RICE

version 4.0

Routines for Implementation of C Expert systems

6 September 2000

René Jager

R.Jager@ET.TUdelft.NL

Delft University of Technology
Department of Electrical Engineering
Control Laboratory

Mekelweg 4
P.O. Box 5031
2600 GA Delft
The Netherlands

Preface

RICE stands for *Routines for Implemention of C Expert systems*. The general idea of the software is to provide a tool for easy implementing small, but powerfull, (fuzzy) expert systems within C or C++ programs. The software is supposed to take away all worries about the inference mechanism: the user provides the knowledge on a high level, but is still able to call plain C-code (or C++) from within the knowledge base. This means in fact that you can build a program or a part of a program in which the execution order of plain C-code is completely determined by the knowledge base which is inferred.

By the way, the inference engine is not the hardest part to make when building a (fuzzy) expert system: designing a knowledge base that makes some sense is the hardest part. But that problem is completely for you to solve...

Contents

2 Introduction

In this introduction the general properties of expert systems, logic and reasoning will be discussed. Both propositional logic and fuzzy logic will be explained as well as several common reasoning methods. How these things are implemented in RICE is not part of this introduction, but the chapters after this.

2.2 Seperation of knowledge and inference

Typical of experts systems is the seperation of knowledge and inference. Unlike other computer programs an expert system has a predefined part, namely the inference engine, and a variable part, namely the knowledge base. In fact an expert system can be described by:

expert system = knowledge base + inference engine

The following two sections will describe the knowledge base and inference engine respectivily.

2.2.2 The knowledge base

The knowledge base of an expert system contains knowledge of a specific domain. For example, an expert system for medical diagnosis contains expert knowledge for a set of deseases. The knowledge is normally stored in the knowledge base using a symbolic representation, for example:

if there is smoke then there is fire

This representation is known as a production rule, or rule for short. Other representation are frames and objects. The main advantage of having a seperation between the knowledge base and the inference engine is the fact that an iterative development of the expert system is possible in a more natural way than in case of normal computer programs. The production rule is specificaly suitable for implementing heuristic knowledge.

It is possible that the knowledge base contains knowledge about knowledge within itself; this is called meta-knowledge. Examples of meta-knowledge are rules which, when fired, will focus on a certain part of the knowledge base (focus of attention). Also possible is knowledge which adapts other knowledge within the knowledge base, for example, a rule which changes the degree of certainty of another rule.

The knowledge base is filled with rules like the one in the example above and the inference engine will acts on this knowledge base to obtain new information. The next section will go into more details of the inference engine.

2.2.4 Inference engine

The inference engine is the part of the expert system that is not changed by expert system developers. It is an already provided part of an expert system shell or toolbox. The main task of the inference engine is inferring new information using the knowledge base and (already) stored information. When we take the example from the previous section, the inference engine is able to infer the fact *there is fire* when the fact there is *smoke is true*, using the rule from the example.

To be able to obtain new information using existing information and knowledge from the knowledge base, the inference engine needs logic, to be able to perform the task in a logical sense, and reasoning methods to use the knowledge base in a consisting way. Both aspects will be discussed in the following sections.

2.4 Propositional and fuzzy logic

Normally rule-based expert systems use propositional logic. Within porpositional logic facts are either true (1) or false (0). To represent knowledge using propositional logic, five connectives are available:

- negation: ¬
- conjunction: v
- disjunction: w
- implication: Y
- bi-implication:]

The precendence of these connectives is: ¬, v, w, Y and]. Normally the inference engine is able to deal with if-then rules, but they can be translated in the connectives of propositional logic. In the following table one can see the result of propositions:

A	B	¬A	AvB	AwB	AYB	A]B
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0

1 1 0 1 1 1 1

For using propositional logic several laws are available: commutativity, associativity, distributivity and De Morgan laws. The next table shows those laws:

¬(¬A)	/	A
AvB	/	BvA
AwB	/	BwA
(AvB)vC	/	Av(BvC)
(AwB)wC	/	Aw(BwC)
Aw(BvC)	/	(AwB)v(AwC)
Av(BwC)	/	(AvB)w(AvC)
A]B	/	(AYB)v(AYB)
AYB	/	¬AvB
¬(AvB)	/	¬Av¬B
¬(AvB)	/	¬Av¬B

However, only dealing with facts that are either true or false is normally not sufficient. To overcome this drawback an inference engine should be able to deal with fuzzy logic. This allows facts to be true, false or having some degree of truth in between. This results in an infinite number of possible operators for the logical and and logical or. Commonly used within the field of fuzzy logic are:

avb : min(a, b)
awb : max(a, b)

The and and or operations are called T-norm and S-norm (or T-conorm), respectivily. In fact every function of two variables can be used as a fuzzy *and*-operation (T-norm) as long as the following criteria are met:

T(a, 1) = a
T(a, b) # T(c, d), whenever a # c and b # d
T(a, b) = T(b, a)
T(T(a, b), c) = T(a, T(b, c))

Examples of such T-norms are: *min(a, b)*, *ab* and *max(a+b-1, 0)*. For the S-norm the same criteria, except for the first, should be met:

S(a, 0) = a
S(a, b) # S(c, d), whenever a # c and b # d
S(a, b) = S(b, a)

S(T(a, b), c) = S(a, S(b, c))

Examples of S-norms are: *max(a, b)*, *a+b-ab* and *min(a+b, 1)*. Many families of T- and S-norms exist, where a parameter is used to alter the behaviour (but still meeting the criteria) of the operator/norm.

2.6 Modus ponens and modus tollens

In the field of logic two main ways of inferring information using existing information and an in implication (knowledge): the modus ponens and the modus tollens. The modus ponens is in classical (true-false) logic defined as:

A	AYB	B
1	1	1
0	1	?

So if we have a rule *if there is smoke then there is fire* we can concude *there is fire* when we know *there is smoke*. However, we cannot conclude *there is no fire* in case *there is no smoke*. The modus tollens is defined by:

B	AYB	A
1	1	?
0	1	0

The modus tollens provides a mechanism to let us conclude, using the previous example, that *there is no smoke* when we know *there is no fire*.

However, when using fuzzy logic there is no clear seperation between completely true or false. In many applications of fuzzy logic in (small) expert systems there will be no seperation. This results in concluding *there is no fire*, when knowing *there is no smoke*. Using fuzzy logic the rule from the above mentioned example would more act like: *if there is smoke then there is fire else there is no fire*. Such an implication is in fact an relational implication: a sort of bi-implication with prescribed direction of inference. For example, in fuzzy control relational implications are used to describe the control rules.

2.8 Reasoning and search methods

In the design of expert systems several reasoning and search methods can be chosen. It depends on the application which methods and techniques are used. More detailed information about the design of expert systems can be found in [Luger and Stubblefield 1989].

2.8.2 Standard methods

In expert systems two main reasoning methods can be found: forward and backward reasoning. Forward reasoning, also known as data-driven or bottom-up reasoning or chaining, uses initial data and infers new data from the known data. The inference of new data is done by applying rules from the rule base. The inference of new knowledge stops in case some predefined goal is reached or no new knowledge can be infered. Backward reasoning, also called goal-driven or top-down reasoning or chaining, starts with certain goals and in order to solve (read: prove) these goals their subgoals are tried to be solved. This mechanism goes on until no subgoals are present or the initial goals are solved. The combination of forward and backward reasoning in one inference engine within an expert system is possible, but not found often in literature.

Additional to the above two reasoning methods one can distinguish, besides the simple exhaustive search method, three main search methods: depth-first, breadth-first and heuristic search. The depth-first search method is normally used in combination with backward reasoning, the breadth-first method with forward reasoning. Whenever it is possible, depth-first search goes deeper into the search tree. The search stops in case no lower levels within the serach tree exist or a stop criterium is met. The breadth-first search method however, searches level by level within the search tree. The exploration of the next level of a search tree is started in case in the current level no more states can be solved or a certain stop criterium is met. Combining depth-first and breadth-first is possible, but, just like the combination of backward and forward reasoning, not found often in literature. Heuristic search is applied to guide the search within the search tree and therefore can be seen as meta-knowledge: knowledge about knowledge.
The best-first search method is an example of heuristic search. In this search method the 'most promising' branch of a search tree is chosen for further search. Confidence factors can be used to indicate which branch is more promising to lead to a satisfying result than others.

2.8.4 Real-time issues

A real-time behaviour is often easier to recognize than to define. As discussed in [O'Reilly 19??] many definitions of real time exist. Real time is mostly related to 'fast': meaning that a system processes data quickly. A formal definition of real time is offered:

> *a hard real-time system is defined as a system in which correctness of the system not only depends on the logical results of a computation, but also on the time at which the results are produced.*

The most important item is the response time, if events are not handled timely, the process can get out of control. Thus, the feature that defines a real-time system is the system's ability to guarantee a response before a certain time has elapsed, where that time related to the dynamic behaviour of the system. If, given an arbitrary event or state of the system, the system always produces a response by the time it is needed, then the system is said to be real-time. Due to the real-time aspect the fuzzy expert system should be able to perform non-monotonic, temporal and progressive reasoning. Non-monotonic reasoning introduces the ability to change already proven data and all the infered data that depend on it. Assumptions which later on in the reasoning process appear to be erroneous can be changed and all the data which depend those assumptions must be reexamined and possibly withdrawn from the data base. The reexamination of the data base is done by a truth-maintenance system [Doyle 1979]. Non-monotonic reasoning is more like the reasoning of humans than monotonic reasoning is, in which previously added knowledge or data can not be reexamined.

In case the expert system deals with past, present and future, it has to be able to outdate previous and accept new data as the time window connected to the present time is moving. This principle is called temporal reasoning [Krijgsman et al. 1991] and an essential aspect of real-time expert systems.

Also an important aspect of the real-time expert systems is the progressive reasoning principle [Lattimer et al. 1986]. In case of progressive reasoning the rule base is divived in several layers, which are in fact also rule bases. The inference engine starts with the inference of the first layer and proceeds with the next (higher) layer in case the inference of the first layer is finished. Each layer is able to use the data inferred by the lower layers as well as the rules within that layer. In case the real-time environment aborts the reasoning by means of an interrupt, the data infered by the last layer which completed its inference can be used. In this way the more time available, the 'better' the conclusions get. This aspect has also close resemblance to human reasoning.

2.10 Fuzzy expert systems

The theoretical approach of fuzzy inference is based on the idea of relational implications rather than on the idea of classical implications. A fuzzy controller is a function of more than one variable, which is a combination (union) of the functions representing the individual fuzzy rules. The 'direction' of inference is dictated by the fuzzy rules. A typical rule *i* can be stated as:

IF x_1 is X_1^i AND ... AND x_j is X_j^i AND ... AND x_N is X_N^i THEN y is Y^i

The complete fuzzy controller, however, is a multi-dimensional relation, combining all individual fuzzy rules:

$$R = \bigvee R_i$$

where the membership of every relation describing a fuzzy rule is given by:

$$R_i = X_1^i(x_1) \vee ... \vee X_j^i(x_j) \vee ... \vee X_N^i(x_N) \vee Y^i(y)$$

As can be seen from these equations the resulting fuzzy system has no longer a *direction* like in rules (*if ... then ...*): it is a relation. This is why, for example, fuzzy models can be used for predictions as well as *inverse* control [Brown et al. 1991]. By applying the compositional-rule-of-inference:

$$Y = X \circ R$$

and using a fuzzification of the inputs, which results in fact also in a relation, a fuzzy output is obtained. The most commonly used inference rule is the *max-min*-rule. One could also denote the compositional-rule-of-inference as an OR-AND operation, which is a more general interpretation. In practical applications however, usually a more efficient method is used [Harris and Moore 1989]. This method is based on the assumption that the fuzzification of an input is represented by a singleton. Applying the compositional-rule-of-inference on the relation, using the fuzzified input, results in a fuzzy output. The fuzzified input is represented by the union of all the singletons describing the individual (numerical) inputs. The same fuzzy output can be obtained by determining for every fuzzy rule *i* the individual fuzzy output:

$$Y^{i'}(y) = (X_1^i(x_1(t)) \vee ... \vee X_j^i(x_j(t)) \vee ... \vee X_N^i(x_N(t))) \vee Y^i(y)$$

Combination of those individual fuzzy outputs by applying a union results in the complete fuzzy output of the system:

$$Y(y) = \bigvee Y^{i'}(y)$$

Note that the inputs in are represented in a numerical way and the output is represented in a fuzzy way. Several defuzzification methods are available to translate the fuzzy output into a crisp, numerical output value [Jager et al. 1992a].

Based on this practical way of applying fuzzy rules and inference, a generalization of the inference mechanism can be made allowing to apply fuzzy rules and inference in a way rules and inference mechanisms are used in *conventional* expert systems.

In case of using a fuzzy inference engine it is possible that the conditions of rules are actions of other rules. The first question that arises is: *what happens with the intermediate results?*. The relevance of this question depends on the way the rule base is composed. Rules with actions which are used as conditions in other rules can be regarded as a kind of short hand notation for implementing more complex rules. In another perspective one can conclude that such a rule actually has defined a new imaginary membership function for a specific input. The correctness of this interpretation depends on the conditions and actions defined in the rule. Wether or not the result of the fuzzy inference engine is in agreement with the compositional rule of inference, depends on the correctness of the fuzzy rules implemented and the coherence between them.

Another possibility is the fact that a rule does not have a condition for every input avaible, which is however not relevant when assuming the following:

- the antecedent of every rule should imaginary contain a classification of every input;
- the membership grade of a *missing* input classification in the rule antecedent is assumed true.

From this point of view, rules are defined for every possible symbolic combination of the inputs, although not every rule is really individually provided in the rule base. Because missing input classifications are assumed to have a membership grade of true, in fact the union according to Lukasiewicz is applied on all classifications of the input in question, assuming consistent choices of membership functions for those classifications. So in fact a large imaginary rule base is defined which has a fuzzy rule for every situation, but only the relevant parts are actually provided. In this way it is possible to design fuzzy controllers which use different inputs in different situations: the inputs which are not relevant are simply ignored by the system.

4 The knowledge base

The knowledge base consists of rules and relations, which represent the knowledge in a logical way. The complete knowledge base can be divided into knowledge layers. The most primitive object is a fact. Rules and relations operate by means of conditions, actions and dimensions on objects, and thus can operate on facts.

4.2 Facts

The basic objects within the knowledge base are facts. Those facts can be pure abstract or can be linked with C-code. Each fact has a grade value, which denotes the truth of that fact. This value can be unknown or in the range from false (0) to true (1). A fact can be linked with C-code. This is done by using a **RICE_Linker** function. An example of linking a few facts with some C-code is

```
RICE_Linker(myLinker)
{
        RICE_Link("is this true?", RICE_grade = 0.6);
        RICE_Link("show truth value", printf("grade: %f\n", RICE_grade));
}
```

where **RICE_grade** is the grade value of the fact. The macro **RICE_Linker** starts a fact linker function and **RICE_Link** actually links a fact with some C-code. Assigning a value to the variable **RICE_grade** means that the inference engine gets that truth value returned when this fact is for example part of a condition. In this case the inference would get a truth value of 0.6 returned when testing the fact *is this true?* in a condition. When the fact is used in an action the variable **RICE_grade** holds the truth value at that moment. So in the second linked fact in the above example concluding the fact *show truth value* in an action will result in printing the string "grade: " followed by the current truth value and a carriage return. When this fact is situated in an action the execution of the action will depend on several other things. In section Actions this will be explained. In the chapter on how to build an application the **RICE_Linker** function will be discussed in more detail.

4.4 Rules

Rules consist of conditions and actions, which on their turn operate on an object, for example a fact. The basic form of a rule in general is

if <antecedent> then <consequent>

in which the antecedent is a set of conditions and the consequent a set of actions. The antecedent can be an intersection by using the **And** operator or a union in case of using the **Or** operator. Because of reasons founded in fuzzy set theory the type of operator should be the same. The next section will go deeper into those operators. The complete syntax of a rule is:

[**Rule** [<*symbolic*> [: <*status* = ***On***>]]]
<*condition*>
...
<*condition*>
<*action*>
...
<*action*>

As can be seen a rule can optionally have a name and an initial status (default **On**). An example of a rule within RICE is:

Rule *a rule for decreasing control signal*
IfRun *error is positive small*
AndRun *error change is negative big*
ThenRun *change control signal negative medium*
ThenInf *control signal is decreased*

In the next section the conditions and actions used within rules will be discussed and will explain why a rule looks like the one in this example.

4.6 Conditions

Conditions form the antecedent of rules. The syntax of a condition is:

[**Condition** [<*symbolic*> [: <*status* = ***On***>]]]
<*type*>(**If** | **And** | **Or**)[**If**][**Not**][<*options*>] <*symbolic*> [: <*cut = 0.0*>]

which could be used to represent a most simple condition like:

ZAnd *weather is nice*

where the type of the **And** operator is of the **Z** type, which means the Zadeh like implementation: *min(a, b)*. This type of **And** operator is default, so the **Z** is redundant. An example of a more complex condition is:

Condition *my condition :* ***Off***
POrIfNotRun *some C-coded condition : 0.34*

This condition needs probably some explanation. First of all this condition has a name *my condition* and an initial status of **Off**, which means the condition is initially not active. The default is an active condition (**On**). Secondly the second keyword construction consists of several parts. First of all the type of **Or** operator is of the **P** type, which means a union from probability theory: *a+b-ab*. The **If** added right after the **Or** will prevent the optional run-time optimization algorithm moving this condition within the antecedent. So if this condition was for example the third within the antecedent, it will be so after optimization. The **Not** part within the keyword speaks for itself: a negation of the condition. The addition of **Run** to the keyword will force the inference engine to execute the corresponding C-code. After the colon a level-cut value is given. This means that if the returned grade value of the fact is lower than this level-cut it will be set to false (0). Another example of a condition is:

LOrLayOff *some layer*

This will test if the layer *some layer* is inactive. The type in this condition is of the **L** type (Lukasiewicz): *max(a+b, 1)*. This condition shows how a condition can be used to get data from another object within the knowledge base.

4.8 Actions

Actions are the basic elements of the consequent of rules. The complete syntax of an action is:

[**Action** [<*symbolic*> [**:** <*status* = ***On***>]]]
<*type*>(**Then** | **Else**)[**Not**][[<*options*>] [**:** <*grade = 1.0*> [**:** <*cut = 0.5*>]]

A simple example of an action is:

PThen *have a nice day*

where the intersection is of the **P** type (see previous section). This intersection is performed on the resulting truth value of the antecedent of the rule and the grade provided for this specific action (default set to true (1)). The **P** type is the default type, so in this example redundant. Another more complicated example is:

ThenRulOff *a rule with a name : 0.9 : 0.8*

which will inactivate the rule *a rule with a name* in case the truth value of the antecedent part of the rule times 0.9 (because of **P** type) is not below 0.8.

4.10 Relations

Relations consist of dimensions, which, like condition and actions, will operate on objects. Relations provide a way to combine a set of facts of which one can be inferred from the others. This means that a dimension can be used like a condition as well as an action, depending upon the way of inference is applied on the relation. The syntax of a relation is:

[**Relation** <*symbolic*> [**:** <*status* = ***On***>]]
<*dimension*>
...
<*dimension*>

A relation defined as above can be compared with an algebraic equation: when all but one elements are known the unknown one can be calculated/inferred. For example, a bi-implication can be implemented by a relation using two dimensions. The next section will explain what dimensions look like.

4.12 Dimensions

A dimension is part of a set of dimension, forming together a relation. They can be regarded as a condition as well as an action. Whether a dimension is used as a condition or an action depends on how the relation is used by the inference engine: which fact is needed in case of backward reasoning, which facts are known in case of forward reasoning. The syntax of a dimension is:

[**Dimension** [<*symbolic*> [**:** <*status* = ***On***>]]

*<type><type>***Dim**[**Not**][*<opts>*] *<symbolic>* [**:** *<grade = 1.0>* [**:** *<cut = 0.0>*]]

which is in fact a mix of the syntax of a condition and an action. For example, the dimension:

ZPDimNotRun *an example dimension : 0.8 : 0.1*

has a **Z** type of intersection when used as a condition and a **P** type of intersection when used as an action. Dimension inferred as conditions has a as default the **Z** intersection type and executed as action the default implication type is **P**. So a short notation for this example is:

DimNotRun *an example dimension : 0.8 : 0.1*

The grade value only has any meaning when the dimension is used as an action. The meaning of the level-cut value depends on the use of the dimension (conditions and actions both have a level-cut value). As can be seen from the syntax, dimensions can be named and assigned an initial status (default **On**). It is just another type of object in the knowledge base.

4.14 Layers

The knowledge in the knowledge base is seperated in one or more knowledge layers, which are inferred sequentially in the order as found in the knowledge base. Each layer is able to use all data inferred in the previous layers as well as it is able to use rules and relations from previous layers. So each layer can be seen as an extension of the previous layer(s). In each layer it is possible to perform initial actions. This is done by using the **Init** keyword. An example of this is:

Init *initialize system*

where *initialize system* can be linked with some C-code performing initialization tasks. In fact **INIT** can be seen as an action, so every construction of the **Then** or **Else** keyword allowed, can be used in combination with **Init**. For example:

InitRelOff *some relation*

will inactivate the relation *some relation*. You could use the **Init** keyword also to set the truth value of some facts to an initial value, like:

Init *one of the facts : 0.3*

which will set the initial truth value of the fact *one of the facts* to a value of 0.3.

6 Inference, reasoning and search

To inference your knowledge base the inference engine needs to perform reasoning and search methods. Currently depth-first backward and breadth-first forward reasoning is implemented. For real-time application the layers provide a way to apply the progressive reasoning principle. The forward and backward reasoning methods and how you will invoke those mechanisms in your knowledge base will be discussed in the following sections. The inference of the knowledge base is done by sequentially inferencing all layers. In case a layer is not active (**Off**) it will be skipped.

6.2 Breadth-first forward reasoning

In order to perform forward reasoning you need to specify starting data. With this starting data the forward reasoning mechanism will fire as much rules and relations as possible, using a breadth-first search method. Many times this can result in almost an exhaustive search. Starting data is denoted by the **Data** keyword:

Data <*fact*>

The inference engine will first check if this starting data is known. If not, it will perform depth-first backward reasoning (see next section) to get the starting data. In this example:

Data *you are ok?*

If *you are ok?*
Then *keep it that way!*

the inference engine will first get the starting data *are you ok?* and then perform forward reasoning, which will result in firing the rule. It is possible to have more than one facts as starting data. The inference engine will not use any rules and relations from succeeding knowledge layers. They will be regarded as inactive.

6.4 Depth-first backward reasoning

In order to inform the inference engine the knowledge layer should be inferred using the backward reasoning method, you should use the **Goal** keyword. The syntax for defining goals of the inference of a knowledge layer is:

Goal <*fact*>

For example:

Goal *the final conlusion*

will result in a backward reasoning (depth first) to prove the fact *the final conclusion*. All data inferred by previous layers and all rules and relation in those previous layers (when active of course) and the current one will be used to do so if necessary. When, for example, the following is found in the knowledge base

Goal *the final conclusion*

If *some fact from a previous layer*
Then *the final conclusion*

and the fact *some fact from a previous layer* is not known, then the inference engine will use rules and relations from previous layers to prove that fact and thus prove the goal of the current inference: the fact *the final conclusion*.

8 Macro's and include files

To make live easier when editing your knowledge base, two C-like things are offered. At every point in the knowledge base you can use two C-like things: the **Define** and the **Include** keyword. The macro definition works as follows:

Define <*macroname*>
... #<*i*> ...
... ... #<*j*>

in which <*i*> and <*j*> are integers (greater that 0) or the character '@', refering to the i^{th} and j^{th} argument or all arguments of the macro respectivily. In case of a macro definition all lines until the first blank line will be regarded as part of the macro. The syntax of using the macro somewhere in your knowledge base is:

<*macroname*> [<*arg1*> : ... : <*argN*>]

All the (optional) arguments of the macro are seperated by a colon (:). It is possible to use previously defined macro's in macro definitions. An example of the use of macro's is:

Define *my_rules*
If *#1*
Then *#2 : 0.7*
Rem *faking an empty line*
If *#1*
And *#2*
Then *#1 and #2 : 0.4*

...

my_rule a fact : another fact

which will result in two rules, namely:

If *a fact*
Then *another fact : 0.7*

If *a fact*
And *another fact*
Then *a fact and another fact : 0.4*

Using the **Rem** keyword an empty line can be introduced in the definition of a macro. Use of the '@' character is done like:

Define *myAnd*
PAnd *#@*

Define *myThen*
ZThen *#@*

myAnd *a condition*
myThen *an action : 0.8*

In this example norm types are defined for **myAnd** and **myThen**. Note that the names of macro's become in fact new keywords and therefor are not case-sensitive!

It is also possible to use include files like in C. The inclusion of other files is done according to the syntax:

Include <*filename*>

For example:

Include *always.kbi*

will include the file *always.kbi*. The function of this keyword is exactly like the '#include' preprocessor statement in the C programming language: the knowledge base is compiled just as if the file to be included is entirely included at the place where the **Include** keyword is found.

10 Building an application

This chapter will go through the steps necessary to build an application with RICE. This in combination with the examples provided with your copy of RICE should be sufficient. In order to build your own application using the RICE software, you should have all the necessary object modules and the include file RICE.H. Normally the object modules will be placed in a library. When building your application you should include the file RICE.H in the application files which call the RICE interface functions. It is also possible to include the header file RICEEASY.H instead of RICE.H. This will redefine all functions without the **rice_** and **RICE_** prefix.

10.2 Linking C(++) code to facts

To provide an open system and making a fast inference possible an easy linking of facts on the abstract level resided in the knowledge base and plain C- or C++ code resided in the application source code is provided. This linking is done by means of so-called **RICE_Linker** functions. An example of such a **RICE_Linker** function is:

```
RICE_Linker(myLinker)
{
        RICE_Link("is this true?", RICE_grade = 0.6);
        RICE_Link("show truth value", printf("grade: %f\n", RICE_grade));
}
```

Such a function is linked to an expert system by the **rice_UseLinker** function and will be explained in the section about user supplied code. Linking fact *sunny country* with C code can be done by:

```
RICE_Link("sunny country",
        float sunshine_fac = 3.0*sun_shine;
        RICE_grade = sunny_cntry(sunshine_fac);
);
```

or for example by:

```
RICE_Link("sunny country",
        if(sun_shine >= rain_fall)
                RICE_grade = 1.0;
        else
                RICE_grade = (sun_shine/rain_fall);
);
```

Connecting the action fact *make signal big* with a real action performed by C-code, can be achieved by:

```
RICE_Link("make signal big", make_big(sig, RICE_grade));
```

or by:

```
RICE_Link("make signal big",
{
        int i;
        for(i = 0; i < RICE_grade*10; i++) sig += SIG_STEP;
});
```

One thing you should known is that most C preprocessors get troubled when you use comma's not surrounded by extra parenthesis within the C-code part of the **RICE_Link** macro. For example

```
RICE_Link("linking this fact can cause a preprocessor error",
{
        int i, j;   /* probably results in error... */
        for(i = 0; i < 3; i++)
                for(j = 0; j < 6; j++)
                        some_function(i, j);
});
```

will get the C preprocessor to produce an error. A way to avoid this is by using the following trick:

```
RICE_Link("linking this fact will not cause a preprocessor error",
{
        int i; int j;   /* use this trick... */
        for(i = 0; i < 3; i++)
                for(j = 0; j < 6; j++)
                        some_function(i, j);
```

});

There is no problem calling functions from within the C-code part of the **RICE_Link** macro, because the have surrounding parenthesis. Within every **RICE_Link** part of a **RICE_Linker** function the following variables are present:

int **rice_id** : expert system identifier
int **rice_argc** : number of arguments
char ***rice_argv**[] : argument list
float ***rice_grade** : grade of truth at current moment

The argument passing is compatible with the argument passing of the command-line arguments to the main-function in C(++). Those variables all have macro equivalents. At the moment they do not differ, except for the variable ***rice_grade** which has a non-pointer macro equivalent **RICE_grade**. However, it is suggested to use the macro equivalents. The following constants are defined:

RICE_TRUE : 1.0
RICE_FALSE : 0.0
RICE_UNKNOWN : -1.0

These constants are defined in the file RICE.H so theoretically their use is not restricted to the **RICE_Linker** functions, but in practise there is not much need for them outside these functions.

10.4 Interfacing functions

A number of functions are provided for interfacing your application with the RICE software. The functions and their working will be discussed now one by one. First the expert system (ES) functions will de described:

Function: int **rice_CheckES**(*id*);
Arguments: Expert system identifier *id*.
Return value: On success 0, otherwise -1.
Explaination: This function checks wether expert system *id* exists or not.

Function: int **rice_CopyES**(int *id1*, int *id2*);
Arguments: Expert system identifiers *id1* and *id2*.
Return value: On success 0 or expert system identifier, otherwise -1.

Explanation: This function copies expert system *id1* to *id2*. In case *id1* equals 0 the function will return the identifier of a new expert system.

Function: int **rice_CreateES**(void);
Arguments: None.
Return value: On success an expert system identifier (>0), otherwise -1.
Explaination: This function creates a new expert system.

Function: int **rice_DeleteES**(int *id*);
Arguments: Expert system identifier *id*.
Return value: On success 0, otherwise -1.
Explaination: This function deletes a previously created (by means of **rice_CreateES**) expert system *id*.

Function: int **rice_MoveES**(int *id1*, int *id2*);
Arguments: Expert system identifiers *id1* and *id2*.
Return value: On success 0 or expert system identifier (>0), otherwise -1.
Explaination: This function moves expert system *id1* to *id2*. In case *id1* equals 0 the function will return the identifier of a new expert system.

To manipulate the knowledge within an expert system kernel, created and manipulated with the previously described functions, one can use the knowledge base (KB) functions:

Function: int **rice_ClearKB**(*id*);
Arguments: Expert system identifier *id*.
Return value: On success 0, otherwise -1.
Explaination: This function clears the knowledge base of expert system *id*. The truth values of all active facts are set to **RICE_UNKNOWN**.

Function: int **rice_CompileKB**(int *id*, char **fnm*);
Arguments: Expert system identifier *id* and name *fnm* of ascii knowledge base file.
Return value: On success 0, otherwise -1.
Explaination: This function compiles the ascii knowledge base in the file *fnm* into the knowledge base of expert system *id*. After this the knowledge base can be inferred.

Function: int **rice_CopyKB**(int *id1*, int *id2*);
Arguments: Expert system identifiers *id1* and *id2*.
Return value: On success 0, otherwise -1.

Explanation: This function copies the knowledge base of expert system *id2* to *id1*.

Function: int **rice_DestroyKB**(int *id*);
Arguments: Expert system identifier *id*.
Return value: On success 0, otherwise -1.
Explanation: This function destroys the knowledge base of expert system *id* completely. You normally do not use this function.

Function: int **rice_InferKB**(int *id*);
Arguments: Expert system identifier *id*.
Return value: On success 0, otherwise -1.
Explanation: This function starts the inference of the knowledge base of expert system *id*, previously compiled or loaded.

Function: int **rice_LoadKB**(int *id*, char **fnm*);
Arguments: Expert system identifier *id* and name *fnm* of binary knowledge base file.
Return value: On success 0, otherwise -1.
Explanation: This function loads the binary knowledge base in the file *fnm* into the knowledge base of expert system *id*. The binary knowledge base should by saved before by the function **rice_SaveKB**.

Function: int **rice_MoveKB**(int *id1*, int *id2*);
Arguments: Expert system identifiers *id1* and *id2*.
Return value: On success 0, otherwise -1.
Explanation: This function moves the knowledge base of expert system *id2* to *id1*.

Function: int **rice_RebuildKB**(int *id*, char **fnm*);
Arguments: Expert system identifier *id* and name *fnm* of ascii knowledge base file.
Return value: On success 0, otherwise -1.
Explanation: This function rebuilds the knowledge base of expert system *id* to an ascii file named *fnm*.

Function: int **rice_SaveKB**(int *id*, char **fnm*);
Arguments: Expert system identifier *id* and name *fnm* of binary knowledge base file.
Return value: On success 0, otherwise -1
Explanation: This function saves the knowledge base of expert system *id* to a binary file named *fnm*.

Function:	int **rice_SwapKB**(int *id*, void (**fnc*)(void));
Arguments:	Expert system identifier *id* and function pointer *fnc*.
Return value:	On success 0, otherwise -1
Explaination:	This function swapes the knowledge base of expert system *id* temporarily from memory to execute function *fnc*.

These functions are the interfacing functions for run-time communication with the expert system kernels. The next section will describe the explaination utilities.

10.6 Explanation utilities

There are two function which can be used for explaination of the inferring of a knowledge base by RICE. These functions are:

Function:	int **rice_ExplainHow**(int *id*, char **fct*, int *dpt*);
Arguments:	Expert system identifier *id* and desired depth value *dpt*.
Return value:	On success 0, otherwise -1
Explaination:	This function explains for *dpt* levels deep how the inference engine came to conclude fact *fct*, when inferring knowledge base of expert system *id*. A *dpt* value of -1 will result in unlimited depth.

Function:	int **rice_ExplainWhy**(int *id*, char **fct*, int *dpt*);
Arguments:	Expert system identifier *id* and desired depth value *dpt*.
Return value:	On success 0, otherwise -1
Explaination:	This function explains for *dpt* levels deep why the inference engine came to infer fact *fct*, when inferring knowledge base of expert system *id*. A *dpt* value of -1 will result in unlimited depth.

These functions can also be used using the following macro:

RICE_Explain(*id*, <***Function***>, *fct*, *dpt*);

which replaces the function:

rice_Explain<**Function**>(*id*, *fct*, *dpt*);

The output of both these functions can be redirected using function **rice_RedirectExplain**, which will be explained also in this chapter. Default all explaination will be done using the standard output.

10.8 User supplied code

To make RICE use your own linked facts and self coded logical operators the so-called Use-functions are avaliable: UseLinker, UseIntersection and UseUnion. The will be discussed hereafter one by one.

Function:	int **rice_UseLinker**(int *id*, int *cnt*, **RICE_LINKER** *lnk*[]);
Arguments:	Expert system identifier *id*, number of linker functions *cnt* and array with linker function pointers *lnk*.
Return value:	On success 0, otherwise -1.
Explaination:	With this function you can supply a number (*cnt*) of linker functions, by means of *lnk*, to expert system *id*.

Function:	int **rice_UseIntersection**(int *id*, int *cnt*, **RICE_NORM** *nrm*[]);
Arguments:	Expert system identifier *id*, number of intersection norms *cnt* and array with norm function pointers *nrm*.
Return value:	On success 0, otherwise -1.
Explaination:	This function can be used to supply a number (*cnt*) of self-written intersection operators (*lnk*) to expert system *id*.

Function:	int **rice_UseUnion**(int *id*, int *cnt*, **RICE_NORM** *nrm*[]);
Arguments:	Expert system identifier *id*, number of union norms *cnt* and array with norm function pointers *nrm*.
Return value:	On success 0, otherwise -1.
Explaination:	This function can be used to supply a number (*cnt*) of self-written union operators (*lnk*) to expert system *id*.

The Use-functions can be replaced by the following macro calls:

RICE_Use(*id*, **Linker**, *cnt*, *lnk*);
RICE_Use(*id*, **Intersection**, *cnt*, *nrm*);
RICE_Use(*id*, **Union**, *cnt*, *nrm*);

These macro's call the corresponding functions, so type checking is performed anyway. In the following example a user-written intersection and a linker function are provided to an expert system:

```
float my_and(float x, float y)
{
        return x*y;  /* is in fact probability intersection */
}
...
RICE_Linker(myLinker)
{
        RICE_Link("is this true?", RICE_grade = 0.6);
        RICE_Link("show truth value", printf("grade: %f\n", RICE_grade));
}
...
void aFunction(void)
{
        RICE_LINKER linkerList[] = {myLinker};
        RICE_NOrM andPtr = myAnd;
        int anES;
        ...
        anES = rice_CreateES();
        rice_UseLinker(anES, 1, linkerList);
        RICE_Use(anES, Intersection, 1, &andPtr);
        ...
}
```

As can be seen in the example the Use-function or macro's need a function pointer array. Two ways of doing this in case there is only one function pointer are shown.

10.10 Adjustment functions

It is possible to make some changes to the default behaviour of the RICE software so it will fit better into your application or will give better performance at run-time. To do so, several macro's and functions are available: Redirect-, Set-, Reset- and Switch-functions. First the Redirect-functions will be discussed. They provide a way to let RICE perform certain things using your functions and not its own. The redirection-functions will be explained now one by one:

Function: int **rice_RedirectAbort**(int *id*, (void **fnc*)(void));
Arguments: Expert system identifier *id* and function pointer *fnc*.
Return value: On success 0, otherwise -1.
Explaination: This function can be used to supply your own abort function *fnc* for expert system *id*. The abort function is called in case of fatal errors.

Function: int **rice_RedirectAsk**(int *id*, (void **fnc*)(int, int, char *[], float *));
Arguments: Expert system identifier *id* and function pointer *fnc*.
Return value: On success 0, otherwise -1.
Explaination: This function can be used to supply your own ask function (**Ask** or **Usr**) for expert system *id*, where *fnc* is the function name. The function *fnc* receives four arguments: an expert system identifier, number of arguments, argument list and a grade.

Function: int **rice_RedirectExplain**(int *id*, (void **fnc*)(int, char *));
Arguments: Expert system identifier *id* and function pointer *fnc*.
Return value: On success 0, otherwise -1.
Explaination: This function can be used to redirect the explaining from the functions **rice_ExplainHow** and **rice_ExplainWhy** to function *fnc* for expert system *id*. The function *fnc* receives two arguments: an expert system identifier and a string.

Function: int **rice_RedirectInform**(int *id*, (void **fnc*)(int, char *[], float *));
Arguments: Expert system identifier *id* and function pointer *fnc*.
Return value: On success 0, otherwise -1.
Explaination: This function can be used to supply your own inform function *fnc* (**Inf** or **Usr**) for expert system *id*. The function *fnc* receives four arguments: an expert system identifier, number of arguments, argument list and a grade.

Function: int **rice_RedirectProtect**(int *id*, (void **fnc*)(int));
Arguments: Expert system identifier *id* and function pointer *fnc*.
Return value: On success 0, otherwise -1.
Explaination: This function can be used to supply your own protect function *fnc* for expert system *id*. The protect function is used in case critical parts within the inference engine need to be proteced from interupting events.

Function: int **rice_RedirectReport**(int *id*, (void **fnc*)(int, int, char *));
Arguments: Expert system identifier *id* and function pointer *fnc*.
Return value: On success 0, otherwise -1.

Explaination: This function can be used to supply your own report function *fnc* for expert system *id*. The report function is used in case of warnings (**RICE_WARNING**), errors (**RICE_ERROR**) or informational messages (**RICE_MESSAGE**) during operations by the interfacing functions. The function *fnc* receives three arguments: an expert system identifier, a message type and a message.

Each Redirect-function:

rice_Redirect<**Function**>(*id*, *fnc*);

can also be replaced by a corresponding macro:

RICE_Redirect(*id*, <***Function***>, *fnc*);

The following example shows redirection of the Ask-function in some application:

```
void myAsk(int id, int argc, char *argv[], float *grd)
{
        printf("%s ? ", argv[0]);
        scanf("%f", grd);
}
...
void aFfunction(void)
{
        int myES;
        ...
        myES = rice_CreateES();
        ...
        rice_RedirectAsk(myES, myAsk);
        /* is equal to: RICE_Redirect(myES, Ask, myAsk) */
        ...
}
```

The set-, reset- and switch-functions provide a way to set, reset and switch the run-time settings of the RICE software. The following settings and their default value are currently implemented:

Interactive: Ask from and inform to user (**On**)
Recursive: Allow recursively firing of rules and relations (**Off**)

Repetitive: Run code each time indicated by **Run**-key (**On**)
Selective: Select crucial conditions to increase speed (**Off**)

The syntax of the macro's is like:

RICE_<**Command**><**Flag**>(*id*);

or like:

RICE_<**Command**>(*id*, <***flag***>);

The set-, reset- and switch-macro's all use the functions:

rice_Set<**Flag**>(int *id*, int *act*);

where the appropriate flag of expert system *id* will be set if *act* is greater than zero, reset if *act* equals zero and switched if *act* is less than zero. For example, setting the Selective-flag for expert system *id* on can be done in the following three ways:

RICE_SetSelective*(id);*
RICE_Set*(id,* ***Selective****);*
rice_SetSelective*(id, 1);*

Whatever seems more convenient to use can be used.

10.12 The C++ wrapping class RICE

For easy use of the RICE toolbox in C++ applications a C++ wrapping class **RICE** is provided. The definition of this wrapping class can be found in the header file RICE.HPP. This header file is automaticaly included in the header file RICE.H when a C++ compiler is used. The class RICE has the following member functions (the working of these member functions can be found in the previous sections). The following constructors are provided:

RICE::**RICE**();
RICE::**RICE**(**RICE**& *id*);

A destructor is provided but is normally not used directly in the application code. Overloaded operators for the RICE class are:

RICE& RICE::operator =(**RICE**& *id*);
RICE& RICE::operator =(char **fnm*);
RICE& RICE::operator +=(char **fnm*);

These functions copy an expert system, compile an ascii knowledge base file and append an ascii knowledge base file, respectivily. The redirection member functions are:

int RICE::**RedirectAsk**(void (**fnc*)(int, int, char *[], float *));
int RICE::**RedirectInform**(void (**fnc*)(int, int, char *[], float *));
int RICE::**RedirectReport**(void (**fnc*)(int, int, char *));
int RICE::**RedirectExplain**(void (**fnc*)(int, char *));
int RICE::**RedirectProtect**(void (**fnc*)(int));
int RICE::**RedirectAbort**(void (**fnc*)(void))

The explaination member functions are:

int RICE::**ExplainHow**(char **fct*, int *dpt*)
int RICE::**ExplainWhy**(char **fct*, int *dpt*)

The set, reset and switch member functions of the **RICE** class are:

int RICE::**SetAdaptive**(int *act* = 1);
int RICE::**SetInteractive**(int *act* = 1);
int RICE::**SetRecursive**(int *act* = 1);
int RICE::**SetRepetitive**(int *act* = 1);
int RICE::**SetSelective**(int *act* = 1);

int RICE::**ResetAdaptive**();
int RICE::**ResetInteractive**();
int RICE::**ResetRecursive**();
int RICE::**ResetRepetitive**();
int RICE::**ResetSelective**();

int RICE::**SwitchAdaptive**();

int RICE::**SwitchInteractive**();
int RICE::**SwitchRecursive**();
int RICE::**SwitchRepetitive**();
int RICE::**SwitchSelective**();

The set member functions provide the **rice_Set<*Flag*>** function as well as the **RICE_Set** macro compatibility using a default value. The use member functions are:

int RICE::**UseLinker**(int *num*, **RICE_LINKER** **lnk*);
int RICE::**UseIntersection**(int *num*, **RICE_NORM** **nrm*);
int RICE::**UseUnion**(int *num*, **RICE_NORM** **nrm*);

The knowledge base manipulating member functions are:

int RICE::**AppendKB**(char **fnm*);
int RICE::**ClearKB**();
int RICE::**CompileKB**(char **fnm*);
int RICE::**DestroyKB**();
int RICE::**InferKB**();
int RICE::**LoadKB**(char **fnm*);
int RICE::**RebuildKB**(char **fnm*);
int RICE::**SaveKB**(char **fnm*);
int RICE::**SwapKB**(void (**fnc*)());
int RICE::**CopyKB**(RICE& *id*);
int RICE::**MoveKB**(RICE& *id*);

The functions manipulation the basic expert system functions (creating, deleting, ...) are hidden by the class definition. All expert system adjustment functions, functions for user supplied code, explanaition functions and knowledge base manipulating functions are transformed into member functions.

10.14 Tools and utilities

When building fuzzy expert systems you normally need some additional software for manipulation of fuzzy sets, defuzzification, etc. In the directory RICE\TOOLS a source file (FUZZY.C) and header file (FUZZY.H), as well as a small documentation file (FUZZY.DOC), can be found for dealing with fuzzy sets in your application.

Several utility programs are provided along with the RICE software. These utilities include a RICE knowledge base command-line compiler (RICECC) and a RICE knowledge base preprocessor (RICEPP). These utilities can be found in the directory RICE\BIN. Starting a utility without arguments will result in some information on how to use the program.

10.16 Examples

When receiving a copy of the RICE software a number of examples are included. They try to explain some of the things possible when using RICE. If you as a user of RICE has build an understandable and compact example, and which you think has some educational value to others, do not hesitate to contact the author. The more examples the more future users can profit of experience of users of the RICE software.

By now there are two examples provided in the RICE\EXAMPLES directory: a simple application named SIMPLE (a C++ version SIMPLEPP, an upgrade version SIMPLE3X and a version using the *easy* macro's, named SIMPLEEC, are also supplied) for performing the interfacing functions and a game example GAME, which lets you play tic-tac-toe against an expert system or let two expert systems play against eachother. Along with the examples small documentation files (*.DOC) with information on how to build the examples and makefiles are provided.

12 Copyright, warranty and updates

For updates you should send a request for the latest version of the RICE software to the author. The address for correspondence is:

René Jager

Delft University of Technology
Department of Electrical Engineering
Control Laboratory
Room 12.06

Mekelweg 4
P.O. Box 5031
2600 GA Delft
The Netherlands

Phone: (015) 78 51 14
Telex: 38151 butud nl
Fax: +31-15 65 67 38
E-mail: R.Jager@ET.TUDelft.NL

In case you discover bugs or have suggestions or remarks, please report those to the author.

References

[Jager *et al.* 1992a] R. Jager, H.B. Verbruggen and P.M. Bruijn. *The role of defuzzification methods in the application of fuzzy control.* Proceedings IFAC SICICA '92, Málaga, Spain, May 1992.

[Jager *et al.* 1992b] R. Jager, H.B. Verbruggen, P.M. Bruijn. *Fuzzy inference in rule-based real-time control.* Proceedings IFAC AIRTC '92, Delft, The Netherlands, June 1992.

[Jager *et al.* 1992c] R. Jager, H.B. Verbruggen and P.M. Bruijn. *Fuzzy inference in rule-based control systems. Proceedings of the First International IEE Conference on Intelligent Systems Engineering*, Edinburgh, Scotland, U.K., August 1992.

[Krijgsman *et al.* 1991] A.J. Krijgsman, R. Jager, H.B. Verbruggen and P.M. Bruijn. *DICE: a framework for intelligent real-time control.* Proceedings 3th IFAC Workshop AIRTC '91, Napa (Ca), U.S.A., September 1991.

Appendix B: Syntax

<layer>
[**Layer** [*<symbolic>* [: *<status = **On***>]]]
...
<data>
...
<goal>
...
<init>
...
<rule>
...
<relation>
...

<data>
Data *<symbolic>*

<goal>
Goal *<symbolic>*

<init>
Init[*<options>*] *<symbolic>* [: *<grade = 1.0>*]

<rule>
[**Rule** [*<symbolic>* [: *<status = **On***>]]]
<condition>
...
<condition>
<action>
...
<action>

<relation>
[**Relation** *<symbolic>* [: *<status = **On***>]]

<dimension>
...
<dimension>

<condition>
[**Condition** [*<symbolic>* [: *<status = **On***>]]]
<type>(**If** | **And** | **Or**)[**If**][**Not**][*<options>*] *<symbolic>* [: *<cut = 0.0>*]

<action>
[**Action** [*<symbolic>* [: *<status = **On***>]]]
<type>(**Then** | **Else**)[**Not**][*<options>*] [: *<grade = 1.0>* [: *<cut = 0.5>*]]

<dimension>
[**Dimension** [*<symbolic>* [: *<status = **On***>]]]
*<type><type>***Dim**[**Not**][*<options*] *<symbolic>* [: *<grade = 1.0>* [: *<cut = 0.0>*]]

<symbolic>
character string

<status>
On, **Off**

<grade>
float in range [0, 1]

<cut>
float in range [0, 1]

<type>
L, **P**, **Z**, **U**[*<index>*]

<index>
integer (default 1)

<options>
Run, **Usr**, **Lay***<mode>*, **InvLay**, (**Con** | **Act** | **Dim**)**Cut**, (**Act** | **Dim**)**Grd**,
(**Lay** | **Rul** | **Rel** | **Con** | **Act** | **Dim**)*<status>*

<*mode*>

Bkw, **Frw**

Appendix D: Keywords

Act	Referencing a action.
Action	Defining a new action with possible symbolic for reference and possible initial status.
And	Logical intersection operator.
Bkw	Backward (depth-first) reasoning mode of layer.
Con	Referencing a condition.
Condition	Defining a new condition with possible symbolic for reference and possible initial status.
Cut	Refers to the level-cut of a condition, an action or a dimension and acts more-or-less as a logic threshold.
Data	Defining a fact to be the starting data for forward reasoning in the current layer.
Define	Defining a macro like in C.
Dim	Logical relation (mixed intersection and implication) operator, or referencing a dimension.
Dimension	Defining a new dimension with possible symbolic for reference and possible initial status.
Else	Implication, performed when degree of antecedent of rule is equal or less than the cut value.
Frw	Forward (breadth-first) reasoning mode of layer.
Goal	Defining a fact to be the goal of backward reasoning in the current layer.
Grd	Refers to grade of an action or a dimension.
How	Starting explaination utility from within an action.

If Preceeding the first condition within the antecedent of a rule or preventing the inference engine to be selective (when this mode is on) in trying to prove conditions above and beneath the condition in question.

Include Includes a file like in C.

Init An initial action for the current layer with a possible.

Inv Invokes a layer (should be followed by **Lay** keyword).

L Lukasiewicz type of fuzzy logic: max(p + q - 1, 0) for intersection and min(p + q, 0) for union.

Lay Referencing a layer.

Layer Defining a new layer with possible symbolic for reference and possible initial status.

Not Negation of (fuzzy) logic: (1 - p).

Off Inactive status of object.

On Active status of object.

Or Logical union operator.

P Probabilistic type of fuzzy logic: (pq) for intersection and (p + q - pq) for union.

Rel Referencing a relation.

Relation Defining a new relation with possible symbolic for reference and possible initial status.

Rul Referencing a rule.

Rule Defining a new rule with possible symbolic for reference and possible initial status.

Run The symbolic in the condition, action or dimension is related to C-code, which has to be executed.

Shw Starting explaination utility from condition, action or dimension.

Then Implication, performed when degree of antecedent of rule is equal or more than the cut value.

U User type of fuzzy logic, a user supplied function for intersection or union is used (see **rice_UseIntersection** and **rice_UseUnion**).

Usr The symbolic is going to be asked when used in condition, informed when used in an action and asked or informed when used in a dimension, depending on the direction the relation is inferred.

Why Starting explaination utility from condition.

Z Zadeh type of fuzzy logic: min(p, q) for intersection and max(p, q) for union.