APPLICATION OF FUZZY LOGIC TECHNIQUES IN THE BSS1 TUTORING SYSTEM

KAI WARENDORF
SU JEN TSAO
School of Applied Science, Nanyang Technogical University,
Nanyang Avenue, Singapore 639798

The Brilliant Scholar Series 1 (BSS1) is a tutoring system currently used by several thousand home and school users in the learning of curricular subjects such as mathematics and sciences. It is an AI based tutoring system using heuristics to interact with users and monitor their progress. It is believed that the use of fuzzy logic techniques can improve the performance of such tutoring systems, by introducing intelligent features which can better manage the student's learning. A general fuzzy logic engine was designed and implemented to support development of intelligent features for BSS1. In order to develop such features, the problem had to be suitably modeled and a knowledge base created, followed by testing and tuning with appropriate procedures.

The usefulness of such a fuzzy system depends on the engineer's ability to model the problem suitably, define fuzzy variables and suitable membership functions for their fuzzy sets, and develop a comprehensive set of rules relating input and output variables. The average engineer who may not be equipped with this knowledge is still able to design (simple) systems by just manipulating the fuzzy set functions and the rules. Internal parameters which are generally provided by the user of the engine (the expert system designer) are fixed in this application by choosing widely used methods which have proven effective and are commonly referred to in the literature.

INTRODUCTION

General Requirements of Intelligent Features for BSS1

BSS1 provides students with a large pool of questions collected from past year examinations. The questions are organized into different topics and sub-topics, and additional information, such as the level of difficulty and expected time needed for each question, is available. Some of these past year examination questions have been adapted such that all are multiple-choice questions. BSS1 has a pool of teachers who prepare all the answer choices as well as solutions, hints and notes for each question.

The aim of BSS1 is to help the student manage his learning as he works through the pile of questions. BSS1 is able to monitor the student's progress, strengths and weaknesses by keeping student profile parameters such as attempted questions, the points scored, the time taken and the trend in performance. Together with other information such as the relative importance of topics, their relation to one another, and the period of time left for the student to prepare for his exams, the system should be able to manage the student's learning intelligently, for example, by suggesting topics/sub-topics the student should embark on first. Introducing such intelligent features is the ultimate aim of the joint-project.

As can be seen, the problem is not a very structured one. A human teacher (expert) trying to solve this problem might use intuitive guidelines like ' *If this sub-topic is very important in the exams, and the student is still weak in it, and he hasn't spent enough time on it yet and the exams are near, then he should embark on it straight away*'. Such a kind of guideline or rule encompasses a lot of vagueness, for example, how important exactly is "very important", or how much time is meant by "not enough." Trying to define all these in a complete and precise manner (in numerical terms), causes the rules to become very cumbersome, especially when borderline cases have to be handled differently in order to avoid awkward discontinuities.

Choice of Fuzzy Logic Model

Fuzzy logic control allows the human description of the physical system and of the required control strategy to be simulated in a reasonably natural way. A fuzzy logic controller consists of a knowledge base which is usually expressed as a number of 'IF-THEN' rules based on the domain expert's knowledge. These rules describe in almost natural language the relationship between the input parameters, and the desired control

output. Fuzzy logic control is very suitable for systems which rely heavily on human or expert experience and intuition.

A general fuzzy logic inference engine based on such a model is a very flexible tool for solving a great variety of problems of the intuitive nature discussed. The most important consideration is to model the problem suitably based on this inference approach and develop a good knowledge base for it. In this project, the fuzzy approximate reasoning approach has been adopted. This is the fuzzy logic model used extensively in fuzzy logic control, the main application of fuzzy logic which has already gained a significant role in daily industrial practice.

A GENERAL FUZZY LOGIC ENGINE

Commercial fuzzy logic knowledge-based system development tools are available, most of which are targeted to the area of building dedicated fuzzy logic control applications. These packages provide a convenient fuzzy programming environment, in which the application programmer is able to specify the knowledge base in a fuzzy programming language (usually with the help of graphical editors) and possibly include fragments of a target programming language source code. This fuzzy programming language source is then converted to the target language, 'C' for example. This generated source code has the knowledge base information embedded in it. With (possibly) some additional code provided by the application programmer, it will be compiled into an application with a dedicated fuzzy logic engine. A general fuzzy logic engine is required for use as a tool in developing a suitable knowledge base for some applications. A fuzzy logic engine is basically an inference machine, which repeatedly

- takes in a set of input parameters,
- performs fuzzy inference based on given knowledge in the form of rules that relate input conditions to required output values (problem solving), and
- channels the inferred output as desired (e.g. a file)

A general engine does not have any knowledge built into it. It becomes a different problem solver with each different knowledge base supplied to it. This would be a simple tool in which the user concentrates on defining the knowledge base and verifying/assessing it by observing the inputs and respective outputs of the engine (simulation). The knowledge base can be modified for tuning the engine, or changed to solve to a different problem These can be done without the need for any programming and compilation.

General Design Considerations

Before the engine program goes into its normal execution (getting input values, fuzzy reasoning and then writing the outputs), it has to read in a proper specification of the knowledge base and store this information in appropriate data structures so that it can later be used for inference. Besides this mechanism to assemble and represent the knowledge base at run time, an inference mechanism is built into the engine, which is closely related to the way the knowledge base is represented. The main design issues in the development of a general fuzzy logic engine falls mainly into the following three areas, which will be discussed in the following sections::

1. Deciding on design parameters such as definition of fuzzy sets, fuzzification methods, reasoning techniques and defuzzification techniques. A truly general fuzzy logic engine used as a tool in helping to develop fuzzy logic knowledge-based systems should cater to the user with some flexibility in choosing these parameters, usually as part of the knowledge base specification.
2. Choosing appropriate data-structures to represent the knowledge base so that they support the various design parameters and are storage and computation efficient.
3. Defining a way to specify the knowledge base (KB) to the engine. For the engine to read in a KB specification and directly build up the internal data structures to represent the KB, this specification would have to be in a format compatible to the chosen data structures. It will usually be too difficult and unnatural for the user to specify the KB in such a format. Instead a natural language like format is desired. Thus some form of a parser for this type of KB specification is needed to translate the information to a suitable format for the engine. This parser can be implemented as part of the engine or as a separate tool to prepare the necessary KB specification file for the engine.

Engine Design Parameters

The main elements of a fuzzy logic engine are: a knowledge base, a fuzzification unit, a fuzzy logic reasoning unit and a defuzzification unit (see Figure 1, adapted from [Yan et al., 1994]). Each of these entails

some design parameter choices which give the engine its characteristic nature. An example of such a knowledge base is given in Appendix A.

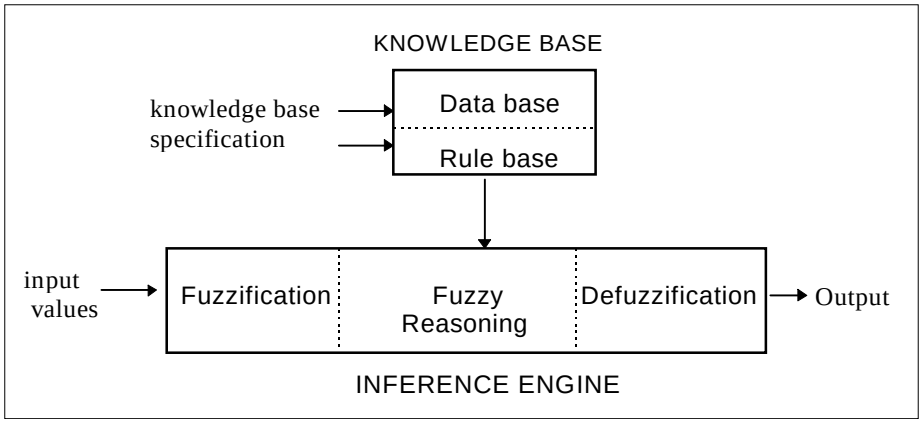

Figure 1: General Structure of a Fuzzy Logic Engine

Knowledge Base. The knowledge base (KB) contains two types of information: a data-base and a rule-base.

- Data-base

  The basic function of the database is to provide the necessary information for the proper functioning of the various modules in the inference engine (fuzzification, reasoning, defuzzification). The membership functions for the fuzzy sets used as linguistic values for each linguistic variable are defined here. They give meaning to the linguistic variables and values used in the rules.

  An important issue here is the way to define the membership function and hence the shape of fuzzy sets. There are two general ways to define this:

  1. Numerical

     A numerical definition for fuzzy set A in discrete universe X specifies all the points (x, $\mu A(x)$) for all $x \in X$.
  2. Functional

     A functional definition is an analytical expression which allows $\mu A(x) \; \forall \; x$ to be computed. This can be used to define both straight-edged shapes like triangles (TRI (a, b, c)) and trapezoids (TRAPEZ(a, b, c, d) and curved shapes such as the bell-shaped π-function (π(a, b)). Only the function type and several parameters to define its exact shape need to be kept.

  The approach chosen is somewhere in-between numerical and functional. It is to specify all the points (x1, $\mu A(x1)$), (x2, $\mu A(x2)$), (x3, $\mu A(x3)$), ..., where x1<x2<x3<... for all the vertices which define the shape of the fuzzy set. These points are connected one to the next by straight lines. For example, the fuzzy set FAST in Figure 2 will be defined by the points (50,0), (75,1) and (100,1).

  This approach is efficient in representing triangular and trapezoidal fuzzy sets (which are commonly used because of their simplicity). The advantages of this approach are simplicity and flexibility. Smoother (curved) shapes can be approximated by using more points. The fact that different shapes can be handled uniformly helps to simplify the task of defuzzification.

- Rule-base

  The rule-base consists of a set of rules to relate input fuzzy values to output fuzzy values. These rules are usually of the following format:

  IF < input state/condition > THEN < output (list)>
  (antecedent) (consequent (list))

  The rule antecedent is a logical combination of atomic fuzzy propositions. If these atomic propositions involve different linguistic variables, it is a multiple input system. For a multiple output system, the rule-consequent can be a list of fuzzy propositions. e.g.:

  IF <antecedent> THEN (y1 is Y1) ALSO (y2 is Y2).

  Notice that ALSO means that both outputs are present. If y1=y2 (same variable), ALSO can be interpreted as a union of the two consequents. Each consequent can also be a logical combination of fuzzy propositions, but is only meaningful between propositions involving the same linguistic variable. For example:

  IF <antecedent> THEN ((y1 is Y1) AND (y1 is Y2)) ALSO (y2 is Y2).

The following characteristics are made available for the developed engine:

1. The rule-antecedent can be a logical combination (of any arbitrary complexity) of atomic fuzzy propositions using the connectives 'OR', 'AND', 'NOT' as well as parentheses '(' and ')'. For example:
   (a is A1) AND ((b is B1) AND (c is C2)) OR NOT ((c is C1) OR NOT NOT (a is A2))
2. Multiple outputs (consequents) in a single rule are supported, though each consequent can only be an atomic fuzzy proposition. The author does not think this is a serious impairment as it is most common for individual rule consequents to be stated as atomic fuzzy propositions.

Fuzzification. Fuzzification is the process of converting crisp values into a fuzzy set, so that they are compatible with reasoning techniques based on fuzzy set representations. In the area of interest, crisp input values will be used because they are usually readily available or can be easily acquired. The most common fuzzification technique is to 'conceptually' map the crisp value into a fuzzy singleton within its respective universe of discourse. This strategy has been widely used in fuzzy control applications since it is natural and simplifies implementation of the reasoning unit.

Fuzzy Reasoning. In the design of an inference engine, the approach used predominantly in fuzzy knowledge based control is known as the individual rule-based inference (firing) [Driankov et al., 1993]. The MAX-MIN reasoning technique was used on this approach [Klir and Yuan, 1995]. The basic function of a reasoning unit is to compute the overall value of the output variable based on the individual contribution of each rule in the rule-base. The process is as follows:

1. The fuzzified inputs (e.g. fuzzy singletons) are matched to the fuzzy sets in the corresponding atomic fuzzy propositions of each rule antecedent, thus giving a degree of satisfaction to each atomic proposition.
2. The overall degree of match (fire-strength) for that rule is then computed by combining the resulting degrees of satisfaction from (1) according to the logical connectives used to combine the atomic fuzzy propositions. This fire-strength represents the degree of satisfaction of the (compound) fuzzy proposition in the antecedent of the rule.
3. Based of the fire-strength from (2), the value of the output in the rule consequent is modified, i.e.: clipped or scaled. This gives the fuzzy set representing the fuzzy value of the inferred output for that particular rule.
4. The set (union) of all clipped (or scaled) output values of matched rules represent the overall fuzzy value of the output.

The method leading to clipped fuzzy sets is based on Mamdani's minimum operation rule Rc, and is known as MAX-MIN reasoning (as illustrated in Figure 2). A very similar method that leads to scaled fuzzy sets is based on Larsen's product rule Rp and is known as MAX-DOT reasoning [Yan et al., 1994]. Figure 2 illustrates the MAX-DOT reasoning for a rulebase with the following two rules:

Rule 1 IF x is A1 AND y is B1 THEN z is C1
Rule 2 IF x is A2 AND y is B2 THEN z is C2

The crisp inputs x_0 and y_0 are converted into the fuzzy singletons A' and B' respectively. Matching A' to A1, the degree of satisfaction of the atomic fuzzy proposition (x is A1) is given by $\mu A (x_0)$. The same applies to (y is B1). Thus the fire-strength of Rule 1 is $\min\{ \mu A (x_0), \mu B (y_0) \}$.

The developed engine uses the same individual rule based firing approach, treating each crisp input as a fuzzy singleton. It can be either MAX-MIN or MAX-DOT depending on the particular method used in the defuzzification module. The strategy employed by the author is to let the fuzzy reasoning module compute the fire-strength for each rule, and save the clipping (or scaling) fire-strength as the 'weight' for each fuzzy set of the output variable. The overall output of the rule-base as an aggregated fuzzy set of a certain shape is not of much interest here. Instead a crisp value that best describes this overall output is desired. The defuzzification unit derives this value based on the 'weight' of each output variable fuzzy set. Some defuzzification methods treat the fuzzy sets as clipped; others treat them as scaled.

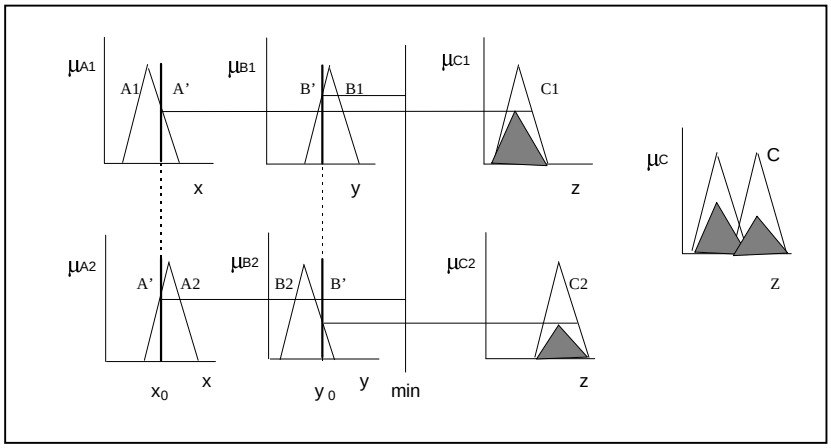

Figure 2: MAX-DOT Fuzzy Inference under Crisp Inputs

Defuzzification. Defuzzification serves to map a fuzzy set to a crisp value which best describes the center of distribution of the fuzzy set. Many different methods exist, some of which will be considered here. For the purpose of discussion, consider a set of rules with output variable Z. Each rule consequent states a fuzzy set Z_i for Z (where Z_i is the ith fuzzy set for Z). For example, a rule's consequent may state (Z is Z_2). Let the inferred output for each rule k be clipped/scaled versions of the corresponding Z_i and let's call this inferred output C_k. The purpose is to defuzzify the aggregated inferred output (from the set of rules) and obtain a crisp value that best describes the output Z.

- Center-of-Area / Gravity

  This is the most well-known defuzzification method. It involves taking the union U of all the C_k 's (consequent fuzzy sets clipped or scaled by the fire-strengths of their respective rules) and then computing the center of gravity CG as follows (adapted from [Driankov et al., 1993]):

  $$CG = \int z\, \mu_U(z)\, dz \,/\, \int \mu_U(z)\, dz \quad = \int z \max(\mu_{C\_k}(z))\, dz \,/\, \int \max(\mu_{C\_k}(z))\, dz$$

  This method (Method 1) of defuzzification is implemented in the engine, although not following exactly the procedure implied by the equation. The procedure designed by the author is as follows:

  1. For each fuzzy set Z_i of an output variable Z, keep a current clipping/scaling factor (which is updated by the fire-strengths of rules which have Z_i in their consequent). This current factor for each Z_i should be the maximum of all fire-strengths encountered (union of consequents involving the same Z_i).
  2. After all the rules are fired, the final clipping/scaling factor f_i of each Z_i is known. The union U is defined by:

  $\mu_U(z) = \max \min(f\_i, \mu_{Zi}(z))$ (union of all clipped Z_i's)

  or

  $\mu_U(z) = \max(f\_i\, \mu_{Zi}(z))$ (union of all scaled Z_i's)

  3. The CG can then be calculated by dividing the universe Z into small steps and taking the sum to approximate the integration:

  $$CG = \Sigma\, z\, \mu_U(z) \,/\, \Sigma\, \mu_U(z)$$

- Center-of-Sums

  A similar but faster method is the center-of-sum (COS) method. The motivation is to avoid computation of the union U which can be quite complex (as seen from the above discussion). This method takes the sum (instead of the union) of all the C_k's and computes the center CS as follows:

  $$CS = \int z\, \Sigma\, \mu_{C\_k}(z)\, dz \,/\, \int \Sigma\, \mu_{C\_k}(z)\, dz$$ (adapted from [Driankov et al., 1993])

  Equivalently (bringing summation out of the integration),

  $$CS = \Sigma\, (\text{moment of } c\_k \text{ about } z=0 \text{ axis}) \,/\, \Sigma\, (\text{area of } c\_k)$$

  In this method, overlapping areas between sets would be reflected more than once. Figure 3 shows an example with scaled fuzzy sets.

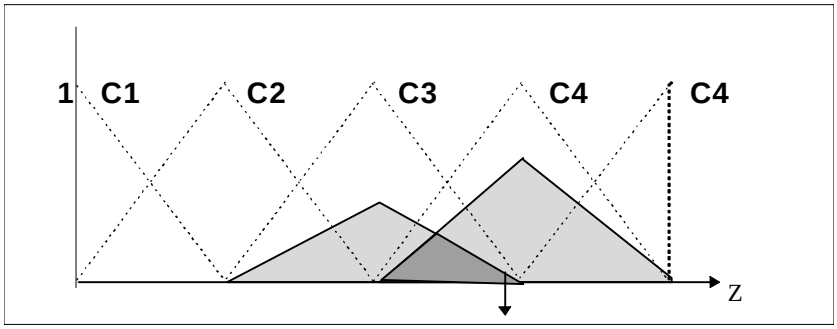

Figure 3: Centre of Sums defuzzification on Scaled Fuzzy Sets

The COS method can be easily implemented in the engine. It only involves computation of the area and moment for each C_k (after rule k is fired), which are then summed up for all the rules. The computation of area and moment is simple here because it only involves a single defined fuzzy set. It is especially efficient if scaled fuzzy sets are used in which case:

Area(C_k) = f_k Area(Z_i)

Moment(C_k) = f_k Moment(Z_i)

f_k is the fire-strength of rule k. Area(Z_i) and Moment(Z_i) are the area and moment of the original fuzzy set Z_i and these can be pre-computed and stored.

However the COS method can give quite different results from the centre-of-area (COA) method especially when the overlaps are large, or when many separate rules have the same fuzzy set Z_i as its consequent. The COA method is in fact a weight-counting-method.

Suppose there are three rules that all have the fuzzy set C1 in the rule-consequent: the first is matched to 0.2, the second to 0.4 and the third to 0.6. Suppose there is a fourth rule with a rule-consequent C2, and it is matched to 0.7. If the COA method is used, the union of C1 clipped (or scaled) to 0.6 and C2 clipped to 0.7 is taken. Thus the result is a defuzzified output belonging more to C2. This does not take into account the fact that there are three rules producing clipped versions of C1. A weight counting method like the COS will take this into consideration. However, it is hard to say whether this effect of weighting is desirable and it all depends on the meanings the rule-designer had in mind [Driankov et al., 1993].

In developing the engine the author implemented another method (Method2) somewhat in between the COA and COS. It involves both summation and union but does not have the same weight-counting property of the COS method. It is in fact more like an approximation to the COA method. As was discussed for the proposed COA implementation, a final scaling (scaling is used here for efficiency) fire-strength f_i is found for each fuzzy set Z_i of the output variable Z. f_i is the maximum fire-strength of all rules with (Z is Z_i) as its consequent (union of consequents with the same Z_i). The defuzzified output is given by

$$CG2 = \Sigma f\_i M_i / \Sigma f\_i A_i$$

where Mi and Ai is the moment (about the z=0 axis) and area of Z_i respectively. As before these values can be pre-computed. This is in fact a very simple and intuitive method that gives quite close results to the COA method.

In summary, the two defuzzification methods implemented in the engine are:

- Method 1: COA method involving the union of 'clipped' fuzzy sets, which corresponds to one of the most well-known defuzzification technique in the literature. However it is computationally expensive. Computing the union U requires dividing the output universe Z into many small steps and evaluating μU (Z) for each Z value. Computation of the CG then requires the summations Σ z $\mu U(z)$ and $\Sigma \mu U(z)$.
- Method 2: Approximation to method 2 (COA) involving summation of 'scaled' fuzzy sets, which is much more computation efficient. Computing CG2 merely requires summing the areas and moments of each fuzzy set. These areas and moments are pre-computed only once before the fuzzy inference and defuzzification begin. This method gives close results to method1, and hence is preferred for practical applications. Appendix B shows some inference / defuzzification results for a sample application of the fuzzy logic engine. The results of both defuzzification methods are given for comparison.

DATA STRUCTURES

Variables and Fuzzy Sets

As the engine reads the knowledge base file, it stores the declared variables and its definitions in a table, each entry of which is a C-structure as illustrated in Figure 4. Entities like the number of variables, number of fuzzy sets for each variable and the number of points to be stored for each set are known to the engine before each list of items are read in. So, the engine is able to allocate the exact memory space for them. The order in which the variables and sets are read in determines their order (IDs) in the variable and set tables. These IDs are important as they will be used by the rules to reference these variables and sets.

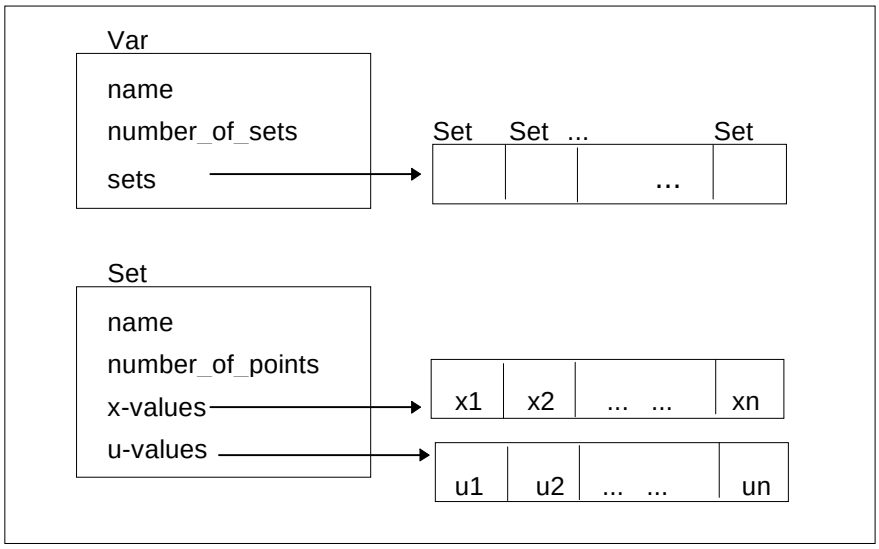

Figure 4: Data Structures for Variables and their Fuzzy Sets

Rule Bases

The rule base structure contains a set of rules together with information about the inputs and outputs of the particular rule base. Figure 5 shows the data structure for a rule-base.

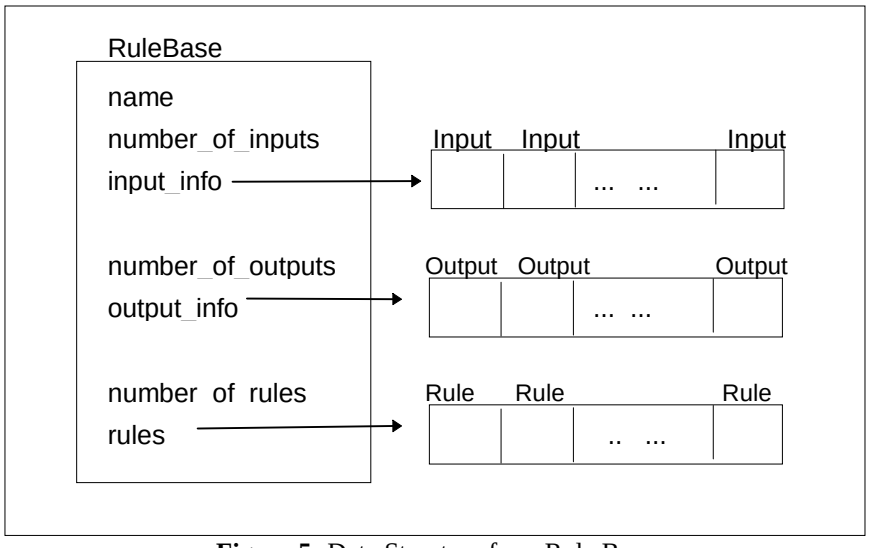

Figure 5: Data Structure for a Rule Base

Inputs and Outputs. The Input and Output structures are associated with the rule-base for convenience of the engine's operations, namely, fuzzification, inference and defuzzification. They are illustrated in Figure 6.

Each input and output has a 'var_id' to associate them with a linguistic variable in the variable table. The Input and Output structures provide a convenient way to set up the inputs and prepare for defuzzification. Before inference, the crisp inputs are filled into the Input structure. The set_areas and set_moments contains the pre-computed area and moment for each fuzzy set for that output variable. This is to facilitate certain de-

fuzzification methods discussed in section about Defuzzification. The 'fire_strengths' table stores the current clipping/scaling fire-strength for each fuzzy set, which will be used for defuzzification later.

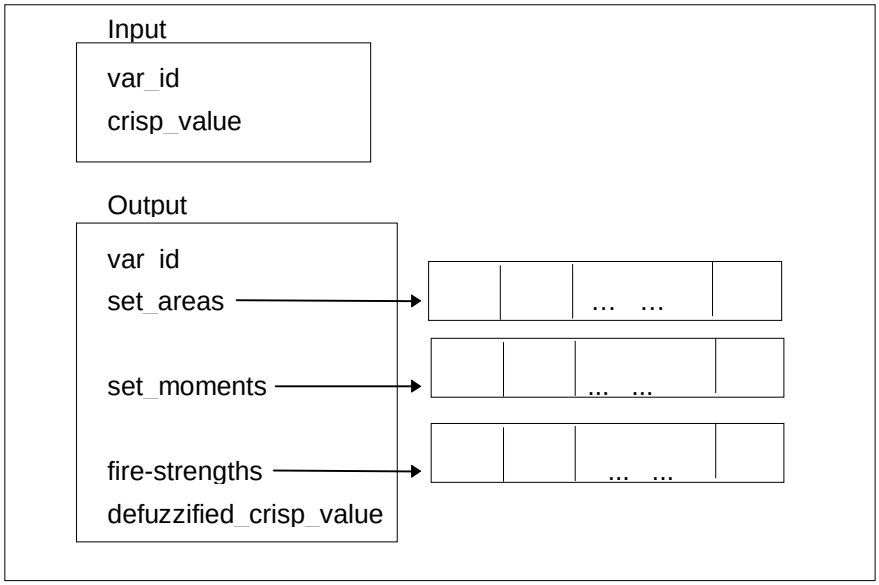

Figure 6: Data Structures for Input and Output

Rules. The most important information in a rule is, of course, its antecedent fuzzy proposition and its (list of) consequent(s). They are kept in a Rule structure shown in Figure 7.

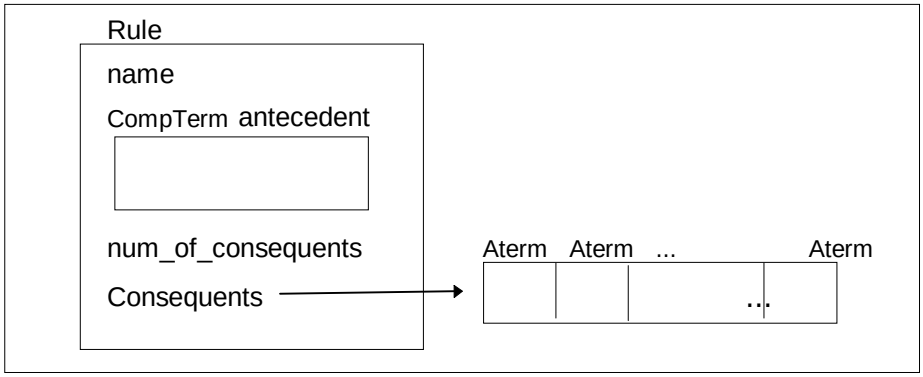

Figure 7: Data Structure for a Fuzzy Rule

The rule antecedent is stored in a structure CompTerm (composite term) used for compound fuzzy propositions. It is called a composite term because it contains a union that either stores an Aterm (atomic term) or a Nterm structure (N-term). An Aterm is an atomic fuzzy proposition, while a Nterm can be the conjunction (AND) or disjunction (OR) of two or more CompTerms (composite terms). These structures are illustrated in Figure 8. In an Aterm (A IS A1), the io_id refers to the entry in the input/output table which is 'linked' to the variable A. The set_id refers to the entry for set A1 in the sets table for A. In both Aterm and Nterm the is_inverted flag indicates whether the represented fuzzy proposition is complemented. Together, the three structures below allow any compound fuzzy propositions to be represented in a recursive way.

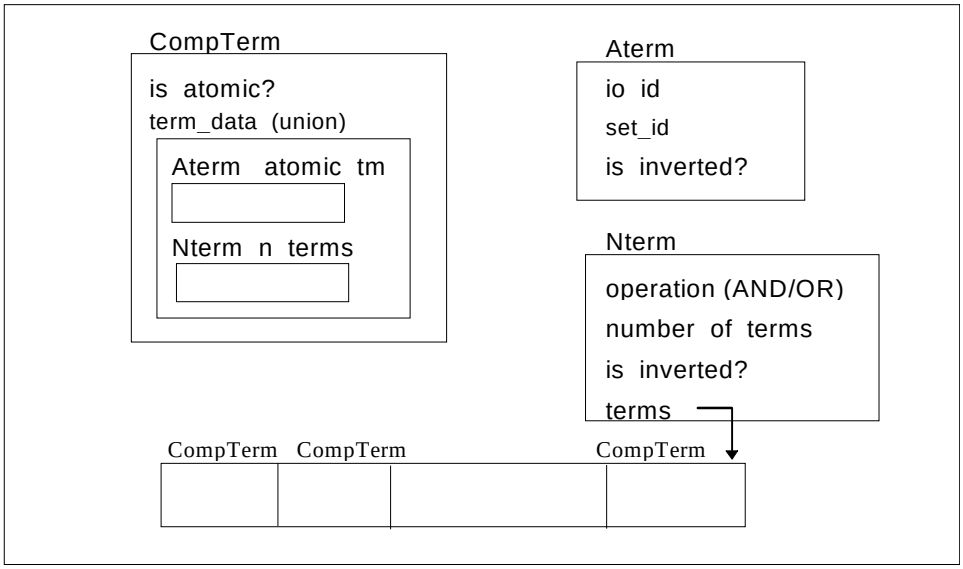

Figure 8: Data Structures for Compound Fuzzy Propositions

To summarize, the above structures hold the following information:

- CompTerm (composite term) holds either an Aterm or an Nterm.
- Aterm (atomic term) holds atomic fuzzy propositions like:
  (A is A1) (not inverted)
  (A is NOT A1) (inverted)
- Nterm (n-terms or a compound term) holds conjunctions or disjunctions of CompTerm's, e.g.:
  CompTerm AND CompTerm AND CompTerm ... (conjunction; not inverted)
  NOT(CompTerm OR CompTerm ...) (disjunction; inverted)

The recursive representation of a compound fuzzy proposition can be illustrated by a tree as shown in Figure 9. The statement represented can be:

(Degree IS VSmall) AND NOT((Distance IS VLarge) OR (Distance IS Large))
or
(Distance IS VSmall) AND ((Degree IS NOT VLarge) AND (Degree IS NOT Large))

or any other equivalent, e.g.:

(Degree IS VSmall) AND NOT NOT NOT((Distance IS VLarge) OR NOT(Distance IS NOT Large))
or
((Distance IS VSmall) AND (Degree IS NOT Vlarge)) AND (Degree IS NOT Large)

The parser developed for this knowledge base specification is able to reduce redundancies (such as un-necessary parentheses and consecutive NOT's which cancel out) to get the simplest equivalent form

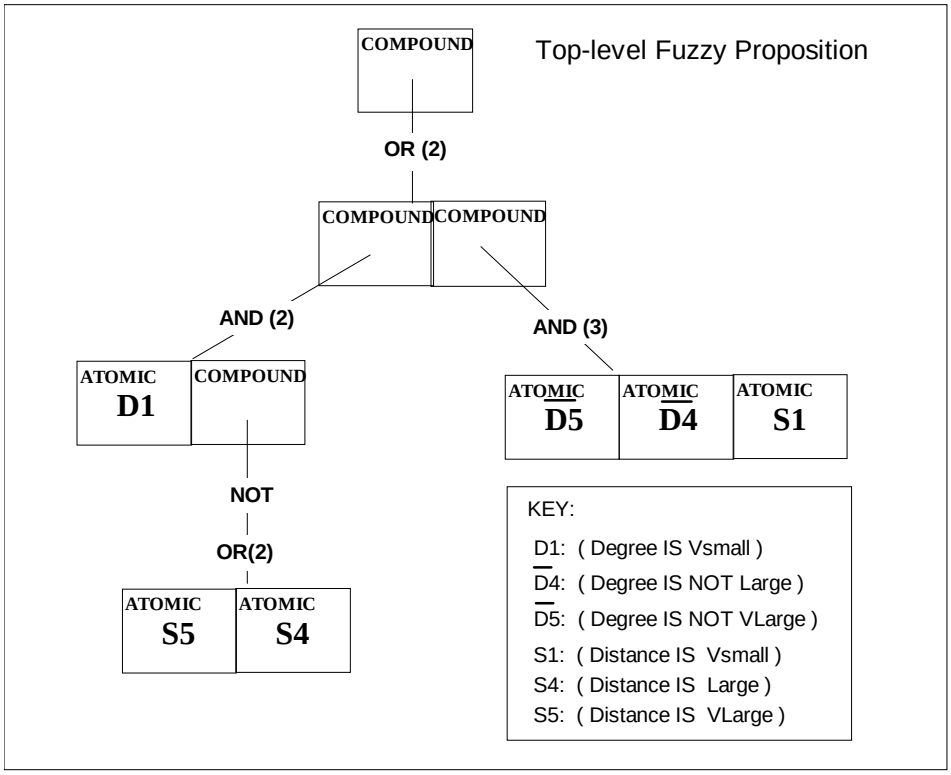

Figure 9: Tree representation for a compound fuzzy proposition

The way to traverse the tree is to use a recursive approach. The reasons for choosing such a representation are as follows:

- Statements with long lists of conjunctions or disjunctions can be evaluated very efficiently since traversing the whole tree and evaluating the truth value of all the atomic terms is not necessary. For conjunction, evaluation of an N-term returns (zero) as long as one of its component terms is evaluated to zero. For disjunction, evaluation of an N-term can stop as soon as 1 is reached.
- The representation almost mirrors the way it is actually laid down by a human expert, and thus it is unlikely to get inefficiently complex. Moreover, it is quite often and natural for the human expert to supply the statement as simple lists of conjunctions or disjunctions, which are handled by this representation very efficiently.

FUZZY KNOWLEDGE BASE PARSER

Need for a Knowledge Base Parser

The knowledge base is the like the brain of a fuzzy logic engine, without which no problem solving of any kind can be performed. It provides the engine with a set of rules together with the definition and meaning of linguistic variables and values used in specifying the rules.

The knowledge base must be written by a human user (expert). It is important to provide a convenient and natural way for him to specify the knowledge base. After all, a major attraction of fuzzy logic techniques is the ability to represent knowledge in linguistic or verbal forms. It would not be justified to let the user worry about the exact sequence and format in which the engine reads each item in the knowledge base and represent them internally. For example, the user should not have to pre-translate his logical expressions into the ideal format for the engine to build its internal tree representation. Also the user should be able to refer to linguistic variables and their values by their names (linguistic labels) rather than keeping track of which entries they are in the variable or set tables.

It is thus desirable to enable the user to write the knowledge base in some structured semi-natural language (English)-like statements, similar to statements in some high level programming languages. There is of course a very big difference and incompatibility between these statements and the engine's internal representations (data structures). A dedicated parser is thus needed to bridge this gap.

Parser as a Separate Tool

There are two choices regarding the implementation of the parser:

1. As part of the engine program

Each time the engine is executed, it first reads in the natural language like specification and parses it to build up the internal representations in the process. These internal representations are held in the main memory and used later for the inference processes.

2. As a separate tool

   The parser is a separate program used to translate the user's knowledge base (KB) specification into the necessary intermediate format (in a text file) for the engine program to read in efficiently and build its internal representations.

The authors decided to have the parser as a separate tool for the following reasons:

1. Parsing only needs to be done during the development stage of the KB, when the user changes his specifications. Thereafter, each time the engine is called (executed), it only has to load the KB from the intermediate specification file. There is no need for parsing which can be time-consuming.
2. The intermediate specification can be such that it allows the engine to load the KB efficiently and allocate the exact amount of memory required. For parsing, there is more overhead in memory usage, since the total amount of input items is not known while building the data-structures.
3. The intermediate specification can be seen as an encoded form of the KB since it will be unreadable for those who do not know the format. If the engine is used in an commercial application for example, only the 'encoded' KB files need to be distributed. The KB information will be hidden from the user.

Knowledge Base Specification Language

A suitable language for specifying the KB was designed. It is called a fuzzy knowledge base specification language (FKBSL). In this language, a KB specification always has the following 2 sections:

1. Declaration of variables and the membership functions of their fuzzy sets
2. Rule base specification which includes: (a) input and output declarations and (b) a set of fuzzy rules which relate the outputs with the inputs in linguistic terms.

Variable Declarations. A variable declaration must give all the information needed in order to use them in the fuzzy rules. The following shows an example of a variable declaration in the FKBSL:

```
VAR Degree
 SET VSmall POINTS (0.00,1.00) (22.50,0.00);
 SET Small POINTS (0.00,0.00) (22.50,1.00) (45.00,0.00);
 SET Ok POINTS (22.50,0.00) (45.00,1.00) (67.50,0.00);
 SET Large POINTS (45.00,0.00) (67.50,1.00) (90.00,0.00);
 SET VLarge POINTS (67.50,0.00) (90.00,1.00);
 END
```

Symbolic names are required for the variables and their fuzzy sets so that they can be referenced in the rules later. Figure 10 illustrates the declared fuzzy sets.

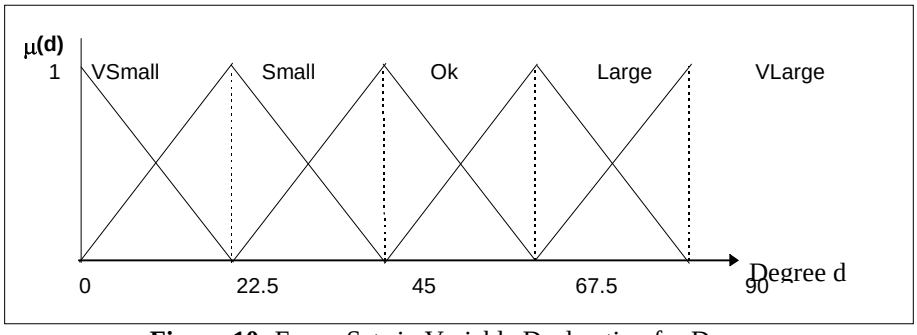

Figure 10: Fuzzy Sets in Variable Declaration for Degree

Rule Base Definitions. Once the variables and their fuzzy sets have been declared, they can be used in specifying the rules in the rule base. Figure 11 shows a complete sample knowledge base with one rule base defined. Each rule base definition has the following components:

1. A name for identification
2. Input and output declaration

This specifies which variables are inputs to the rulebase and which are outputs so that the parser can check the usage of variables in the rules later on. In the sample rule base, the variables Degree and

Distance are declared as inputs to the rulebase while Truth is declared as the output. These input and output variables must have already been declared in the variable declaration section. Also in the current implementation of the engine, the order in which the input variables are listed determines the order in which the various corresponding crisp values should be input to the engine for fuzzy inference.

3. A set of rules
   Each rule is a simple conditional statement of the form IF... THEN... The antecedent of the rule consist of a logical combination of atomic fuzzy propositions of the form (A IS A1) where A is the input variable name and A1 is the name of one of its fuzzy sets. The logical combination can be formed with the use of the connectives 'AND', 'OR' and 'NOT'. 'NOT' has the highest precedence and is right associative. 'AND' has a higher precedence than 'OR'. Together with the use of parentheses '(' and ')', expressions of any complexity is allowed. The consequent part of each rule consists of one or more results of the form (C IS C1) separated by 'ALSO'. C is an output variable and C1 is one of its fuzzy sets. An example is: IF ... THEN (C IS C1) ALSO (D IS D1);

```
// Sample Knowledge Base specification

VAR Degree      // Degree change
 SET VSmall POINTS (0.00,1.00) (22.50,0.00);
 SET Small POINTS (0.00,0.00) (22.50,1.00) (45.00,0.00);
 SET Ok POINTS (22.50,0.00) (45.00,1.00) (67.50,0.00);
 SET Large POINTS (45.00,0.00) (67.50,1.00) (90.00,0.00);
 SET VLarge POINTS (67.50,0.00) (90.00,1.00);
 END

VAR Distance    // Distance from original position
 SET VSmall POINTS (0.00,1.00) (25.00,0.00);
 SET Small POINTS (0.00,0.00) (25.00,1.00) (50.00,0.00);
 SET Ok POINTS (25.00,0.00) (50.00,1.00) (75.00,0.00);
 SET Large POINTS (50.00,0.00) (75.00,1.00) (100.00,0.00);
 SET VLarge POINTS (75.00,0.00) (100.00,1.00);
 END

VAR Truth       // Correctness of path indicated by new position
 SET VSmall POINTS (0.00,1.00) (25.00,0.00);
 SET Small POINTS (0.00,0.00) (25.00,1.00) (50.00,0.00);
 SET Ok POINTS (25.00,0.00) (50.00,1.00) (75.00,0.00);
 SET Large POINTS (50.00,0.00) (75.00,1.00) (100.00,0.00);
 SET VLarge POINTS (75.00,0.00) (100.00,1.00);
 END


RULEBASE Vehicle_Problem

INPUT Degree Distance;
OUTPUT Truth;

RULE VL1
 IF (Degree IS VSmall) AND
 NOT ( (Distance IS VLarge) OR (Distance IS Large) )

 OR (Distance IS VSmall) AND
 ( (Degree IS NOT VLarge) AND (Degree IS NOT Large) )

 THEN (Truth IS VLarge);

RULE VS1
 IF (Degree IS VLarge) AND (Distance IS VLarge)
 THEN (Truth IS VSmall);

END
```

Figure 11: Sample Knowledge Base Specification in FKBSL

The following points on syntax of the FKBSL should be noted:

- 'END' is used to mark the end of the variable declaration section as well as the end of each rule base specification.
- Comments can be added anywhere in the KB specification. The rest of a line after // is treated as comment.
- ';' is used to mark the end of a list of items, such as the list of points in set definitions, the list of variable names for input/output declarations and the list of consequent terms for each rule.

Writing the Parser

Introduction. For efficiency and maintainability, the KB parser is implemented using LEX and YACC. LEX and YACC are tools designed for writers of compilers and interpreters, although they have proved to be useful in many other applications involving the transformation of structured inputs. Any application that looks for patterns in its input, or has an input or command language is a good candidate for LEX and YACC. Furthermore, they allow for rapid application prototyping, easy modification, and simple maintenance of programs.

The analysis of a source program with structured input can be separated into two parts: lexical analysis and syntax analysis.

Lexical analysis is performed by a scanner. The scanner divides the input into small meaningful units known as tokens. Given a set of description of possible tokens, LEX is able to generate a scanner (in the form of a C routine) to identify these tokens.

Syntax analysis is carried out by a parser. The parser takes in a stream of tokens and tries to establish the relationship among these tokens. The list of rules defining this relationship defines the grammar of the language. YACC takes a concise description of a grammar and produces a parser (also a C routine) which automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and detects syntax errors whenever the input doesn't match any of the rules.

Lexical Analysis and LEX. The purpose of lexical analysis or scanning is basically to 'tidy up' the source program (KB specification in this case) and prepare it for parsing. At this stage comments, spaces, tabs and any other unimportant detail from the source program can be eliminated. Also, at this stage the program can checked if it consists only of variable names, keywords, numbers and punctuation symbols. Each of these identified entity is a token and has a token number to identify its token type.

The LEX tool has been used to generate the scanner for the designed language (FKBSL). LEX takes as input specifications a series of character patterns to be matched and the corresponding action (written in C code) to be performed on matching the pattern. Usually the action is to return a token number to the parser (which calls a scanner internally).

Syntax Analysis and YACC. In syntax analysis or parsing the input is a stream of tokens whose relationship to one another must conform to a defined grammar. The grammar definition in YACC consists of a list of production rules, each of which may have an associated action (again, just a piece of C code). Figure 12 shows a list of production rules for a variable declaration (definition). All of the names in upper case represent tokens which are expected from the scanner; all of the names in lower case will be defined in production rules later on. The left hand side of a rule states what is being defined and the right side is the definition. For example, the first rule in Figure 12 states that in order to recognize a variable definition (var_def), the parser must first receive the tokens VAR and NAME. Then it goes on to recognize a set definition list (set_def_list) and finally the token END which completes the variable definition.

```
/* variable definition */
var_def : VAR NAME set_def_list END
 ;

/* set definition list */
set_def_list : set_def /* start of a list */
 | set_def_list set_def
 ;

/* set definition */
set_def : SET NAME POINTS pt_list ';'
 ;

/* point list */
pt_list : point
 | pt_list point
 ;

/* point definition */
point : '(' NUMBER ',' NUMBER ')'
 ;
```

Figure 12: Production Rules for a Variable Declaration

Some C codes can be put on the right-hand-side of the rule. These will be executed whenever the parser has recognized everything immediately to the left of it. For example:

```
A       :       B       C       { action1 }     D       { action2 }
        |       E       F       G { action3 }
        ;
```

Notice that action1 is embedded in the middle of the rule which allows the programmer to gain control before the rule is fully parsed. The vertical bar '|' separates two or more which have the same left-hand-side. These C code actions are placed where semantic checking (e.g. checking the variable names to eliminate duplicate declarations) is done and the internal representations of this knowledge base is built up. In this C code the markers $1, $2, etc. can be used to refer to the attributes (data/value) associated with the symbols of the right of each rule. $$ refers to the attribute associated with the symbol on the left-hand-side of the rule.

Parser Operations

The parser goes through the KB source (written in a KB specification language) only once, building up its own internal representation of the KB as it goes along. At the end of the parse (if there are no syntax errors), it writes the entire KB to an intermediate file in the exact format to be read by the engine.

The parser is able to report syntax errors with some helpful information such as line and column number of the error. Semantic errors such as using an undeclared variable names, undeclared inputs/outputs and duplicate declarations can also be reported. Having completed parsing, the parser also performs some check on the integrity of the fuzzy set definitions.

The data-structures used in the KB parser program (to hold the knowledge base) are analogous to those discussed in the chapter about Data Structures for the fuzzy logic engine. There is just some difference in that, in quite a few places, pointers to structures are used instead of the structures directly. This is to facilitate the dynamic construction of the data representations and the passing of parameters via YACC's value stack for this purpose.

Processing Variable Declarations. The principal activity in this stage of parsing would be to store the variables and its fuzzy sets in tables and keep a count of the number of entities like variables, fuzzy sets for each variable, number of points in each set. When writing the intermediate file for the engine, these numbers would be supplied first to let the engine know how many items to expect and so on.

Processing Rule Base Definitions. When processing the rule-base, every variable or set name has to be checked against those stored in the Variables table, to make sure that they have been declared. Also, variables found in the rule antecedent must have been declared as INPUT. Similarly variables found in the consequent must have been declared as OUTPUT. Upon parsing the input/output declarations, an Inputs table and an Outputs table is created. These store information relating each input/output to a variable in the Variables table.

In representing a fuzzy proposition (VarX is SetY), the index into the Inputs/Outputs table corresponding to VarX is used, rather than using the name directly. Also, a set index into the Sets table will be used to represent SetY as the proposition. These measures facilitate the operations of the fuzzy inference engine. Thus an important job of the parser is mapping names of various entities to appropriate numerical indices into various tables.

Parsing the Rule Antecedent. One of the most challenging tasks (in the author's opinion) in developing the general fuzzy logic engine involves the fuzzy rules (particularly the rule antecedents). This includes:

- devising suitable data structures for them (as discussed in the section Rule Base) to be manipulated easily.
- working out algorithms to translate the rules given in structured verbal statements into their internal representations.
- writing the fuzzy rule in an intermediate file (as texts) for the engine to read and build up this internal representation directly.

The production rules (in YACC specifications) for parsing a rule antecedent is shown in Figure 13. At the top level, the rule antecedent (ante_term) is represented as composite term (CompTerm). It can be one of the following:

- or_list (Nterm: a number of CompTerm's 'OR' together)
- and_list (Nterm: a number of CompTerm's 'AND' together)
- aterm (Aterm: an atomic fuzzy proposition)

```
%left OR   /* left associative, lowest precedence */
%left AND  /* left associative, higher precedence than OR */
%right NOT /* right associative, highest precedence */

%%

/* antecedent a rule */
ante_term : or_list       /* R-C1 */
 | and_list       /* R-C2 */
 | aterm          /* R-C3 */
 ;


/* compound term (proposition) consisting a list of terms OR together */
or_list : aterm OR aterm          /* R-O1 */
 | or_list OR aterm               /* R-O2 */
 | aterm OR or_list               /* R-O3 */
 | or_list OR or_list             /* R-O4 */
 | and_list OR aterm              /* R-05 */
 | aterm OR and_list              /* R-O6 */
 | or_list OR and_list            /* R-O7 */
 | and_list OR or_list            /* R-O8 */
 | and_list OR and_list           /* R-O9 */
 | '(' or_list ')'                /* R-O10 */
 | NOT or_list                    /* R-O11 */
 ;


/* compound term (proposition) consisting a list of terms AND together */
and_list : aterm AND aterm        /* R-A1 */
 | or_list AND aterm      /* R-A2 */
 | aterm AND or_list      /* R-A3 */
 | or_list AND or_list            /* R-A4 */
 | and_list AND aterm     /* R-A5 */
 | aterm AND and_list     /* R-A6 */
 | or_list AND and_list   /* R-A7 */
 | and_list AND or_list   /* R-A8 */
 | and_list AND and_list  /* R-A9 */
 | '(' and_list ')'       /* R-A10 */
 | NOT and_list                   /* R-A11 */
 ;


/* function to modify a set definition, only 'NOT' is available here */
set_modifier : /*empty*/
 | NOT
 ;


/* an atomic fuzzy proposition */
aterm : '(' NAME IS set_modifier NAME ')'  /* R-T1 */
 | NOT aterm                              /* R-T2 */
 ;
```

Figure 13: Production Rules for a Fuzzy Rule Antecedent

The YACC generated parser reads in the tokens from left to right and recognizes the above items (aterm, or_list, and_list) as relevant production rules to be matched and reduced. It is instructive to follow through the sequence with which reduction takes place for the sample rule antecedent given in Table 6.1. Each entry shows the constructs recognized by the parser for the tokens up to the position indicated by the corresponding arrow. '[EOT]' represents the end of input for the rule antecedent.

RULE	(A IS A1) OR (A IS A2) OR (A IS A3) AND (B IS NOT B1) AND NOT NOT ((B IS B2) AND NOT (B IS B3)) [EOT]
R-T1	aterm
R-T1	aterm OR aterm
R-O1	or_list OR
R-T1	or_list OR aterm
R-T1	or_list OR aterm AND aterm
R-A1	or_list OR and_list
R-T1	or_list OR and_list AND NOT NOT (aterm
R-T1	or_list OR and_list AND NOT NOT (aterm AND NOT aterm
R-T2	or_list OR and_list AND NOT NOT (aterm AND aterm
R-A1	or_list OR and_list AND NOT NOT (and_list
R-A10	or_list OR and_list AND NOT NOT and_list
R-A11	or_list OR and_list AND NOT and_list
R-A11	or_list OR and_list AND and_list
R-A9	or_list OR and_list
R-O7	or_list
R-C1	ante_term

Table 1: Parser Reduction Sequence for a Fuzzy Rule Antecedent

As each production rule is matched (and reduced), an associated action is executed. These actions are for processing the matched symbols and building up the internal tree representation of the rule antecedent.

Tree Construction. Each element of aterm, or_list and and_list is treated uniformly as a composite term (CompTerm) and represents a node in the tree. An aterm is considered a leaf node (without child nodes) while an or_list or and_list is an internal node with a list of child nodes. Depending on the constructs recognized, the tree-building actions can be of three types:

- **Parent_Node():**
relate two nodes by constructing a parent node for them
- **Concat_Lists():**
merge two 'list-nodes' (or_list / and_list) into one by concatenating the two lists.
- **Add_Term():**
merge two nodes n1 and n2 (where at least n1 is a 'list-node) by adding n2 to n1's list of child nodes.

Figure 14 illustrates the above three operations and lists the production rules required by each tree construction operation. The discussion thus far is with respect to the formation of or_list's but analogous results are obtained for and_list's simply by replacing OR and or_list with AND and and_list respectively. The algorithm was developed with the aim of keeping the tree height to a minimum for efficiency in storage and processing.

With an understanding of how the various terms in the rule antecedent are combined into an equivalent tree representation, it should be obvious that extra parentheses (for grouping purposes) which do not actually affect the meaning of the fuzzy proposition would not lead to a different tree representation. For example the two fuzzy propositions below would be reduced to the same equivalent form.

- (A IS A1) OR ((A IS A2) OR (A IS A3)) OR ((A IS A4) AND (A IS A5))
- (A IS A1) OR (A IS A2) OR (A IS A3) OR (A IS A4) AND (A IS A5)

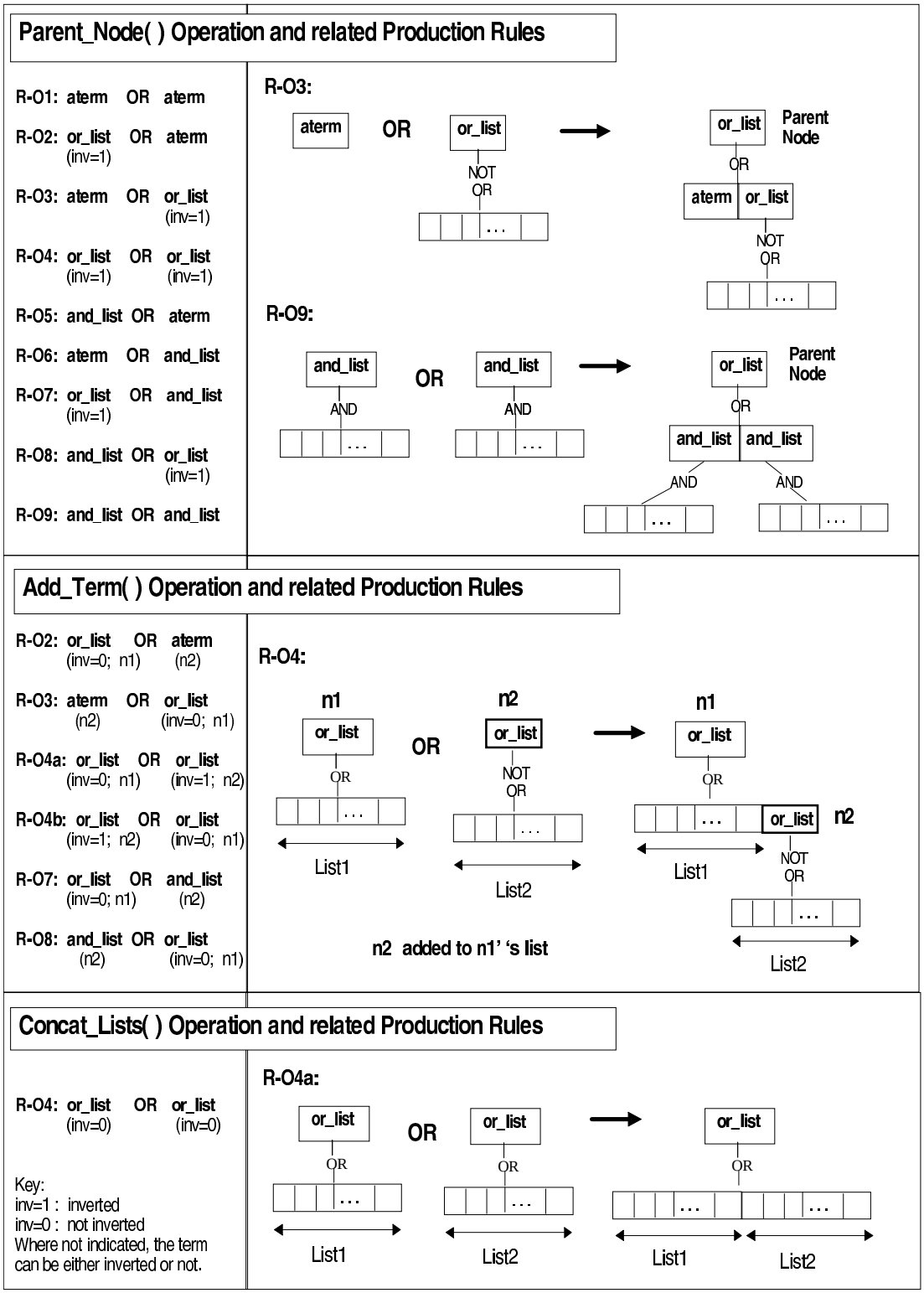

Figure 14: Tree Construction Operations and Associated Production Rules

Consecutive NOT's resulting in redundancy can also be resolved. As each NOT construct is recognized, the 'inv' flag associated with each term is set/toggled appropriately. In the final representation, and hence in subsequent processing, these redundancies will not be reflected.

Writing the Rulebase in Formatted Form. Having parsed the entire rulebase and built up the internal representations, the rulebase then is written to an intermediate KB file (for the engine) in a format compatible to the internal representations. This enables the engine to load the KB efficiently into its internal data structures.

CONCLUSIONS

Upon studying the requirements of the BSS1 tutoring system, the approximate reasoning technique used in most fuzzy controllers was selected. The general fuzzy logic engine is able to read in a knowledge base at runtime and use this knowledge to perform inference on input data. A fuzzy knowledge base specification language (FKBSL) is designed to let the user of the engine define the knowledge base in a structured but reasonably verbal way. A knowledge base parser is developed for use together with the fuzzy logic engine. The parser translates the knowledge base in the FKBSL into an intermediate format to be used directly by the engine.

There is not just one unique way to approximate reasoning. Even for the generalized modus ponens which is the data driven approach selected, there exists a wide choice of internal operation parameters such as fuzzy implication functions, fuzzy set operators, composition operators and defuzzification techniques. The choices made are those most commonly seen in the literature for fuzzy control, which have been the most successful application area for fuzzy systems. The methodology used in fuzzy logic controllers (which are special expert systems) is suitable for this purpose (use in intelligent tutoring systems). The intelligent agent can be considered as a tool which helps to manage the student's learning as an expert system, and emulates the reasoning process of a human expert (in this case a teacher) within a specific domain of knowledge (such as tutoring strategies).

The usefulness of such an expert system depends on the engineer's ability to model the problem suitably, define fuzzy variables and suitable membership functions for their fuzzy sets, and develop a comprehensive set of rules relating input and output variables. This is not an easy task and usually requires considerable fine-tuning of the knowledge base through observing simulation results of the system. For simple systems at least, the engineer would not need to explore alternatives for internal operation parameters for the inference engine. A thorough understanding of some of these parameters (in terms of the effects of various proposed alternatives in the literature) requires extensive experimental studies or a sound foundation in fuzzy set and fuzzy logic theories. The average engineer who may not be equipped with this knowledge is still able to design (simple) systems by just manipulating the fuzzy set functions and the rules. This is why the authors think it is not important to provide the user of the engine (the expert system designer) with a choice in these internal parameters. Instead the authors fixed these parameters by choosing widely used methods which have proven effective and commonly appear in the literature.

Acknowledgments

We would like to thank Rajat Kumar Das, Managing Director of Brilliant Systems Pte. Ltd. and Prof. Wang Peizhang, formerly Research Staff of the Institute of Systems Science (ISS) for coming up with this approach to student monitoring. Further we would like to thank Joel Loo Peing Ling, Research Staff at ISS for his encouraging support.

References

Driankov D., Hellendoorn H. and Reinfrank M. [1993], An Introduction to Fuzzy Control ,Springer-Verlag Berlin Heidelberg

Klir G. J. and Yuan B. [1995], Fuzzy Sets and Fuzzy Logic - Theory and Applications, Prentice-Hall International, Inc.

Yan J., Ryan M. and Power J. [1994], Using Fuzzy Logic, Prentice Hall

APPENDIX A Knowledge Base in FKBSL

```
// Knowledge Base for Ranking Topic for Study

// Topics with high rank are more highly recommended for study
// Ranking Based On Competence and TimeSpent.
// General Rationale:
// The less the competence & the less the time spent, the higher
// the rank. However, if student has already spent much time,
// yet competence is low, then no point carrying on at the moment.
// Also, if the competence is already very high, don't need to
// work on it anymore, even if time-spent is not much.


// Each Variable takes values from 0 to 100

VAR Competence // Student's present competence in the topic
 SET VSmall POINTS (0.00,1.00) (25.00,0.00);
 SET Small POINTS (0.00,0.00) (25.00,1.00) (50.00,0.00);
 SET Ok POINTS (25.00,0.00) (50.00,1.00) (75.00,0.00);
 SET Large POINTS (50.00,0.00) (75.00,1.00) (100.00,0.00);
 SET VLarge POINTS (75.00,0.00) (100.00,1.00);
 END

VAR TimeSpent // time spent in topic
 SET VSmall POINTS (0.00,1.00) (25.00,0.00);
 SET Small POINTS (0.00,0.00) (25.00,1.00) (50.00,0.00);
 SET Ok POINTS (25.00,0.00) (50.00,1.00) (75.00,0.00);
 SET Large POINTS (50.00,0.00) (75.00,1.00) (100.00,0.00);
 SET VLarge POINTS (75.00,0.00) (100.00,1.00);
 END


VAR Rank /* higher rank, better choice to study */
 SET VSmall POINTS (0.00,1.00) (25.00,0.00);
 SET Small POINTS (0.00,0.00) (25.00,1.00) (50.00,0.00);
 SET Ok POINTS (25.00,0.00) (50.00,1.00) (75.00,0.00);
 SET Large POINTS (50.00,0.00) (75.00,1.00) (100.00,0.00);
 SET VLarge POINTS (75.00,0.00) (100.00,1.00);
 END
```

```
RULEBASE Rank_Topic


INPUT Competence TimeSpent;
OUTPUT Rank;


RULE VL000
 IF (Competence IS VSmall) AND (TimeSpent IS VSmall)
 THEN (Rank IS VLarge);


RULE L000
 IF (Competence IS Small) AND (TimeSpent IS Small)
 THEN (Rank IS Large);


RULE Ok000
 IF (Competence IS Ok) AND (TimeSpent IS Ok)
 THEN (Rank IS Ok);


RULE S000
 IF (Competence IS Large) AND (TimeSpent IS Large)
 THEN (Rank IS Small);


RULE S001
 IF ( (Competence IS NOT Large) AND (Competence IS NOT VLarge) )
 AND (TimeSpent IS VLarge)
 THEN (Rank IS Small);



RULE VS000
 IF (Competence IS VLarge) AND (TimeSpent IS VLarge)
 THEN (Rank IS VSmall);


RULE VS001
 IF (Competence IS VLarge)
 THEN (Rank IS VSmall);


RULE VS002
 IF (Competence IS VSmall) AND (TimeSpent IS VLarge)
 THEN (Rank IS VSmall);


END
```

APPENDIX B Results for Sample Input Data

Competence	TimeSpent	Expected Rank Value	Inferred Rank	Rationale for Expected Value
0	0	90	Rank2 = 91.666672 Rank1 = 91.708336	
10	10	80	Rank2 = 82.142860 Rank1 = 79.465675	
20	20	70	Rank2 = 76.851852 Rank1 = 75.501274	
30	30	60	Rank2 = 70.000000 Rank1 = 68.966118	
40	40	55	Rank2 = 59.999996 Rank1 = 60.483376	
50	50	50	Rank2 = 50.000000 Rank1 = 50.000000	
60	60	40	Rank2 = 40.000000 Rank1 = 39.516624	
70	70	30	Rank2 = 30.000000 Rank1 = 31.033880	
80	80	20	Rank2 = 23.148148 Rank1 = 24.498726	
90	90	10	Rank2 = 17.857143 Rank1 = 20.534325	
100	100	10	Rank2 = 8.333333 Rank1 = 8.291667	
30	30	70	Rank2 = 70.000000 Rank1 = 68.966118	
30	40	65	Rank2 = 66.666664 Rank1 = 66.667259	
30	50	60	Rank2 = 50.000000 Rank1 = 50.000000	
30	60	55	Rank2 = 50.000000 Rank1 = 50.000000	
30	70	40	Rank2 = 50.000000 Rank1 = 50.000000	
30	80	30	Rank2 = 25.000000 Rank1 = 25.000000	
30	90	25	Rank2 = 24.999998 Rank1 = 25.000000	

As Competence & Time-Spent increases, Rank Drops

If Competence stays Low while TimeSpent becomes high, rank should be lowered.

Competence	TimeSpent	Expected Rank Value	Inferred Rank	Rationale for Expected Value
30	20	70	Rank2 = 75.000000 Rank1 = 75.000000	
40	20	65	Rank2 = 75.000000 Rank1 = 75.000000	
50	20	60	Rank2 = Null! Rank1 = Null!	TimeSpent maybe low, but if competence becomes high, rank can be lowered.
60	20	50	Rank2 = Null! Rank1 = Null!	
70	20	40	Rank2 = Null! Rank1 = Null!	
80	20	30	Rank2 = 8.333334 Rank1 = 11.264889	
90	20	20	Rank2 = 8.333333 Rank1 = 9.252521	

For each pair of input values (for Competence and TimeSpent), the expected value for Rank is first written down by a human observer (the author), based on intuitive understanding of the problem. The same understanding is used by the author in drafting the prototype knowledge base in Appendix A. The expected values are then compared with the corresponding ranks inferred by the fuzzy logic engine. Two methods of defuzzification has been implemented in the engine. Rank1 is obtained by method1: Centre of Area (COA) method. Rank2 is obtained by method2: approximation to COA by summing scaled fuzzy sets. It can be seen from the above data that the two methods give quite close results.

The inferred results generally follow the expected trend, although for certain combinations of input values, the inferred Rank is ‘null’. This means that none of the rules has a fire-strength greater than zero. Effectively, no rule can be fired and thus no inference can be made. Indication of ‘null’ outputs is useful to help us track combinations of input values neglected by the rules in the knowledge base. Appropriate rules can be subsequently added or modified to fine-tune the inference engine for solving the particular problem. Fuzzy set definitions for the various variable can also be adjusted to alter the engine behavior. Generally, the more the overlap between fuzzy sets of a variable, the less the number of rules needed to cover all possible combinations of input values.