Since McESE built-in inference engine automatically prompts the user for the result of the invocation of a predicate service procedure in the case that the predicate service procedure is not available to the system (and similarly for cvpf's), one can just test and modify rules in the knowledge base without the overhead of building the complete application. Moreover since McESE interactions and inferences are identical in McESE-C, McESE-FranzLISP, and McESE-SCHEME, one can quickly build a prototype in McESE-FranzLISP (utilizing versatility and flexibility of FranzLISP) to verify the methods and approaches, and when satisfied, the knowledge bases can be used as they are for the McESE-C application.

(2.6)

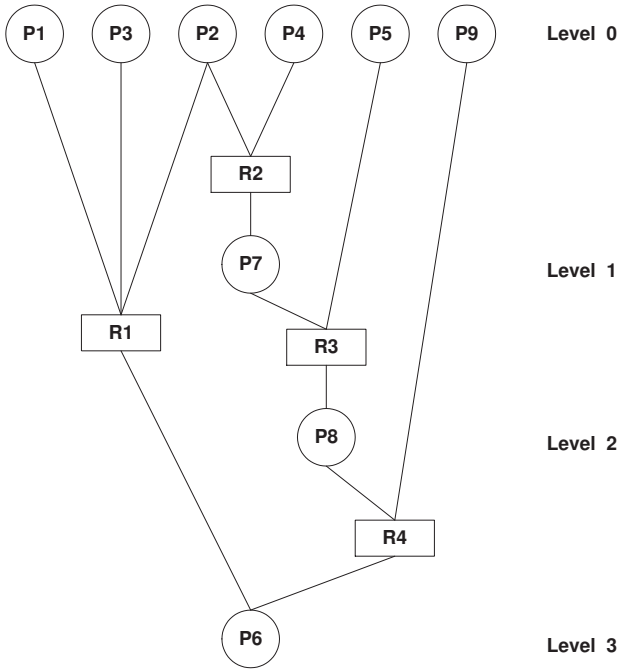

Figure 1: Fig. 1

McESE knowledge bases are first compiled before they can be used in an application. The compiled knowledge base (for short called knowledge tree) allows for direct linking of relevant predicates, so only relevant rules are in fact considered when a predicate must be evaluated. Thus inferring with such a knowledge tree amounts to a *walk* through the tree, and hence the speed of inferring depends entirely on the depth of the knowledge tree rather than on its size. The result is a fast performance, knowledge base queries are quickly evaluated and returned to the application program.

3. McESE COMPONENTS

McESE source knowledge base

McESE source knowledge base consists of two separate sets: the set (RSET) of McESE rules (in descriptive form), and the set (FSET) of corresponding cvpf's (in procedural form). In addition to the above mentioned syntax of McESE rules, each rule has to satisfy the condition that all predicate variables occurring in predicates of the LHS, must be variables of the RHS predicate, and vice versa, all RHS predicate variables must occur as predicate variables of some predicate of the LHS.

Although RSET and FSET are maintained in separate files, McESE built-in

editor allows the user to edit both parts together in two windows on the screen. This simplifies the task of knowledge and software engineering with McESE.

McESE compiled knowledge base

The McESE source knowledge base as a set of rules (RSET) and a set of cvpf's (FSET) is a structure well suited to store the desired knowledge in its declarative and procedural form respectively, suitable for humans to understand, modify and manipulate easily. But for many reasons it is not a well suited structure for a computer program to access and *infer* with it. Thus, McESE first *compiles* the knowledge base into a data structure which may be visualized as a tree capturing the essential relations between predicates.

McESE compiler parses source RSET providing syntactical checking of rules, providing as well other checking as described above, and builds the knowledge tree in main memory with address links relative to the beginning of the knowledge tree. After successful compilation the resulting data structure is recorded in a disk file.

McESE inference engine and inferring

McESE inference engine provides the mechanism for *inferring*. It can work in two basic modes, forward chaining and backward chaining. Backward chaining from a given predicate (node) with given bindings for its variables is performed as depth-first walk down to level 0 nodes (with simultaneous propagation of bindings for predicate variables). Only the required 0 level nodes are activated (and so appropriate known facts are fetched and/or appropriate observations are made) and then the resulting certainty values are propagated (and recorded in the knowledge tree, too) back through the selected subtree to the required node. The backward chaining mode has four submodes which amount to rule conflict resolution: *max* mode, *sufficient max* mode, *min* mode, and *sufficient min* mode. Forward chaining is implemented only from 0 level up, to a specified level. Specified nodes from level 0 and their ascendents up to the specified level are evaluated.

The inference engine can work in two modes as far as explaining what it is doing: the silent mode when all inferring is transparent to the user and only the resulting value is available, or in trace mode when all inferring is done on the screen, rule by rule, predicate by predicate, with all relevant information being displayed, too. The trace mode is useful mainly for testing and debugging of knowledge bases.

A run time consistency checking takes place: McESE allows to preset for the knowledge base what inconsistency level can be tolerated. If the inconsistency level tolerance is exceeded, ALARM is issued and the inferred value is returned to the application. Also, a run time completness checking takes place: in the case a predicate cannot be evaluated, ALARM is also issued and -1 is returned by the inference engine.

Explanation component

As a simple explanation mechanism the inference engine keeps track of the subtree used for the *max* (and *min*) evaluation for each predicate evaluated during the last inference cycle (and similarly for the min evaluation), and displays it when asked for, together with the input data which affected the particular values of facts on level 0 at the time of the evaluation.

References

[FB] F. Franek, I. Bruha, **The McESE project**, *Tech. Rep.*, Dept. of Comp. Sci. & Systems, McMaster University, Hamilton, Ont., Canada, 1988.