J Indian Inst Sci., Nov.–Dec. 1996, 76, 793–812.

SRT: A knowledge engineering tool for scene recognition

ANUPAM BASU*, S. SARKAR, A. K. MAJUMDAR AND K. A. RAO[†]
*Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721302, India.
email: anupam@cse. iitkgp.ernet.in
[†]University of North Carolina, Chapel Hill, USA.

Received on January 1, 1995; Revised on March 8, 1995.

Abstract

This paper describes the salient features of a knowledge engineering tool developed for scene recognition. Though the tool has been developed based on the requirements of a specific application area, the tool has been made general enough to be able to cater to a large number of applications. The tool, in its extended version, can deal with fuzzy as well as crisp data and knowledge. A novel technique, termed rule stratification, has been introduced and utilized in this tool. The paper also describes an application of the tool and presents a comparison of the features with those of the commercially available tools.

Keywords: Expert systems, scene recognition, fuzzy logic, predicate logic, inference machine, rule-based systems.

1. Introduction

Expert systems[1–3] have proved to be useful in application areas which involve domain-specific knowledge and expert heuristics for solving problems. Typical examples of such domains include medical diagnosis[4,5], computer-aided design and diagnostics[6,7], image processing[8], complex scheduling problems[9], etc. A number of expert system shells, such as EMYCIN[10], ART, CLIPS, KES[11], OPS5[12] are presently available, which provide the knowledge engineers with an environment for incorporating domain-specific knowledge to build an expert system. Such shells or tools can be characterized by the knowledge representation scheme supported, the inference mechanism, the conflict-resolution strategies adopted, the facilities to hook up with external programs, etc. Depending on the application domain, a knowledge engineer has to analyze the facilities offered by the tools and select a proper one for developing expert systems. Since the available shells are developed to provide a general-purpose environment, without any problem area in view, often a lot of tailoring is required to fit in the domain-specific requirements in an existing tool. This, in turn, reduces the efficiency and flexibility of inferencing.

In this paper, we address the problem of scene recognition. The problem can be defined as: Given a set of attributes of the parts forming a scene, or of the component regions of a scene, the scene has to be identified. The task is equivalent to that of high-level image interpretation and requires application of considerable domain knowledge

*For correspondence.

and heuristics. The necessity of knowledge-intensive computation is more pronounced because often all the attributes are not known, and hence algorithmic approach does not turn out to be a suitable candidate.

In order to build an expert system for the purpose, we analyzed the domain and identified a number of requirements to be specified by the expert system tool, so that efficiency and flexibility are not traded out. It was found that none of the available tools directly cater to all the needs identified.

In this paper, we first identify the requirements which are to be satisfied by a tool for developing an expert system for scene recognition. Next, we present the basic architecture, knowledge representation and inference mechanism of SRT, a scene recognition tool. We then discuss the extensions incorporated in SRT to accommodate fuzzy reasoning capability. Finally, we present an application of SRT for identifying a residential complex, along with a comparison of the features of SRT with those of well-known shells and developing environments.

2. Overview of SRT

2.1. *Requirement analysis for the domain*

In the class of problems, addressed by SRT, a scene composed of many regions or an object composed of many parts has to be recogized. Information about a region may not be complete at the time of information acquisition. For instance, the identity of the regions comprising a scene may not be known at the outset. The system will have to first infer the missing data about the regions and then infer about the identity of the scene.

For example, information about an unidentified region might be: *The region is a square shaped one with an area of 350 sq m and there is a road and a pond nearby.* A typical rule might be: *If a region has a road nearby, with a square shape and area more than 299 sq m then the region is a house.*

The regions are entities comprising the attributes. Some typical attributes are:

name	: The region name, which acts as the region identifier.
shape	: The shape of the region, *e.g.*, 'square', 'polygon', 'rectangular', 'irregular', etc.
length	: The length of the region.
width	: The width of the region.
perimeter	: The perimeter of the region.
centroid	: The coordinates of the centroid.
major-axis	: The major axis of the region.
minor-axis	: The minor axis of the region.

The different regions may be related to each other. A few example relations are:

nearby	: This is a set of all those regions which lie near to this region. This set is provided to the system.
surrounded by	: This is a set of all those regions which surround this region.
adjacent to	: This is a set of all the adjacent regions.
intersects	: This is s set of all the regions which intersect it.
faraway from	: This is a set of all the regions far away from it.

These relations are precomputed and provided as inputs to the system. The input data may be viewed as consisting of a set of different region entities (of a particular scene) which can be represented as frames. Each entity in turn has several attributes that may be known or unknown. The attributes of a region can be represented as slots of the corresponding frame. The unknown region attributes have to be inferred from the known attributes using a set of rules. Through additional inference, involving more rules and the known attributes, the identity of the entire scene has to be determined. Thus, a combination of frames and production rules form an ideal knowledge representation scheme for this problem.

The following are the required features identified which have to be addressed by a tool to be applicable to the scene recognition problem.

(1) In this particular problem (scene identification) the same set of rules have to be applied for identifying multiple regions. Thus the rules have to be of generic nature which can apply to different instances of regions. Hence, the facility to define and handle quantified rules has to be provided in the proposed system.

(2) In order to handle relations like *nearby* (there may be a varying number of regions nearby different regions), it was found that this relation is best represented by a set. Therefore, besides the conventional data types like real, string, etc., a set data-type is required. Over and above, it is found that in some cases the set contained only strings while in others, only real numbers suffice. Hence, a set each of string and real should be incorporated.

(3) To effectively utilize the 'set' data-type, whether real or string, set-related predicates and actions are needed. For example, it is found that the predicate 'belongs to'—which tests for set inclusion, and a new action 'addinto'—which performs the addition of a new element into a set, enhance the performance of the system.

(4) Some of the rules provided may have both AND and OR connectives in their antecedent fields. Therefore, in order to reduce the number of rules in the rule base, facility to define rules with both AND and OR connectives should be allowed.

(5) It was found that there were a number of instances of rules being recursively dependent. So a strategy has to be adopted to handle such rules.

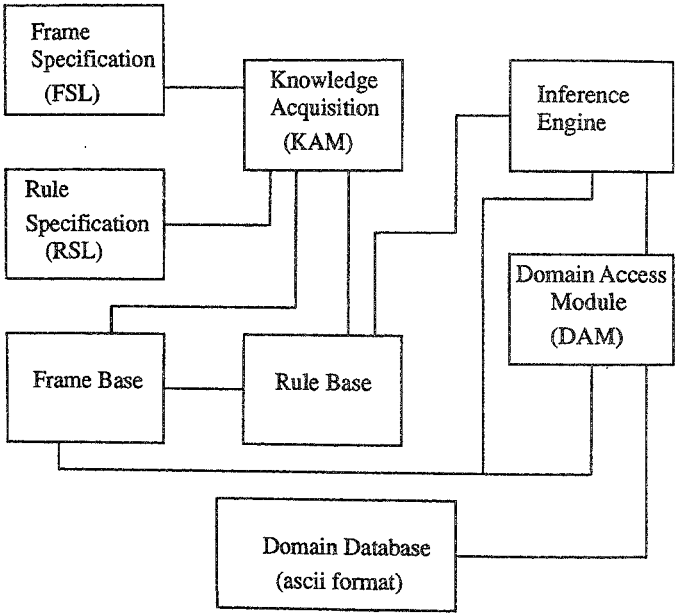
(6) A *stratification strategy* is necessary to club the rules into various strata which would be the order in which they would be fired. For instance, to identify if the scene is a residential area, the presence of houses must be established. Thus the rules to identify houses must be fired before those required for identifying a residential area.

(7) In some cases external functions have to be called. Therefore, a facility to execute external procedures is required. This also provides a general-purpose procedural construct that can be executed during inferencing.

Considering all the above features, it was found that no single existing commercial system would provide all the features required. Using any of the available systems would require considerable tailoring and would result in inefficient reasoning. Moreover, the problem of termination of recursive rules would be present in these systems. These reasons prompted us to develop a new shell SRT to deal with the problem of scene recognition. In the following sections we present first the knowledge representation and acquisition methods followed by inferencing mechanism of SRT.

3. Overall architecture and knowledge representation

Figure 1 presents the schematic architecture of SRT. Each module of the architecture is described in brief.



Frame Specification (FSL)
Knowledge Acquisition (KAM)
Inference Engine
Rule Specification (RSL)
Domain Access Module (DAM)
Frame Base
Rule Base
Domain Database (ascii format)

FIG. 1. Architecture of the system (nonfuzzy).

KAM: This is the knowledge acquisition module which is required in every expert system shell. KAM acquires the knowledge represented in a suitable form and converts it into the internal representation required by the shell. KAM also has additional functions like the ordering of rules, etc., which will be described in Section 3.

Inference engine: This module performs the actual processing. It uses the knowledge acquired by KAM and infers new facts from existing ones. The inference engine uses various strategies to carry out the process of inference. SRT's inference engine uses depth first search and forward chaining mode of inferencing.

DAM: This stands for domain access module. This is required for the implementation of the predicate rules. DAM has two main functions: To load the domainbase into the system at the beginning of the inferencing session, and to instantiate a variable during the process of inferencing.

Rulebase and Framebase: These two modules store the domain-specific rules and frames, respectively. The rules and the frames are specified in a rule-specification language (RSL) and a frame-specification language (FSL). This knowledge description is obtained by KAM and converted into its internal representation.

Domain: Domain is essentially an external database. This database maintains an ASCII file for each variable used in the rules. A file corresponding to a variable stores the admissible values of the variable.

In SRT, domain knowledge is represented using FSL and RSL. KAM compiles the input file and extracts the knowledge into its internal representation.

SRT expects a text file containing the framebase and rulebase descriptions in FSL and RSL, respectively, as input. The file can be created using any editor. In the input format, the framebase description is followed by rulebase description.

3.1. *Fact representation*

The input facts are viewed as consisting of a set of different region entities. Each entity in turn has several attributes that may be known or unknown. Examples of attributes in the present case are length, width, area, radiometric characteristics, etc., of a region. In SRT, an entity is represented as a frame and its attributes are denoted by the slots of the corresponding frame. The schema for the framebase has to be specified as input to the system. In the schema description, the type of the slot has to be specified. The value may or may not be specified.

The slots in SRT may contain data which may be real/integers, strings, set of real/integers, and set of strings. It should be noted that the real and integers are treated in the same fashion. Thus, essentially only real is used. The value of a slot can be changed during the execution only by the actions specified in the consequent of a rule.

Frames may also be used for storing control data such as no-of-houses, no-of-roads, parks-identified, etc. The slots of a frame may be used as temporary locations.

3.2. *Variable frames*

In SRT, rules may include quantified variables. Each variable corresponds to a frame, referred to as variable frames. A 'variable frame' can assume any value from a file stored in the external database domain. It may be added that whether a frame is a constant or corresponds to a variable cannot be known by inspecting it. In other words, no syntactic distinction is made. Thus, care has to be taken by the programmer to ensure that a frame is used consistently as either a constant or variable throughout the rulebase. The mechanism of a variable frame and variable binding is explained in detail in the sections on Stratification and Predicate rules.

3.3. *FSL*

FSL is used to specify the framebase to the system. In FSL, a framebase is specified by specifying all the frames in it enclosed by the two keywords *begin* and *end.* A typical framebase specification can thus be specified as <*begin* frame1, frame 2, ...frame *n* *end*>. Example of a sample frame written in FSL is shown below:

```
RegionX = frame
string obj-name = 'object1';
string obj-type;
string radio-attrib;
string obj-shape = 'elongated';
real length;
real width;
real area = 400;
real perimeter = 100;
set of string nearby = {'xx', 'aa'};
```

The above description is of a frame RegionX which has the following structure: slots containing string data: obj-name (initialised to 'object1'), obj-type, radio-attrib, obj-shape (initialised to 'elongated'); slots containing real data: length, width, area (initialised to 400), perimeter (initialised to 100); the slot nearby is of type set of strings, initialised to {'xx', 'aa'}.

3.4. *Rulebase representation*

The declarative knowledge of the knowledge base is stored in the rulebase in the form of production rules. Rules having similar functional characteristics can be grouped together into rule-blocks.

The production rules are of the typical form IF <*antecedent field*> THEN <*consequent field*>. The antecedent field is a conjunction or disjunction of predicates. The consequent field is a list of actions. In the antecedent field it checks for certain conditions on the slot values of the frames. The consequents may modify the slot values of frames or perform other actions such as input and output. SRT, as has been already mentioned, is based on predicate calculus, where variables appearing in the rule may be quantified. The variables (frames) occurring in the consequents are universally quantified, and the variables occurring in the antecedents are existentially quantified. To facilitate this, the structure of a general rule in the SRT is as follows:

<quantifierlist> IF *<antecedent field>* THEN *<consequent field>*

Each ruleblock may contain several rules. In turn, each rule has a number of clauses, consequents and quantifiers. The rulebase is specified in RSL.

Predicates: The antecedent field is in disjunctive normal form. It is a series of clauses linked by the OR connective. There should be at least one nontrivial clause for it to be valid. Each clause is a list of antecedents linked together by the AND connective. The antecedents themselves are in one of the following forms:

*<Frame-slot*1*><predicate><frame-slot*2*>*

*<frame-slot*1*><predicate><constant>*

If at least one of the clauses evaluates to true, the consequent part will be executed. The truth of the antecedent list is checked once before firing any of the consequent in the consequent list. So, after checking the truth of the antecedent field, the consequent actions are taken one after the other in the order in which they appear in the list of actions. The predicates allowed in SRT are =, <, >, *belongs* (*if an item belongs to the set mentioned*).

Actions: The consequent field is a list of consequents which specify what action to take and on which slot. The order of firing is the order in which they are written. The actions available in SRT are shown in Appendix I.

Quantifiers: The quantifiers may be either universal and existential. The quantifiers are maintained in the form of doublets : *<frame, domain file>*. The *frame* is any frame declared in the framebase. It should be noted here that such a frame is a variable in this rule. The domainfile is the file that contains the various instances of the variable. Care should be taken to ensure that the format of this file is the same as that of the frame. It may be observed that the variables occurring in the consequent are universally quantified and the others are existentially quantified.

3.5. *RSL*

RSL is the format in which the rules are to be written for SRT to be able to understand. The ruleblocks are specified one after another within pairs of "begin" and "end". A typical rulebase may thus be presented as

<begin Ruleblock1 *end, begin* Ruleblock2 *end,...., begin* Ruleblock *n end>*

A ruleblock may contain several rules. The shell keeps the knowledge represented in RSL in a file. The following is an example rule written in RSL:

Example 1

```
($RegionX.obj-shape == 'circular')
($RegionX.radio-attrib == 'water')
($RegionX.area > 799)
($RegionX.area< 1501)
OR
($RegionX.obj-shape == 'rectangular')
($RegionX.radio-attrib == 'water')
($RegionX.area > 799)
($RegionX.area <1501)
then (set $RegionX.obj-type 'pool')
(add $Status.no-of-pools 1)
(addinto $Status.pools$RegionX.obj-name);
```

The rule can be interpreted as: For all instances of RegionX in the domain file 'Resd.dat', if its shape is either circular or rectangular, and its radiometric attribute is water, and its area is between 799 and 1501, then mark it as a pool.

The region is marked as a pool by the actions taken by the consequents. Here the slot 'obj-type' is set to 'pool', the slot 'no-of-pools' of the frame 'Status' is incremented by one, and the region name (identifier) is added into the set 'pools' maintained as a slot of the frame 'Status'. It may be noted that the frame Status is not named in the quantifier list. Hence, it is treated as a constant frame and there is no domain attached to this frame. All updates are done in the framebase. However, at a later point of time the user may use it as a variable by simply referring to it in the quantifier list. In such a case, the earlier values stored have no meaning. Thus the user has to ensure that the frames are used consistently throughout.

3.6. *Stratification*

The statification feature is a novelty of SRT as this feature is not found in any of the commercial expert systems. It reduces the memory and time overheads posed by the Rete-matching algorithm[13]-based production systems. We resort to stratification algorithm to decide the firing order. Here, the rules are stratified into various strata each of which denotes a level of priority of firing the rules; for example, if rule-A is at a higher level than rule-B, then rule-A is fired first.

SRT has the ability to handle quantified rules. Forward chaining strategy is adopted for ruleblocks which have quantified rules. To backtrack across quantified rules, it is necessary to maintain the storage of all the past bindings. Therefore, a very simple strategy has been adopted in which the rules are fired in a precomputed order. This is carried on till all the rules have been fired. Here the assumption is that when the new round of firing is done, the values of the previous bindings do not get altered, *i.e.*, the rules are monotonic. The order in which the rules should be fired is governed by the following principle:

If rule-A is fired depending on the condition or value contained in slot-X (referred to in the antecedent field of rule-A), and if rule-B, when fired, modifies the contents of slot-X (referred to in the consequent field of rule-B), then rule-B should be fired first.

This is necessary since, after rule-B modifies the content of slot-X, the bindings for which rule-A may be fired may change. However, it may happen that there may be cyclic dependencies among the rules, for example, rule-A → rule-B → rule-C → rule-A.

Here, say, the rules are fired in the order A, B, C; then the contents of the slots on which rule-A depends may change after rule-C is fired. This may result in rule-A being fired for some new bindings for the variables. To take care of this situation all the rules in the cycle are fired until the bindings of variables attached to none of these rules change. Thus, not only the firing order but also the cycles have to be identified and passed on to the inference engine. In order to deal with such situations we utilize a *dependency graph* of the rules.

3.6.1. *The dependency graph*

A dependency graph is constructed to illustrate the dependencies that exist within the rules of a ruleblock. This is done after the rules have been acquired. The graph has directed edges and may have cycles. The nodes of the graph denote the rules . An edge from node 'a' to node 'b' indicates that the rule corresponding to node 'b' depends upon the rule corresponding to node 'a'. A rule "rule-A" is said to *depend* on another rule "rule-B" if at least one antecedent of rule-A contains a reference to a slot that is modified in the consequents of rule-B. It should however be noted that the action 'put' need not be checked since it does not modify any slot. In SRT dependency graphs are formed for each individual rule block.

3.6.2. *Utilization of the dependency graph*

After the rules have been acquired, a dependency graph is constructed for each rule-block. Using this graph, the cycles and the order of firing are determined using the stratification procedure. Once the order is decided, the inferene engine fires the rules according to the precomputed order.

3.6.3. *Stratification procedure*

When the dependency graph is acyclic, the firing order can be identified by topological sorting of the graph. When cycles are present, then firing of one rule may require the firing of some other rules, which in turn may require the first rule to be refired. And this could go on till any more firings make no difference to the framebase. Therefore, to resolve this, all the rules belonging to a cycle are clubbed together. There may be cases of cycles within cycles and overlapping cycles. To deal with those, all the cycles that overlap are merged. Therefore, the following course of action is taken:

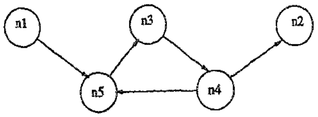


FIG. 2. Dependency graph after stratification.

1. Fundamental cycles are extracted from the dependency graph (a DFS strategy is used).
2. If two cycles have common node then the two cycles are merged to create a larger cycle. This removes the cases of overlapping cycles and cycles within cycles.
3. All the nodes which belong to a cycle are merged into a supernode. Each supernode, therefore, points to a list of rules corresponding to the nodes that were merged to give rise to the supernode. This completely removes the cycles from the graph which is now reduced to a DAG.
4. Topological sort is performed on the DAG to obtain a firing order of the nodes. The ordered list of nodes is then attached to the ruleblock.

Figures 2 and 3 illustrate how a given sample dependency graph is converted into a DAG by stratification.

4. Inference engine

The inference engine of SRT uses forward chaining. We have seen that two types of ruleblocks exist: propositional ruleblocks in which all the rules are propositional, and predicate ruleblocks in which at least one rule is a quantified rule. The inference engine uses two kinds of strategies to handle each of the ruleblock varieties. The strategies are enumerated in the following sections.

4.1. *Propositional rules*

The initial state is determined by the facts supplied. Based on this information, the inference engine attempts to reason from the initial state to the goal state (backtracking if required). The search space is already reduced by the division of the rulebase into rule- blocks. To further reduce the search space an active rule list (ARL) is formed.

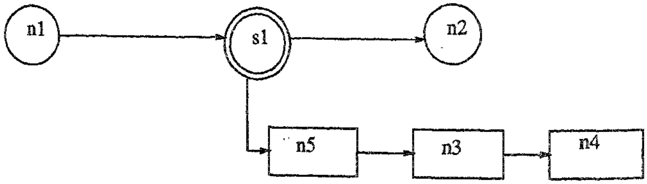


FIG. 3. Dependency graph after stratification (s1 is the supernode).

ARL is a list of those rules whose (its antecedent field's) truth value may be affected (TRUE).

SRT uses the depth first approach for selecting the next rule to be fired.

4.1.1. *Implementation of the DFS strategy*

The following strategy is followed:

1. First rule whose antecedent field evaluates to TRUE is fired.
2. All those rules whose antecedents access the slots which are modified by the firing of the rule are put on the ARL. This update is done using 'refblocks' associated with the modified slots. If a rule has already been fired then it is not put on the ARL. This ensures that the same path is not followed while backtracking.
3. The next rule to be fired is selected from the ARL. The first rule in ARL whose antecedents evaluate to TRUE is fired. The fired rule's status is updated to FIRED and it is removed from the ARL.
4. Steps 2 and 3 are repeated until no more rules can be fired and then the option for backtracking is given.

In SRT, a flag is reserved, in the status field of a rule, for indicating whether that rule is in ARL. A list of pointers to the rules in ARL is maintained. ARL is updated by adding deleting elements from the list.

4.1.2. *Backtracking*

When a rule is fired, with the firing of each consequent, the previous values of the modified slots are pushed on to a stack. The depth of the stack is equal to the number of rules fired. The stack element is a pointer to a list of slot-like structures which contain the old slot values and the address of the slot that has been modified.

Backtracking is done on users' choice. The number of levels to backtrack is left as a user option. On backtracking, the old slot values are retrieved from the stack and copied back.

4.2. *Quantified rules*

A different strategy is adopted for ruleblocks which have quantified rules. This subsection deals with the firing of rules in such ruleblocks.

4.2.1. *Firing a rule*

Once the firing order of rules in a block is decided, the next step is to fire the rules in that order whenever such a ruleblock is selected. The firing order is available to the inference engine in the form of a list of nodes and supernodes attached to the ruleblock. We have seen in the previous section that each node of the dependency supernode stands for a cycle and therefore points to a list of rules.

Execution involves traversing the firing order list and firing the rule associated with the nodes. However, the supernodes being different, they point to a list of rules, and are evaluated differently.

4.2.1.1. *Firing a node*

Obtain the rule associated with the node. Instantiate the variables with new bindings (this is done by the domain access module (DAM) at the request of the inference engine) obtained from their respective domains and fire the rule if the antecedents are satisfied. This process is repeated till the universally quantified variables have been instantiated with all possible combinations of the domain elements, and the existentially quantified variables assume values from a combination of domain elements that satisfy the antecedents.

4.2.1.2. *Firing a supernode*

A supernode represents a cycle. We have seen that evaluation of a cycle involves repeated firing of the rules in the cycle until there is no addition to the bindings attached to the variables, *i.e.*, a fixed point is reached. Here the assumption is that when the new round of firing is done, the values of the previous bindings do not get altered, *i.e.*, the rules are monotonic.

This is implemented by keeping a list of addresses of the domain elements (frames) with which the rule has been fired. If, by one round of firing of the rules in the cycle, new domain elements are added, then the process is repeated. If none of the rules have been fired with new bindings then the fixed point has reached, and hence, the process terminated.

4.3. *DAM*

We have seen in the rule structure that in a quantified rule the type of a variable is determined by the frame, to which it is associated, and the admissible domain values are kept in a file. The domain database is the collection of all such files. DAM interacts with the inference engine at the time of firing of the rule that requires multiple instantiations of the quantified variables.

Thus, DAM has broadly two functions:

(i) to load the domain values from the files into the data structures before firing a rule.

(ii) to instantiate a variable while inferencing at the request of the inference engine.

For example, suppose a rule with the quantifier (x:X) is fired, where the variable x appears in a consequent (universal quantification). The firing of this rule demands that the rule is to be fired for all bindings of x from domain X. In order to perform this, the inference engine calls for DAM which accesses the appropriate file, converts it into a set of frames and passes the set as a domain to the inference engine. The domain is restored back by the module on termination of the inference engine.

DAM is independent of the remaining part of the shell, whereby the organization of the database is not fixed. In fact, by suitably modifying DAM, SRT could be interfaced to any kind of database organization, *e.g.*, Ingres.

5. Fuzzy extensions of SRT

In the domain of scene analysis, information often suffers from linguistic imprecision. For instance, consider a typical decision rule

if ((region X has radiometric attribute water) AND
(region X shape circular or rectangular) AND
(region X has area > 799 sq m and < 1501 sq m))
then region X may be a POOL.

Here, the shape circular is imprecise since the actual shape available as data will never be exactly circular or rectangular. A better measure would be the degree of circularity and it may take a wide range of values. Thus, the factbase supplied can often be imprecise and uncertain in nature. Also the rule itself is not certain in identifying a pool.

In order to deal with such inherent uncertainties, SRT has been extended to support both exact and inexact knowledge. Fuzzy logic[14] has been adopted to capture the uncertainties.

In the proposed modification of SRT, we have allowed fuzzy as well as crisp terms and uncertainities in the rules and facts, and have employed fuzzy logic to handle approximate reasoning.

In this section, we briefly report the extensions of SRT, which has made it a useful tool, for fuzzy and crisp reasoning.

5.1. *Architectural extensions*

The architecture of the enhanced version of SRT is shown in Fig. 4. There is a user interface through which an user interacts with the system. Rules and facts can be specified using RSL and FSL as before. Facts can assume fuzzy values. The possibility function μ_F, associated with a fuzzy qualifier F, can be defined in the form of a table with an interpolation or in the form of a function. The fuzzy tables and functions are kept separately in the database and associated with the factbase. User-defined queries can also be defined and stored through the user interface. Rulebase, factbase, fuzzy qualifiers and user-defined query programs constitute the database termed the long-term database (LDB). KAM residing under the user interface is responsible for creation, modification and augmentation of LDB.

The fuzzy inference engine interacts with LDB for inferencing. The inferred facts and the intermediate computational results are kept in a temporary storage called the short-term database (SDB). DAM is active when inferencing is in progres. Details of DAM have already been discussed earlier.

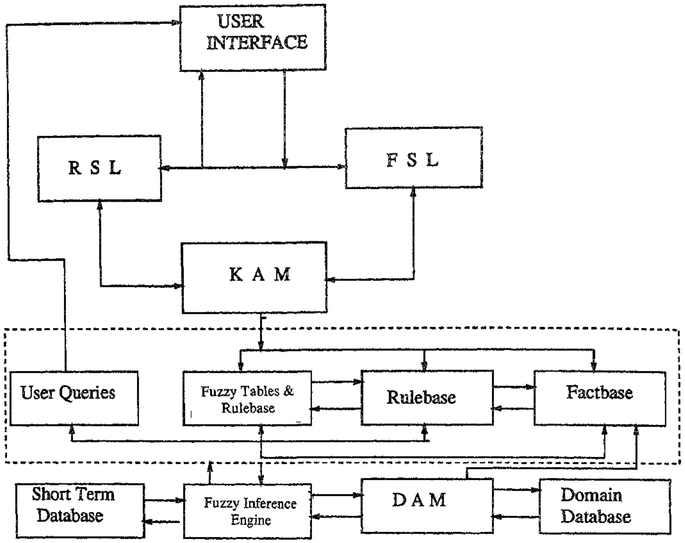


FIG. 4. Architecture of the proposed system (fuzzy).

Fuzzy knowledge is represented as a set of initial information organized as frames, slots, a set of fuzzy qualifiers and the domain-specific knowledge, organized as a set of production rules.

5.2. *Fuzzy extensions to the factbase*

The factbase is organized in the form of frames and slots as discussed earlier. A slot defines an attribute and the attribute value can be stored in the slot. Each value is associated with a *certainty factor* indicating that the fact may not be known to be completely true.

In the proposed system, a slot may be *crisp* or *fuzzy*. A crisp slot stores a nonfuzzy, deterministic value. A fuzzy slot stores fuzzy values. A fuzzy slot is associated with a fuzzy qualifier F and the associated membership function μ_F. μ_F is stored in the form of a table. We allow three types of fuzzy slots, namely, real number *with* or *without* interpolation and string *without* interpolation. Interpolation is necessary to obtain the membership value of an intermediate point that has not been stored in the table. However, in some cases, interpolation of an intermediate data point may not be meaningful due to the nature of the data itself.

A fuzzy slot may have a quantifier. A quantifier σ is a function from [0,1] to [0,1] that operates on the possibility function μ_F where F is a fuzzy qualifier. In the present version of the extended system we allow only a fixed quantifier function that can be one of the three types, namely, *not*, *very* and *more-or-less*. If a fuzzy slot allows quantifier, it is obvious that the slot must be real with interpolation.

The certainty associated with a slot may be fuzzy or crisp. In case of a crisp certainity, it is a value between 0 and 1 and the slot may be of any type. For fuzzy certainty, the type real with interpolation is allowed. The fuzzy certainty table is also associated with the slot.

5.3. *Extensions to rulebase organization*

The structure of the rule in the proposed system is more or less identical to the previous system. The general structure is an augmentation of the previous version of RSL,

IF <*quantifiers*> <*antecedents*> THEN <*consequents*> BELIEF <*belief-measure*>

Here *quantifier* denotes a predicate quantifier such as *for-all* or *exists*. An antecedent structure is also the same as the previous version of RSL

<frame>.<slot> <predicate> <value>
<frame>.<slot> <predicate> <frame>.<slot>

There may be a belief measure associated with a rule. The belief measure can be crisp or fuzzy. For a crisp belief, the *belief-measure* is a value between 0 and 1. A *fuzzy belief* is expressed as a fuzzy set with a membership function μ_T stored in the form of a table.

5.4. *Extensions to the inference engine*

The inference engine utilizes the rulebase, factbase, fuzzy information and the domain database for inferencing. The basic inference strategy remains the same as discussed earlier. However, the truth value evaluation of the antecedents and the truth assigmnent of the consequents will differ from the previous version due to the fact that the truth value of an antecedent in this case is no more from a domain of {0,1} but from a real number in the range [0,1]. Moreover, the certainty of a slot value being modified by a rule may not be done with total confidence. Here, it depends on the certainty of the antecedents as well as the belief of the rule itself. We now discuss the method adopted for evaluating the truth value of the antecedents and the certainty factors of the consequents.

Consider a general rule r containing fuzzy-valued antecedents and consequents.

IF $(A_1 = p_1)$ &..., & $(A_n = p_n)$ THEN Action $A = a$ BELIEF b.

Suppose, during execution, the facts obtained are:

$[A_1 = q_1, \ldots, A_n = q_n]$.

Let the truth of antecedent i be denoted as T_i. Four cases are possible. These are:

Case 1: p_i are q_i are crisp.

In that case T_i = IsEqual(p_i, q_i); where IsEqual returns 1 if $p_i = q_i$ else returns 0.

Case 2: p_i is crisp and q_i, fuzzy.

Here $T_i = \mu_{qi} p_i$.

Case 3: p_i is fuzzy and q_i is crisp.

Here $T_i = \mu_{pi} q_i$.

Case 4: p_i and q_i are both fuzzy.

Here $T_i = max[card(p_i \cap q_i)/card(p_i), card(p_i \cap q_i)/card(q_i)]$.

Truth of conjunction of two antecedents is

$T = min(T_1, T_2)$, where T_1 and T_2 are the truth values of the two antecedents.

Truth of disjunction of two antecedents is given by

$T = max(T_1, T_2)$.

If T exceeds a threshold determined by a prespecified α-cut, then the rule is made 'fireable'. When the rule is fired, it modifies slot values. The value of the slot getting modified will have some certainty factor which is a measure of the degree of correctness of the value. We compute the certainty factor as follows.

Suppose the rule being fired modifies the slot S with a value a which may be crisp or fuzzy. The rule itself has a belief measure b. If a is fuzzy, the corresponding membership function μ_a is already available. There may be four cases.

Case 1: a and b are crisp.

Then the certainty of the value $a = min(T,b)$.

Case 2: a is crisp and b, fuzzy.

Then the certainty of $a = \mu_b(T)$.

Case 3: a is fuzzy and b, crisp.

Then we assign A with a fuzzy set a^* having $\mu_a^*(x) = \min(T, b, \mu_a(x))$.

Case 4: a and b are both fuzzy.

Then we assign A with the fuzzy set $a^{**} = \min_x(\mu_{a*}^*(x), \mu_b(x))$; where a^* is a fuzzy set with $\mu_{a*}^*(x) = \min(T, \mu_a(x))$.

The modified value and the associated certainty factors are stored with the slot. They can be used for firing the subsequent rules that are dependent on this slot.

6. Applications

SRT has been successfully applied for identifying scenes in remote-sensing applications. An excerpt from the session on expert system execution for identifying a residential area is given next.

```
              Frames and rules satisfactorily acquired
                                .
                                .
                                .
                      Stratifying the rules ....

                  The dependency graph extracted:

           1 →
           2 → 1 5 6 7
           3 → 7
           4 → 7
           5 → 4 7
           6 → 4
           7 → 1

            The dependency graph, after stratification:

           1 →
           2 → 1 5 6 7
           3 → 7
           4 → 7
           5 → 4 7
           6 → 4
           7 → 1
           3 2 6 5 4 7 1
              Do you want to run in verbose mode ? n

                    IT IS A RESIDENTIAL AREA

   The roads are: Regions: { 'R10' 'R14' 'R20'}
   The pools are: Regions: {'R25'}
   The houses are: Regions: {'R1' 'R5' 'R7' 'R8' 'R9' '}
   The parks are: Regions: {'R30'}

                         This is DFS executing
                            SELECT WORKS

                           SELECT WORKS*

No more rules to fire in ruleblock no. 2, do you want to backtrack? y
Enter the number of levels to backtrack (maximum 2) : 2

                          BACKTRACK WORKS
                                END
Do you want another run ? n

                            Good Bye !!!
```

Table I

Criteria	*CLIPS*	*ART*	*OPS5*	*SRT*
Expressive power	Qualified rules	Qualified rules	Qualified rules	Qualified rules
Data types	Real string	Real string	Real symbols	Real, string, set
Object-oriented features	No	Yes	No	No
Implementation language	C	LISP	BLISS	C
Procedural constructs	if-then-else, while	if-then-else, while	None	Call
Disjunction (OR)	Yes	No	No	Yes
Inference paradigm	Forward chaining	Forward chaining	Forward chaining	Forward chaining
Rule matching	Rete	Rete	Rete	Simpler
Conflict resolution	Salience	Salience	Recency and specificity	FCFS
Rule selection	Recognize act	Recognize act	Recognize act	Precomputed and DFS
Stratification	No	No	No	Yes
Recursive rule evaluation	No	No	No	Yes
Fuzzy reasoning	No	No	No	Yes

6. Conclusion

In this paper, we have described the salient features of the expert system shell SRT, developed to deal with scene-recognition problems. In the extended mode it can deal with fuzzy as well as crisp knowledge. SRT supports predicate rules as well as propositional rules. Depth first search strategy and forward chaining are the inferencing strategies incorporated in the inference engine. While following the forward chaining strategy it is ensured that the order of firing of the rules is independent of the order in which they are input. This is ensured by *stratifying* the rules in the ruleblocks. SRT can also evaluate recursive rules which are identified in the course of stratification. Thus, it has the capacity to handle rules which are monotonic in nature regardless of the cyclic dependencies.

The knowledge acquisition module is user-friendly. It supports suitable languages for specification of rules and frames. Even though it has been developed expressly for the purpose of scene recognition, the shell developed is general purpose, with a wide range of possible applications with similar requirements. Table I compares various features of some commercial expert system tools with those of SRT.

Presently the features of SRT are being enhanced to incorporate facilities for explanation generation in "English-like" language. Also, the conflict-resolution strategy is being made more sophisticated.

Acknowledgement

This work was supported by a project sponsored by the Ministry of Defence, Govt. of India.

References

1. PETER, J.	*Introduction to expert systems*, 1986, Addison Wesley.
2. BRAMER, M. A. (ED)	*Research and development in expert systems*, 1985, Cambridge University Press.
3. DAVIS, R. AND LENAT, D.	*Knowledge-based systems in artificial intelligence*, 1980, McGraw-Hill.
4. SHORTLIFFE, E. H.	*Computer-based medical consultation: MYCIN*, 1976, Elsevier.
5. SHORTLIFFE, E. H., BUCHANAN, B. G. AND FEIGENBAUM, E. A.	Knowledge engineering for medical decision making: A review of computer-based clinical decision aids, *Proc. IEEE*, 1979, **67**, 1207–1224.
6. TAYLOR, J. H., AND FREDERICK, D. K.	An expert system architecture for computer-aided control engineering, *Proc. IEEE*, 1984, **72**, 1795–1805.
7. FINK, P. K., LUSTH, J. C. AND DURAN, J. W.	A general expert system design for diagnostic problem solving, *IEEE Trans.*, 1988, **PAMI-7**, 553–560.
8. NAZIF, A. AND LEVINE, M. D.	Low-level image segmentation: an expert system, 1984, *IEEE Trans.*, 1984, **PAMI-6**, 555–577.
9. NORONHA, S. J. AND SARMA, V. V. S.	Knowledge-based approaches for scheduling problems: A survey, *IEEE Trans.*, 1991, **KDE-3**, 160–171.
10. VAN MELLE, W., SCOTT, A. C., BENNET, J. S. AND PEARS, M. A. S.	*The EMYCIN Manual*, Report No. HPP-81-16, Heuristic Programming Project, Computer Science Department, Stanford University, 1984.
11. GEVARTER, W. B.	The nature and evolution of the commercial expert system building tools, *IEEE Computer*, May 1987, 24–41.
12. BROWNSTON, L. *et.al.*	*Programming expert systems in OPS5. An introduction to rule-based programming*, 1985, Addison Wesley.
13. FORGY, C. L.	Rete: a fast algorithm for the many pattern/many object pattern matching, *Artif. Intell.*, 1982, **19**, 17–37.
14. ZADEH, L. A.	Knowledge representation in fuzzy logic, *IEEE Trans.*, 1989, **KDE-1**, 89–100.

Appendix I

Actions supported by SRT productions

get: (get < frame-name >< slot-name >)

(get < frame-name >< slot-name >< string >)

This consequent gets the value of a slot from the user. It prints any message, if given along with the frame name and slot name.

set: (set < frame-name >< slot-name >< value>)

(set <frame-name1><slot-name1><frame-name2><slot-name2>)

The value of slot is set equal to the given 'value' in the 1st case, and the value of the second slot in 2nd case.

put: (put < frame-name ><slot-name>) or (put < stringvalue >)

This displays either a message or the value of a slot.

putf: (putf < frame-name >< slot-name >)

(putf <stringvalue>)

This inputs either a message or the value of a slot into a standard file 'putfile'.

add: Syntax is similar to 'set'.

It adds 'value' to slot value and stores the result in the slot. It may also add the content of the 2nd slot to the 1st.

sub: Syntax is similar to 'add'. Subtraction is performed.

mul: Syntax is similar to 'add'. Multiplication is performed.

call: (call < frame-name >)

This action is used to run an external program. The syntax of this action is: The frame name specifies the frame which contains the information about the program to be run. The format of this frame is fixed. The following slots should be present in the same order for the 'call' to be executed:

name: It contains the name of the executable file to be executed.

path: It contains the full path of the file to be executed.

no-of-ip-pars: It indicates the total no. of input parameters to be given to the program to be run.

ip-parl...n: These are the input slots which may be of type real or string. Care has to be taken to ensure that they are in the order required by the program. The external program reads the inputs from these slots.

no-of-op-pars: It indicates the total no. of output parameters to be read from the program to be run.

op-par1 ... n: These are the output slots which may be of type real or string. Care has to be taken to ensure that they are in the order they appear in the output of called program. The external program returns the output values in these slots.

addinto: The syntax is similar to 'set'. It is a set inclusion operator. It operates only on those slots which are of the type 'set of strings' or 'set of real'. The 'value' is added into the slot. It may also add the content of the second slot to the first. The types of the slots should be consistent. It should be noted that since this is a set inclusion operation.

select: (select < Ruleblockno. >)

It performs a select operation, *i.e.*, a new ruleblock selected and only the rules in this block are fired. This is a kind of 'jump' statement.