A PARALLEL SELF-MODIFYING DEFAULT REASONING SYSTEM

Jack Minker Donald Perlis Krishnan Subramanian
Department of Computer Science
University of Maryland
University of Maryland Institute
for Advanced Computer Studies

College Park, MD 20742

ABSTRACT

As a step in our efforts toward the study of real-time monitoring of the inferential process in reasoning systems, we have devised a method of representing knowledge for the purpose of default reasoning. A meta-level implementation that permits effective monitoring of the deductive process as it proceeds, providing information on the state of the answer procurement process, has been developed on the Parallel Inference System (PRISM) at the University of Maryland. Also described is an implementation in PROLOG (and to be incorporated in the above) of a learning feature used to calculate, for purposes of issuing default answers, the current depth of inference for a query from that obtained from similar queries posed earlier.

Keywords: automated (default) reasoning, learning, cognitive modelling, user interface technology

1. Introduction

In continuing the study of default reasoning in real-time systems [13] we have encountered the phenomenon of *tentative* answers to queries, which may alter as the system continues to search and to perform deductions. The idea underlying this is that for queries having natural default responses when no other response is available, it may be the case that the failure of the reasoning system to respond with a positive answer *quickly* is an indication that no such answer is likely to be forthcoming; in such a case the default answer may be provided, even though the system has not finished all possible lines of reasoning.

To carry out such reasoning, the deductive engine must be monitored so that at any time it is known whether an answer has been returned, allowing a decision as to whether to issue a default conclusion in the absence of an answer. We have implemented a mechanism for this purpose, both in PRISM (a parallel inference system) and in PROLOG.

We state the problem below. Then we describe the approach adopted, and briefly the PRISM system. The implementation is described in section 2. We give examples of application of our methods in section 3, and in section 4 we discuss related future work.

The problem addressed can be stated abstractly as follows: Given an inference engine, we wish to monitor its behavior so that while deductive efforts are in progress, another mechanism can decide when (and whether) to issue default answers based on the (so-far) failure of the original engine to find an answer. That is, our new mechanism will be an interface between the user and the deductive engine. However, the interface is to react in real-time to the real-time behavior of the engine, this being the key to its default conclusions.

This also has ties with human cognitive behavior. When asked a question, such as 'what is Tom's phone number?' we may respond by cogitating, then saying 'I don't know' only later to amend this with 'Wait! Yes, I do know, it's 346-9344.' The possibility of error is explicitly present in such reasoning. For certain queries, it may be inappropriate to conclude the falsity simply because it is not answered quickly, while for others it may not.

As a practical matter, the interface that is to make these decisions can be part of the deductive engine itself; but conceptually it is perhaps more easily regarded as separate. In the next section we describe the operation of the particular mechanisms we have developed. At the present time, we do not have a mechanical procedure in the main system to decide when to employ a default; i.e., defaults are employed at all times if no answer is (so-far) provided by the engine.

Much of the work in default reasoning has been of a theoretic and formal nature, e.g. [7,8,9,10,11,14,16]. We are here concerned with issues involving the practical aspects. The primary motivation is the study of intelligent and parallel question-answering capabilities in computers. Our initial attempt was a simple parallel meta-interpreter, with a desire to examine and study its functioning at modelling human answering behavior. An inference step count exhibits, in some sense, the 'depth' or 'intensity' of the reasoning involved. A dynamic feedback capability keeps the user informed of the status of the inference process, in real time. A simple learning feature has been implemented in PROLOG using depth information from previous queries in the inference for the current query. An exclusive object-level implementation would have yielded a much less flexible system.

PRISM (PaRallel Inference SysteM), developed at the University of Maryland, is the inference engine that we used to exploit parallelism. It employs logic programming techniques and affords explicit control of goals, in an evolving logic programming environment. It is designed to run on ZMOB, the Department's experimental parallel computing system [2,5,15]. Currently, PRISM runs with a software belt that simulates the ZMOB hardware belt.

The PRISM system is an integration of four major subsystems: the Problem Solving Machines (PSMs) that manage the tree of goals, the Intensional Database Machines (IDBs) that contain the general axioms, the Extensional Database Machines (EDBs) that contain fully ground function-free atoms, and the Constraint Machine (CSM) that contains integrity constraints. A host subsystem serves as the interface between the user and PRISM, receiving queries and relaying back answers. PRISM supports goal types by a notation that consists of angle brackets (for sequential, left-to-right execution) and braces (for parallel execution) [15].

A query posed to PRISM system can belong to one of the following categories: (1) A single goal, e.g. G1 or <G1> or {G1}; (2) A list of goals that have to be solved, strictly sequentially, e.g. <G1,G2,G3>; (3) A list of goals, all of which may be solved in parallel, e.g. {G1,G2,G3}; (4) A goal list to be solved basically sequentially, but contains sublists solvable in parallel e.g. <G1,{G2,G3}> , <G1,{G2,<G3,G4>}> etc.; and (5) A list of goals that can be solved basically in parallel, but contains sublists that have to be solved sequentially, e.g. {G1,<G2,G3>} , {G1,<G2,{G3,G4}>} etc.

2. System Description

The basic system centers around parallelism. The key notion is a mechanism that provides default reasoning and comes to decisions rapidly even at the expense of making mistakes, and can revise when to make a default decision based on past performance.

Initially defaults are made as follows. Given that a predicate letter is fully extensional, the system should conclude on lookup either that it has the answer or no answer is possible. However, given a predicate letter that is fully intensional, it should conclude, after the system has gone along the shortest possible path to a solution, either failure or success. As the system progresses it may learn that the (final) solution on the average takes longer (or shorter) than anticipated by the current default specification, obtained as the depth (AID, or actual inference depth) of the and-or tree that corresponds to the inference. After each query, the depth is reestimated for a subsequent query that would involve the same predicate letter. In the case of a predicate letter that is both extensional and intensional, we ignore the extensional possibility and calculate the depth as if it were fully intensional. In fact, we can arbitrarily specify default depth conditions and let the system learn the appropriate value to use. Indeed, this will be seen in some of the experiments that we describe in section 3. One should, however, start with reasonable values rather than arbitrary values since, the system will converge more rapidly to the correct default depth values.

A simple formula is used for the purpose of 'learning' new default depth values:

New depth = (Old depth * N + Latest depth)/(N + 1)

where N is the number of queries (before the current one) that involved the same predicate and latest depth is current AID. Thus at any instant, a predicate has tagged to it a depth estimate and the number of queries thus far posed involving it. Clearly, the more the number of queries, the better the estimate (supposedly indicating better learning). As PRISM hasn't an assert/retract capability as yet, a sequential version has been implemented in PROLOG. When a depth is reached an answer is returned; however, if the answer is negative (nothing found) the system continues to search for an answer in order to find any possible greater depth it 'should' have gone to; this latter is the 'latest depth' in the formula.

We employ two binary predicates to represent all information that a user wishes to supply. One identifies all facts that go into the EDBs, while the other identifies the clauses (facts and rules) placed in the IDBs. These correspond to the object-level placement of knowledge, and encompass all information in the user's programs. This is used by the meta-interpreter (discussed next) that issues the meta-answers, while monitoring the answer-deduction process.

2.1 Design Aspects

The basic structure of the system is shown in Fig. (1). The knowledge base module is exclusive for all user-supplied information, and consists of atomic clauses each of which is identified as belonging either to the EDB or the IDB with the two predicates. This is the only section that is directly relevant to the user, as all meta-level activity is transparent to him.

The second module comprises the inference machinery, responsible for the meta-interpretation activity upon the knowledge base.

In this module, we have maintained a clear distinction between a kernel that exclusively handles all interaction with the user-supplied knowledge, and a layered structure that encompasses the kernel. The latter serves to reduce any demand for an interaction with the knowledge base to elementary level interactions, which the kernel routines can directly act upon. Such a methodology proves to be advantageous for experiments that need a dynamic and evolving inferential structure, particularly in a parallel logic programming environment.

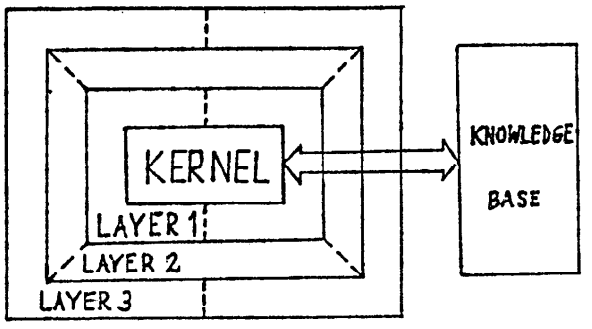


Fig. (1)

A single layer is responsible for a class of queries that would comprise one or more categories. A partition within a layer is responsible for all queries that fall in the same category. A layer may only make use of the facilities available in any layer ensconced within and those at its own level. Such a discipline enables one to construct the entire assemblage (inside-out) in an incremental, structured fashion.

2.2 A Modular, Multi-layered Implementation

Solving a single goal is fundamental to solving any query, and this is the function of the kernel in the inference module. This may be regarded as the elementary level at which any inferencing has to begin. Solving any query, irrespective of the number of goals involved or their nature (i.e. sequential/ parallel), can be reduced to solving single-goals. This isolates the kernel from interactions between the inference assembly and the knowledge base, and proves to be beneficial when one constantly needs to adjust the inference mechanism (to model the answering process of a human reasoner) i.e. it can be iteratively refined till its operation begins to exhibit the intended characteristics. Also, this helps focus attention on a small and compact section that contains but a few routines, needed to meta-interpret a single goal.

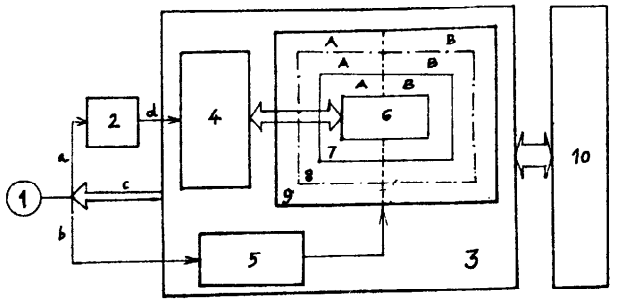
Just as important would be the way a query is reduced to elementary level interactions. The layers that encompass the kernel effect this reduction, which would then require solving single-goals only i.e. exclusive kernel activity. The layer that immediately surrounds the kernel handles the categories 2 and 3. That half of this layer that handles a sequential list hands in a goal to the kernel, and only when the kernel is through with it does it hand in the next goal in the sequence. The other half of the layer that is responsible for the parallel ones, i.e., category 3, causes PRISM to spawn the requisite number of kernels to handle all the goals in the list in parallel. The complete system is shown in Fig. (2).

The kernel is used for solving a single goal, be it a query by itself, or part of one. We view a goal as an ordered pair of the predicate and the argument list. The skeleton kernel as tailored for the answer-behavior model is presented here.

The underlying notion of default reasoning that we are exploiting is implemented in part by the kernel, as follows: We consider that EDB data is readily accessible for immediate retrieval, and that IDB data may take longer to utilize. Thus if a query is such that the system's database would normally be expected to contain an answer to a given query in EDB, such an answer should be forthcoming quickly. If none so appears, the appropriate default assumption is that the query is false. However, this does not rule out the possibility of a later answer being found, that is, as IDB is searched and inferences are made.

These ideas are (partially) illustrated in the following sample axioms from the kernel:

```
(K1) *Solve(Pred,Arglist) <- EDB(Pred,Arglist), Report_Success().
(K2) *Solve(Pred,Arglist) <- Report_Tentative_Failure(),
                             IDB([Pred,Arglist],Matching_Body),
                             Analyze(Arglist,Matching_Body).
(K3) Analyze(Arglist, NIL).
(K4) Analyze(Arglist, Subgoal_List) <- Recurse(Subgoal_List).
(K5) Recurse(NIL).
(K6) Recurse([[Pred1,Arglist1]|Rest_Subgoals]) <-
                             Solve_Aux(Pred1,Arglist1),
                             Recurse(Rest_Subgoals).
(K7) *Solve_Aux(Pred, Arglist) <- EDB(Pred, Arglist).
(K8) *Solve_Aux(Pred,Arglist) <-
                             IDB([Pred, Arglist],Matching_Body),
                             Analyze(Arglist,Matching_body).
```



1. User
2. Preprocessor
3. Meta-interpreter
4. Knowledge Base
5. Dispatcher
6. Kernel
7. Inner Layer
8. Outer Layer - Inner Tier
9. Outer Layer - Outer Tier
10. The PRISM System

A. The half handling sequential queries

B. The half handling parallel queries

a. Data/Programs **b. Queries**

c. Response **d. Internal Format**

Fig. (2)

In the above clauses, ´EDB´ and ´IDB´ are the two predicates used for providing all the user-supplied information. Since the EDB models the short-term memory, it is searched first and only if it fails is the IDB search taken up. The use of the asterisk is imperative in this instance, since in our model we do not want any attempt at the IDB until after the EDB has failed to produce a viable answer. Further we intend to have a success/failure report from the EDB and other status information, only for the top-level goals in the immediate query posed and not for any subgoals. And this necessitates the auxiliary ´Solve_Aux´ procedure above. (The kernel handles category (1) of the queries discussed earlier).

Thus a query P(X) will be solved if it matches an EDB fact, or can be inferred from the IDB. In the latter case, a tentative failure as the default assumption is issued (i.e. if the EDB failed, then in all likelihood the query is false), followed by reports on the IDB search.

3. Examples

We present here two illustrations - one in PRISM and one in PROLOG - in order to give the reader an idea of the answering process capabilities as performed by our model. It seems to us that within the limits of the answering behavior modelled by the meta-interpreter, most real-life questions to a human reasoner would fall, in the majority of the cases in the first category. Only on a few occasions would they fall in the second (the third is different from the second only in the nature of execution, and not in the nature of the query per se), and rarely in the other. As most queries are single goals (i.e. category 1), the kernel captures fairly adequately the answering process simulation, with much lesser activity from the other layers. (The complete testing was done using a genealogy database).

Example 1 :

The meta-answers in the following example should exhibit a flavor of the monitoring activity performed by the system in the process of answering the query.

Short term memory :

´A plane is an air_vehicle. Pup is a light-weight, compact, airy tent. Tarpaulin is tough and compact´.

```
Air_vehicle(Plane).
Tent(Pup).
Light_weight(Pup).
Airy(Pup).
Compact(Pup).
Compact(Tarpaulin).
Tough(Tarpaulin).
```

Long term memory :

´Air-vehicles and shelters are water-proof. Anything that is portable and light in weight is a shelter. A tent is portable. And so is something that is tough and compact´.

```
Water_proof(X) <- Air_vehicle(X).
Water_proof(X) <- Shelter(X).
Shelter(X) <- <Portable(X), Light_weight(X)>.
Portable(X) <- Tent(X).
Portable(X) <- <Tough(X), Compact(X)>.
```

Query: ´Give me something that´s a water_proof and airy shelter´.

<-<Water_proof(X),{Shelter(X), Airy(X)}>.

[This is a sequential query with embedded parallelism. This choice of control has been made for purposes of illustration, but can be motivated in terms of the example by supposing we are *especially* eager for something that is waterproof, after which we are equally concerned about its being a shelter and its airiness].

The Meta-answers :

```
I can´t affirm "Water-proof(X)"; it is false, tentatively.   ( A1 )
Well, here is some answer... Water_proof(Plane).             ( A2 )
I can´t affirm "Shelter(Plane)"; it is false, tentatively.   ( A3 )
I can´t affirm "Airy(Plane)"; it is false, tentatively.      ( A4 )
Well, here is some answer... Water_proof(Pup).               ( A5 )
I can´t affirm "Shelter(Pup); it is false, tentatively.      ( A6 )
I can definitely say that "Shelter(Plane) is false now.      ( A7 )
I can definitely say that "Airy(Plane)" is false now.        ( A8 )
Sure, I can answer this...                                   ( A9 )
here goes... Airy(Pup).                                      ( A10)
I got some answer you want... Shelter(Pup)                   ( A11)
and that´s it...! Here you go!!                              ( A12)
```

The Answers :

! Answer obtained in 10 inference steps ! (A13)

X = Pup (A14)

Query Succeeded. (A15)

Explanation:

First, the query has overall sequential control, with an embedded sublist which has goals amenable to parallel solving. The first goal is taken up, and passed over to the kernel. The Short Term Memory (STM) reports failure (A1). Augmented with the Long Term Memory (LTM), the goal succeeds, and X is bound to "Plane" (A2). Work is begun on the sublist, with X instantiated to "Plane" in both the goals, and at the same time an attempt is made to find an alternative solution for "Water_proof(X)".

Two kernels are spawned simultaneously, and the two take up solving "Shelter(Plane)" and "Airy(Plane)", in parallel. Both fail in the STM, and have the failures reported (A3, A4). However, they continue to work in the LTM. Note that this would not have been possible in a sequential environment, where a goal is not even attempted until the preceding one succeeds. The benefit of the parallel environment would be particularly striking when the user has several mutually non-interdependent goals running, since he can get the status of each and every goal independently of their ordering in the query.

The first goal succeeds again with "Pup" for X (A5), and two more kernels are created for the two goals in the sublist. "Shelter(Pup)" fails in STM, and this is reported (A6). By this time, the two original goals fail in LTM as well (A7, A8), while "Airy(Pup)" succeeds in STM (A9). Accordingly, immediate confirmation of the latter is issued (A10). Eventually, the other kernel also succeeds and reports success, affirming "Shelter(Pup)" (A11). When all the goals are thus solved (A12), the number of inference steps for an answer (if success) (A13) and the final answer (if any) are issued (A14), and the system reports success/failure (A15).

Example 2:

The plots in Fig. (3) show the system asymptotically stabilizing at a certain value of depth for a given predicate. That is, the more the number of queries encountered earlier (with the same predicate), the more representative is its estimate of the inference required for the next query. Recall the formula for calculating the new default depth (at which the effective search is cut off and a negative "closed-world" default is invoked):

New depth = (Old depth * N + Latest depth)/(N + 1)

In effect, a deductive database can over time become more familiar with itself, i.e., with its own particular configuration of data in regard to the likelihood of determining an answer to a particular kind of query within a certain number of inferences. In order to illustrate our idea here, we have chosen three simple examples in which default depths approach an average value. That is, as more queries are entered, the level of inference that is allowed before a default answer is invoked is modified to better represent the average level at which the actual search ended. The database is a variant of that in example 1, in which 'Compact(Pup)' and 'Compact(Tarpaulin)' in the EDB are replaced with 'Light_weight(Tarpaulin)', and the last IDB axiom is replaced with the following three: 'Portable(X) <- Tough(X).', 'Tent(X) <- Tough(X).' and 'WSA(X) <- Water_proof(X),Shelter(X),Airy(X).'. Note that WSA is a new predicate symbol.

In plot (a) we perform repeated queries of 'Water_proof(X)' with an initial depth tag of 1. The plot shows that as queries are answered, the depth tag is repeatedly reset at increased levels, approaching an asymptotic value. This indicates that the initial tag was too low, in that ''experience'' with the database has shown the system there are answers beyond the point at which defaults are invoked. The system gradually corrects this (in the average case) as time goes on. In particular, the user will notice increased cautiousness of the system (and fewer changes of mind). For instance a query of Water_proof(Plane) will initially be answered (negatively) by default after one inference step only to be corrected later (AID: 2). But after three further queries the depth tag becomes > 2, and so the *very same query* will now be answered positively.

In plot (b) the query is WSA(X). The initial depth tag for 'WSA' is 1. We see here that performance changes over time much as in (a).

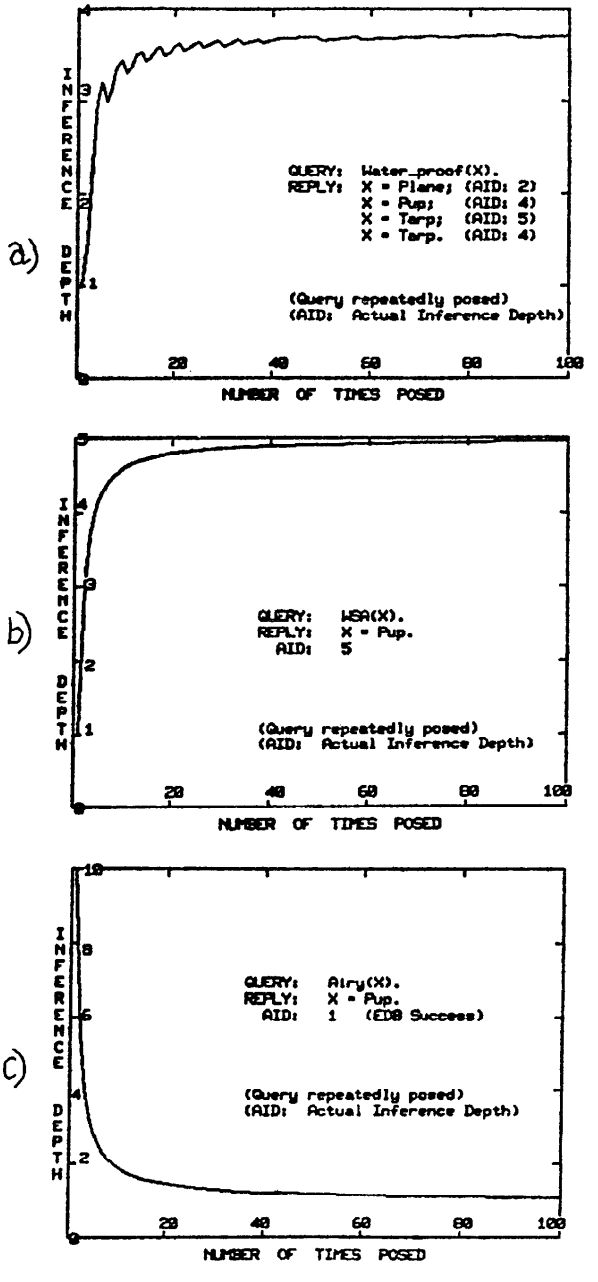


Fig. (3)

In plot (c) the query is Airy(X), which is a purely EDB query. The initial depth tag is set at 10, which is far too cautious. This illustrates the case of it not being known in advance that all the data is EDB and that therefore all queries will terminate rapidly. However, over time, the system adjusts the depth tag to be near 1. Note that if new axioms were later to be added which involved IDB for Airy, then defaults would come into play *as a result of this prior alteration of the depth tag.*

4. Conclusions and Future Work

We have shown that real-time monitoring of query-answering to provide default answers is feasible, at least in a limited context. We have illustrated this with an implementation on the PRISM parallel inference system and partially in PROLOG, using a meta-level approach. This has focussed on the existing structure of PRISM, which has a natural division between look-ups (EDB) and inference steps (IDB); the meta-level allows explicit declarative statments to be made and proven concerning these two notions. Meta-interpretation is the main technique that we have used to implement a flexible system that can evolve easily with changes, to exhibit some rudimentary 'cognitive' or self-modifying behavior.

Future work includes several extensions of the current work. For one thing, we want to pursue the idea of placing a query in background mode once a new query comes in and the original one has not yet finished executing. This would be then enhanced by the inclusion of dynamic proof-tree generation in interactive mode, so that the user can direct the system's behavior as it executes. Additional extensions will include tackling the problem of interacting defaults, providing informative answers, and deciding automatically when a given query is to be treated in default mode or in normal mode.

ACKNOWLEDGEMENTS

This research was supported in part by grants from the following organizations: AFOSR-82-0303, ARO-DAAG-29-85-K-0177, and the Martin Marietta Corporation.

REFERENCES

(1) Bowen,K.A. & Kowalski,R.A.,"Amalgamating Language and Meta-language in Logic Programming" in "Logic Programming",eds. Clark,K.L. & Tarnlund,S.A., Academic Press, London and N.Y., 1982.

(2) Chakravarthy,U.S. et al., "Logic Programming on ZMOB: A Highly Parallel Machine", Procs. Intl' Conference on Parallel Processing, 1982.

(3) Dincbas,M. & le Pape,J-P., "Meta-control of Logic Programs in METALOG", Procs. Intl' Conference on Fifth Generation Systems, ICOT, 1984.

(4) Gallaire,H. & Lasserre,C., "Meta-control for Logic Programs" in "Logic Programming",eds. Clark,K.L. & Tarnlund,S.A., 1982.

(5) Kasif,S., Kohli,M. & Minker,J., "PRISM - A Parallel Inference System for Problem Solving", Procs. IJCAI, 1983.

(6) Kowalski,R.A., "Logic for Problem Solving", Elsevier Science Publishing Co., 1979.

(7) Lifschitz,V., "Some results on circumscription", Workshop on Non-monotonic Reasoning, (Mohonk) 1984.

(8) McCarthy,J., "Circumscription: A form of Non-monotonic Reasoning", AI Journal v.13, 1980.

(9) McCarthy,J., "Applications of Circumscription to Formalizing Common Sense Knowledge", Workshop on Non-monotonic Reasoning, 1984.

(10) McDermott,D. & Doyle, J., "Non-monotonic Logic", AI Journal v.13, 1980

(11) Minker,J. & Perlis,D., "Protected Circumscription", Workshop on Non-monotonic Reasoning, 1984.

(12) Minker,J. & Perlis,D., "Computing Protected Circumscription", Journal of Logic Programming v.4, 1985

(13) Perlis,D., "Non-monotonicity and Real-time Reasoning", Workshop on Non-monotonic Reasoning, 1984.

(14) Perlis,D. & Minker,M., "Completeness Results for Circumscription", AI Journal (to appear).

(15) PRISM Reference Manual, Dept. of Computer Science, Univ. of Maryland, 1983

(16) Reiter,R., "A Logic for Default Reasoning", AI Journal v.13, 1980

(17) Sterling,L., "Expert System = Knowledge + Meta-interpreter", Weizmann Institute of Sciences, Rehovot, Israel, 1984.