Assigned: 17.11.2009 Due: 5 Jan. 2010	KRR Written Assignment **PROJECT No.1 - A Nonmonotonic Rule Based System**

Main task

Design and implement a JTMS (Justification based Truth Maintenance System) non-monotonic rule-based system and demonstrate its capabilities for a situation-recognition problem in a domain of your choice.

Context

A rule-based system (RBS) is formed of a set of inference rules called Situation-Action rules or "if-then rules" (**if** Situation **then** Action / Conclusion), a rule interpreter (executor) or Inference Engine (IE), acting on the set of rules, an initial situation which describes the input data of a particular problem instance and a terminal situation or goal which describes the desired results. Each rule that recognizes a Situation leads to an Action executed causing change in situation. The IE will incrementally transform a description of some "initial" situation into a description of some "terminal" situation, by using one rule at a time. The current situation description is held in a Working Memory (WM). The description in the WM consists of a set of descriptions, or facts. The simplest Action is to add a fact to the WM (corresponding thus to a conclusion drawn if the Situation is true).

Knowledge representation

The rules that the RBS should handle are non-monotonic rules or "rules with assumptions" (see KRR Lecture_3). The left hand side (LHS), or "situation recognition" side of a rule can be expressed as a conjunction of arbitrary predicates and assumptions (which are also predicates). The right hand side (RHS), or "action" side can be any action (or sequence of actions; Optional = Bonus).

(Optional = Bonus) The predicates and the actions should be pre-defined. By pre-defined we mean that they are provided in files that are read in by the system (i.e., by the IE), along with the rules, prior to any rule execution, and are not "built into" the IE. This makes the IE independent of any particular domain.

The rules will have the form:

if Condition1 **and** Condition2 **and** … **and M** Assumption

then Action

Here **M** is a special meta-predicate (a nonmonotonic operator) that means "may assume", or "there is no reason to believe the opposite". Assumption may be a predicate or a negated predicate.

Such a rule corresponds to a Reiter's Default Logic rule

Condition1 ∧ Condition2 ∧ … : Assumption / Action

or to a rule

if Condition1 **and** Condition2 **and** … **and ifnot** ~Assumption

then Action

from KRR Lecture_3

Example: Consider the following set of rules:

if C **and M**A **then** D

if B **then** C

if F **then** D

Given an initial belief *Bel* = {B}, we can infer C and D. That is, the set of beliefs, *Bel*, now becomes

Bel = {B, **M**A, C, D}.

Assume next that F and (not A) are added to Bel, i.e.

Bel = {B, **M**A, C, D, F, not A}

contradiction.

To resolve this contradiction we must be able to withdraw assumption **M**A, and revise everything that follows from it.

Here are examples of rules and facts. Yours do not have to follow this exact syntax, you may define your own syntax for the rules.

Rule

if Color(X,red) **and** Shape(X,round) **and M** Fruit(X) **then** Type(X,apple)

In natural language the rule is to be read:

"If the Color of an object X is red and its Shape is round and we may assume that the object is a Fruit then the Type is apple"

Facts

Color(o1,red) Shape(o1,round) Type(o1,nil)

Color(o2,red) Shape(o2,round) Type(o2,nil)

There are two reasons why we want to be able to introduce assumptions:

1. To use indirect proof (prove P, assuming not P);
2. The RBS may not have all the information needed to solve a problem.

This requires the following two problems to be addressed:

1. How to introduce and handle assumptions?
2. How to retract assumptions if they later prove wrong?

An easy way to address the first question is to use a stack where a "local context" is defined once an assumption is pushed on the stack (this is called "a stack-oriented context mechanism"). When this context is no longer needed (the conclusion that required an assumption has been proved) it is popped and the stack now contain only true facts.

The stack-oriented context mechanism, however, does not provide a solution to the second problem.

A more sophisticated way to handle assumptions is provided by the Truth Maintenance Systems (TMS).

Control

A "terminal" or goal is initially given to the RBS. The rule interpreter should use forward chaining of rules to prove the goal (arrive in the "terminal" situation). A goal may have unbound variables that will be instantiated as a result of goal satisfaction, e.g, Type(o1,X)). The control of rule selection and execution works as follows. All of the rules are examined to see if one or more matches the situation. If only one matches then the action (RHS) of that rule is executed. If more than one rule matches, the one that is most specific is used. Specificity is up to you to define, but it probably has something to do with the length of the LHS. If there are two or more rules with the same specificity then the one chosen is the earlier in the rule list. You will have to worry about the same rule firing again in an unchanged situation, and when to stop the system.

The IE will be coupled with a JTMS (there are different sorts of TMS: JTMS – Justification based TMS, ATMS – Assumption based TMS, etc.) that is able to cache inferences performed by the IE, generate explanations, compute the current set of believed assertions, identify contradictions and ask IE to recover from it.

Remember that a Truth Maintenance System (TMS) is a problem solving module responsible for:

1. Enforcing logical relations (constraints) among beliefs
2. Generating explanations for conclusions
3. Supporting default reasoning
4. Identifying causes for failure and ask IE to recover from inconsistencies

1. Enforcing logical relations (constraints) among beliefs

Every AI problem which is not completely specified requires search. Search utilizes assumptions, which may eventually change. Changing assumptions requires updating consequences of beliefs. Re-derivation of those consequences is most often not desirable, therefore we need a mechanism to maintain and update relations among beliefs.

Beliefs can be viewed as propositional variables, and a TMS can be viewed as a mechanism for processing large collections of logical relations on propositional variables.

2. Generating explanations for conclusions

TMS uses cached inferences. The fundamental assumption behind this idea is that caching inferences once is more beneficial than running inference rules that have generated those inferences more than once.

3. Supporting default reasoning

Many real-world problems cannot be completely specified. That is, the system must make conclusions based on incomplete information. Typically the assumption under which such conclusions are drawn is that X is true unless there is evidence to the contrary. This is known as the "Closed-World Assumption" (CWA). Notice that the CWA helps us limit the underlying search space by assuming only a certain choice and ignoring the others. The reasoning scheme that utilizes this assumption is called "default (or non-monotonic) reasoning".

4. Identifying causes for failure and ask IE recover from inconsistencies

Inconsistencies among beliefs in the KB are always possible, especially if the system makes its conclusions based on insufficient information. The most common reasons for inconsistencies or other failures are the following:

- Wrong data. Example: "Outside temperature is 320 degrees."
- Impossible constraints. Example: (Big-house and Cheap-house and Nice-house).
- Non-monotonic inference. The system is forced to "jump" to a conclusion, because of the lack of information, or lack of time to derive the conclusion.
- Contradictions due to inconsistent data, conclusions contradicting the existing data, or inconsistent assumptions.
- Dynamic data. When the domain evolves, the new domain state may be considerably different from the previous domain state, and inferences made in the previous state may no longer be valid.

Cashed dependences among beliefs that TMS maintains help identify the reason for an inconsistency and allows the TMS to recover from it.

In this project, your TMS should be able to perform 1, 2, and 3. Capability 4 is Optional = Bonus.

Input/output

Input the rules and the initial state of the working memory from separate files. The program handles input, output, rule selection and rule execution.

For pre-defined predicates (Optional facility) input the definitions of all predicates and actions from a separate file.

Requirements

This handout is a design and programming assignment.

The first part of the handout must contain a written explanation of your design decisions, including the rule syntax you have chosen (you are required to specify the rule grammar you have chosen for the rule syntax in BNF - Bakus Naur Form) and the capabilities of your system (at least 2 pages, at most how many pages you want).

The second part of the handout will be the program itself, written in a language of your choice. Bonus points will be given for extra facilities or interfaces that comprise more than text based input and output. Use a clean and clear programming style and comment your code.

Your program must be able to do at least the following:

- Display rules
- Set goals, prove goals and show how they are proven
- Add and retract assumptions
- Build the associated JTMS
- Display the justification for a node
- Answer queries concerning belief states: in <belief>, out <belief>, why <belief>, assumptions-of <belief>

The program may be written in a language of your choice. You should provide a Readme file with details about the environment required by your program.

The problem that you are to solve as a demonstration is up to you. A set of about 10 rules will be adequate but fill free to use as many rules as you want to make the application closer to a real-world one!

You should not copy code from books, Web sites, colleagues. The solution must be yours!