# An Overview of Meta-Level Architecture

by

Michael R. Genesereth
Stanford University
Computer Science Department
Stanford, California 94305

Abstract: One of the biggest problems in AI programming is the difficulty of specifying control. Meta-level architecture is a knowledge engineering approach to coping with this difficulty. The key feature of the architecture is a declarative control language that allows one to write partial specifications of program behavior. This flexibility facilitates incremental system development and the integration of disparate architectures like demons, object-oriented programming, and controlled deduction. This paper presents the language, describes an appropriate, and discusses the issues of compiling. It illustrates the architecture with a variety of examples and reports some experience in using the architecture in building expert systems.

## 1. Introduction

The actions of most Artificial Intelligence programs can be divided into distinct "base-level" and "meta-level" categories. Base-level actions are those which when strung together achieve the program's goals. Meta-level actions are those involved in deciding which base-level actions to perform. For example, in the Blocks World, all physical movements of a robot arm would be base-level actions, and all planning would be meta-level. In a data base management system, data base accesses and modifications would be base-level actions, and all query optimization would be meta-level. In an automated deduction system, inference steps would be base-level, and the activity of deciding the order in which to perform inference steps would be meta-level.

A good way of keeping this distinction clear is to view the base-level and meta-level components as separate agents, as suggested in figure 1. The base-level agent perceives and modifies a given environment. The meta-level agent operates in an enlarged environment that includes the base-level agent. Its goal is to observe the base-level agent, decide the ideal actions for it to perform in order for it to achieve its goals efficiently, and finally to modify the base-level agent so that it performs those actions.

The role of the meta-level agent is similar to that of an "instruction-fetch unit" in many computer systems. The primary difference is that for AI programs the instruction-fetch can be quite complex. Meta-level architecture is a knowledge engineering approach to coping with this complexity.

The central idea in meta-level architecture is the use of a declarative language for describing behavior. Within this language one can write sentences about the state of the base-level agent, its actions, and its goals; and the meta-level agent can reason with these sentences in deciding the ideal action for the base-level to perform.

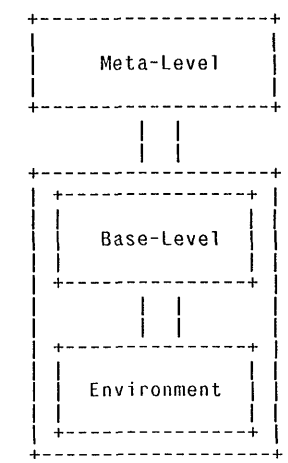


Figure 1 - Relationship of base-level and meta-level

A key feature of the control language is that it allows partial specifications of behavior. One advantage of this is incremental system development. For example, one could start from a (possibly search-intensive) logic program and incrementally add control information until one obtains a (possibly search-free) traditional program. Another advantage is the integration of different AI architectures, e.g. demons, object-oriented programming, and controlled deduction. In the absence of a language in which partial specifications of behavior can be expressed, many arbitrary decisions must be made in implementing an architecture, thus inhibiting integration. A partial specification language allows an architectural idea to be formalized without this arbitrary detail.

The idea of meta-level architecture is not a new one. McCarthy described a similar idea in 1959 [McCarthy]; and, over the years a number of other researchers [Brown, Kowalski] [Davis] [Doyle 1980] [Gallaire, Lasserre], [Hayes 1973] [Sandewall] [B. Smith] [Warren] [Weyhrauch] have followed suit. However, to date explicit introspective reasoning has met with little serious application. This

paper is concerned with the problems of putting the idea of meta-level architecture to practical use.

Section 2 describes the control language and shows how it can be used to encode various other architectures. Section 3 presents an interpreter for the language, and section 4 discusses the issues of compiling. The conclusion describes some experience using meta-level architecture, mentions some directions for future work, and summarizes the key points of the paper. This paper is an abridged version of a previous paper [Genesereth, Smith].

## 2. Control Language

In this paper the syntax of predicate calculus is used for writing meta-level axioms, though any other language with equivalent expressive power would do as well. Several syntactic conventions are used to simplify the examples. Upper case letters are used exclusively for constants, functions, and relations. Lower case letters are used for variables. All free variables are universally quantified.

When the base-level agent is a data base program, care must be taken to avoid confusion between base-level and meta-level expressions. In the examples below, expressions in the base-level language itself are designated by enclosing quotation marks, e.g. "FATHER(A,B)". The examples are sufficiently simple that variables occurring within quotation marks can be assumed to be meta-variables, i.e. they range over expressions in the base-level language.

The vocabulary of the language consists of the logical symbols of predicate calculus, a small application-independent vocabulary of actions, and all the application-dependent symbols necessary to describe the base-level agent and its environment. The vocabulary of actions is listed below. In the descriptions of each symbol, the word "task" is used to refer to an individual occurrence of an action, e.g. the printing of a file or the computation of a sum. If an action is performed repeatedly, each occurrence is considered a separate task, even if the arguments are exactly the same.

OPR(<k>) designates the operation of which the task <k> is an instance.

IN(<i>,<k>) designates the <i>th input to the task <k>.

OUT(<i>,<k>) designates the <i>th output of the task <k>.

BEG(<k>) designates the start time of the task <k>.

END(<k>) designates the stop time of the task <k>.

TIME(<t>) states that <t> is the current time.

EXECUTED(<k>) states the the task <k> has taken place or definitely will take place.

RECOMMENDED(<k>) states that task <k> is the recommended action for a program to take. During inference, it's typical for more than one task to be applicable, and user-supplied ordering information can be used to determine which of these is recommended. If more than one task is recommended, it can be assumed that they are all equally good.

While this small vocabulary allows one to formally specify a wide variety of control structures, it isn't particularly perspicuous. Expressive power and expressive facility aren't always compatible. On the other hand, the language does provide a foundation for more perspicuous languages, and control specifications expressed within these languages can be automatically re-expressed in terms of the vocabulary above and thereby intermingled.

## 3. Interpreting Control Specifications

An interpreter for this language can be written in LISP as shown in figure 2. Each time around the loop, the program updates the time and computes a subroutine for the base level to perform. The subroutine is then executed, the execution is noted, and the cycle repeats.

```
(DEFINE SCHEDULER ()
  (REPEAT (SETQ TIME (FIND 't '(TIME t)))
          (UNASSERT '(TIME ,TIME))
          (ASSERT '(TIME ,(+ TIME 1)))
          (SETQ TASK (FIND 'z '(RECOMMENDED z)))
          (APPLY (GETOPR TASK) (GETARGS TASK))
          (ASSERT '(EXECUTED ,TASK))))
```

Figure 2 - An interpreter for meta-level architecture

In deciding on a base-level action, SCHEDULER uses the inference procedure FIND to obtain a binding for the variable z that makes the proposition (RECOMMENDED z) true. This is the only way in which the meta-level is called; and so, to be useful, all control axioms must contribute either directly or indirectly to this deduction. A good choice of inference procedure for the axioms in this paper would be some version of forward or backward chaining, with a nonmonotonic treatment of NOT. Efficiency could be enhanced through the use of caching [Lenat] to avoid recomputation and truth maintenance [Doyle 1979] to handle changes in state.

In order for FIND to draw proper conclusions from the control axioms, all information about the state of the base-level agent and its environment must be correct. There are several practical ways of keeping this information up-to-date. One approach is for the meta-level to have an explicit model of the base-level program and its environment. To do this, it must have a description (axiomatization) of the initial state of the world and a description of all the effects and non-effects of the base-level actions. After each action the meta-level can use these axioms to update its model of the world. A second approach is to equip the meta-level component with sensory capabilities and allow it to examine the base-level when necessary. For example, if the meta-level needs to know whether a proposition is in the base-level's data base, it can be given a sensory action which examines the base-level data base. The appropriate sensory action for each proposition can be determined at system-building time and can be recorded by an appropriate meta-meta-level procedural attachment (or "semantic attachment" as described in [Weyhrauch]). A third approach is to modify

the actions of the base-level so that they automatically update the data base of the meta-level. The advantage of this approach is that the addition of facts to the meta-level data base can trigger whatever demons depend on them. The choice of approach to use in a given application depends on the domain and the type of control axioms one intends to write. In many situations a combination of approaches is best.

## 4. Examples

The examples in this section illustrate how a variety of different control structures can be implemented within meta-level architecture. The examples are not meant to be complete, only suggestive. For more detail, the reader should see [Genesereth, Smith].

### 4.1 Demon Systems

The characteristic control feature of the blackboard architecture [Erman, Lesser] is the use of demons, or "knowledge sources", to specify program activity. A demon consists of a set of trigger conditions and an action. Whenever the data base, or "blackboard", satisfies the trigger conditions for a demon, its action becomes applicable. The system then selects an applicable action and executes it.

The encoding of demons in a meta-level architecture is quite simple. The action associated with the demon is given a name, and its conditions are expressed on the left hand side of a corresponding applicability axiom. The axioms below are the control axioms for a simple consultation system. Axiom A1 states that, if a proposition is proved and there is a base-level axiom that mentions it on the left hand side, the system should forward chain on that rule. The second axiom states that, if there is a proposition to be proved and there is a rule that mentions it on the right hand side, then the system should backchain on that rule. The third axiom states that, if there is a proposition to be proved, it is okay to ask the user so long as the proposition is "askable".

```
A1: PROVED(p) & INDB("p=>q")
      => APPLICABLE(ADDINDB(q,p,"p=>q"))
A2: WANTPROVED(q) & INDB("p=>q")
      => APPLICABLE(ADDGOAL(p,q,"p=>q"))
A3: WANTPROVED(q) & ASKABLE(q)
      => APPLICABLE(ASK(q))
```

Axiom A4 below guarantees that every applicable demon is recommended. For applications where a demon should be run only once after being triggered, axiom A4 can be replaced by axiom A5.

```
A4: APPLICABLE(k) => RECOMMENDED(k)
A5: APPLICABLE & NOT EXECUTED(k) => RECOMMENDED(k)
```

### 4.2 Search Control

In many AI programs it is common for more than one task to be applicable at each point in time. Axiom B1 shows how ordering axioms can be used in determining which applicable task is recommended. It states that an applicable task is recommended only if no other applicable task is "preferred" to it.

```
B1: APPLICABLE(k) &
    NOT(Ex APPLICABLE(x) & PREFERRED(x,k))
      => RECOMMENDED(k)
```

The axioms below are some examples of how this capability might be used. Axiom B2 constrains a program to perform all backward chaining before asking its user any questions. Axiom B3 states that a program should use rules of greater certainty before rules of lesser certainty. Axiom B4 states that, whenever a program has a choice of backchaining tasks to perform, it should work on the one with fewer solutions.

```
B2: OPR(k1)=ADDGOAL & OPR(k2)=ASK
      => PREFERRED(k1,k2)
B3: OPR(k1)=ADDGOAL & OPR(k2)=ADDGOAL &
    CF(IN(3,k1))>CF(IN(3,k2))
      => PREFERRED(k1,k2)
B4: OPR(k1)=ADDGOAL &
    NUMOFSOLNS(IN(1,k1))<NUMOFSOLNS(IN(1,k2))
      => PREFERRED(k1,k2)
```

The axioms below show how depth-first and breadth-first search can be described. Axioms B5 and B6 define the depth of a goal in terms of its "distance" from an initial goal. Axiom B7 expresses a preference for deeper goals and hence implements depth-first search. Axiom B8 expresses a preference for shallow goals and hence implements breadth-first search.

```
B5: DESIRE(g) => DEPTH(g)=0
B6: SUBGOAL(g1,g2,e,j) & DEPTH(g2)=n
      => DEPTH(g1)=n+1
B7: DEPTH(IN(1,k1)) > DEPTH(IN(1,k2))
      => PREFERRED(k1,k2)
B8: DEPTH(IN(1,k1)) > DEPTH(IN(1,k2))
      => PREFERRED(k2,k1)
```

Obviously these examples do not exhaust the space of possibilities. Breadth-tapering, quiescence, and a large variety of other search techniques can be specified in similar fashion.

### 4.3 Traditional Programming

In certain situations it is desirable to specify control in procedural terms. The vocabulary below, along with the vocabulary for tasks, allows the description of incompletely specified programs (having arbitrary concurrency).

FIRST(<k1>,<k>) states that the task <k1> is the first subtask of the composite task <k>.

CPATH(<k1>,<k2>) means that there is a direct control path from task <k1> to task <k2>. After <k1> is executed, <k2> becomes executable.

LAST(<kn>,<k>) states that task <kn> is the last subtask of task <k>. Due to control branching, a composite task may have more than one last subtask.

PRIMITIVE(<op>) states that the action <op> is a base-level subroutine.

As an example, consider a simple blocks world program to build a tower of three blocks. A LISP version is shown below.

```
(DEFUN TOWER (X Y Z)
  (PUTON Y Z)
  (PUTON X Y))
```

Using the vocabulary above, this program can be specified as shown below. The axiom states that an action K is an instance of TOWER if it has substeps K1 and K2 as shown. One should remember in looking at this description that its size is due to the fact that all control information is stated explicitly. This explicit statement is what enables the partial description of programs. Such partial descriptions are not possible in traditional programming languages because much of the control and data flow is implicit in the code.

```
OPR(k)=TOWER <=>
      E k1,k2
        OPR(k1)=PUTON & IN(1,k1)=B & IN(2,k1)=C &
        OPR(k2)=PUTON & IN(1,k2)=A & IN(2,k2)=B &
        CPATH(k1,k2) & FIRST(k1,k) & LAST(k2,k)
```

The axioms that enable the execution of this code are shown below. With these axioms the substeps of the TOWER program would be executed in the proper order. For example, if the task of building a tower of blocks A, B, and C became applicable, axiom C1 would make the call to PUTON with arguments B and C applicable. Then by axioms C2 and C5 it would be recommended as well. Once that is executed, the call to PUTON with arguments A and B would become applicable by axiom C3. Axiom C4 guarantees that when all of the subactions of a procedure are executed, the procedure is also considered executed.

```
C1: APPLICABLE(k) & FIRST(k1,k) => APPLICABLE(k1)
C2: APPLICABLE(k) & PRIMITIVE(k) & ~EXECUTED(k)
      => RECOMMENDED(k)
C3: CPATH(k1,k2) & EXECUTED(k1) => APPLICABLE(k)
C4: EXECUTED(kn) & LAST(kn,k) => EXECUTED(k)
C5: PRIMITIVE(PUTON)
```

### 4.4 Object-Oriented Programming and Procedural Attachment

From the point of view of implementation, the primary difference between traditional programming and object-oriented programming lies in the way one's code is organized. In traditional programming the code to perform an operation on an object is associated with the operation and is usually conditional on the object or its type. In object-oriented programming the code is associated with the object, its class or some superclass, and it is usually conditional on the type of operation involved. This orientation allows one to interpret objects as active agents and operations as the results of passing messages to those agents.

A typical application is in the world of graphics. One can define each desired shape as a distinct type of object, each with its own methods for redisplay, rotation, panning, etc. The specific subroutine for each object-operation pair can be recorded by writing meta-level axioms. For example, the sentences below specify the subroutines for manipulating squares.

```
D1: SQUARE(x) => TODO(DISPLAY,x,DRAWSQUARE)
D2: SQUARE(x) => POLYGON(x)
D3: POLYGON(x) => TODO(ROTATE,x,ROTATEPOLYGON)
D4: POLYGON(x) => PLANAR(x)

D5: PLANAR(x) => TODO(PAN,x,PANPLANE)
```

Note that this formulation is a slight generalization of object-oriented programming. First of all, subroutines are not associated primarily with objects or operations but rather with operation-object tuples. Secondly, the handling of multi-argument operations is simpler than in pure object-oriented programming. Axioms D6 and D7 operationalize the above axioms.

```
D6: APPLICABLE(g(x)) & TODO(g,x,f)
      => APPLICABLE(f(x))
D7: APPLICABLE(k) & ~EXECUTED(k)
      => RECOMMENDED(k)
```

### 4.5 Planning

The fundamental characteristic of meta-level architecture is that it allows a programmer to exert an arbitrary amount of control over a program's operation. The inference procedures defined above exhibit a moderate amount of programmer control. Programming clearly represents an extreme in which the programmer makes all of the decisions. Planning represents the other extreme in which the programmer specifies only the program's goal and the program must decide for itself what actions to take to achieve that goal.

The axioms for a planning program consist of those for object-oriented programming together with those for traditional programming. The difference from object-oriented programming is the that the computation of the TODO relation is more elaborate. Once a plan has been found, the traditional programming axioms are necessary to execute it.

## 5. Compiling Control Specifications

The primary advantage meta-level architecture is that it allows one to give a program an arbitrary amount of procedural advice. In some cases this can lead to substantial (possibly exponential) savings in runtime. Unfortunately, there is a cost in using a meta-level architecture. Meta-level reasoning can be at least as expensive as reasoning about an application area; and even when this reasoning is minimal, there is some overhead. The problem is particularly aggravating in situations where meta-level control yields no computational savings.

One way to enhance program efficiency is to rewrite the control axioms in a form that lessens meta-level overhead. Given an appropriate inference procedure for the meta-level, the least expensive control structure is that of a traditional program. Therefore, the ideal is to reformulate an axiom set as a traditional program. This can frequently be done, if information about the application area is used.

A more drastic approach to increasing the efficiency of a meta-level architecture is to eliminate meta-level processing whenever possible, e.g. in the execution of a fully specified deterministic program. Given an adequate set of base-level actions, any procedure described at the

meta-level can be "compiled" into a base-level program. This sort of compilation is more complex than that done in logic programming systems like PROLOG because the compiler must take into account arbitrary control specifications.

The possibility of partial compilation is particularly interesting. In some cases it is possible to compile part of a program into base-level subroutines while retaining a few meta-level "hooks". A good example of this is an interpreter for an object-oriented programming language. The code for each subroutine can be written as shown below.

```
(DEFINE DRAW (X) (FINDMETHOD 'DRAW X))

(DEFINE ROTATE(X) (FINDMETHOD 'ROTATE X))

(DEFINE PAN (X) (FINDMETHOD 'PAN X))

(DEFINE FINDMETHOD (OP ARG)
  (CALL (FIND 'z '(TODO ,OP ,ARG z)) ARG))
```

Each subroutine "traps" to the meta-level to find out which specific subroutine to use. Answering this question may involve inheritance over a type hierarchy or a more general inference. However, it is more efficient than the standard meta-level architecture because there is a built in assumption that only one subroutine is relevant. Furthermore, the trap can be omitted where the flexibility of meta-level deduction is unnecessary.

## 6. Conclusions

The feasibility of meta-level architecture for practical programming has been tested by its use in the construction of a number of different systems. These include an automated diagnostician for computer hardware faults (DART [Genesereth 8/82]), a calculus program (MINIMA [Brown]), a simulator for digital hardware (SHAM [Singh]), a mini-tax consultant (IDWONNA [Barnes, Joyce, Tsuji]), and an implementation of the NEOMYCIN infectious disease diagnosis/tutoring system [Bock, Clancey]. All of these programs were built with the help of MRS [Genesereth, Greiner, Smith] [Clayton] [Genesereth 11/82], a knowledge representation system aimed at facilitating the construction of meta-level programs.

The work on DART best illustrates the way in which one uses meta-level architecture. The first step was to build a data base of facts about digital electronics, e.g. the behavior of an "and" gate, and the circuit to be diagnosed, e.g. its part types and their interconnections. A general inference procedure (linear input resolution) was then applied to these propositions to generate tests. In order to enhance efficiency, a variety of control axioms were then added. Finally, many of the axioms were compiled (partly by hand) into the resolution procedure.

While the architecture described here answers many of the questions concerned with the practical use of meta-level control, there remain enormous opportunities for significant further research. The most important of these is the development of a compiler able to translate arbitrary control specifications into reasonable LISP code. Also important is the discovery of powerful domain-independent control rules, e.g. the ordering of conjuncts in problem solving, the choice of problem solving method, knowing when to cache, etc. Finally, the architecture needs to be extended to the handling of multiple agents.

In summary, the purpose of this paper is to present the details of a practical meta-level architecture for programs. A program written in this style includes a set of base-level actions and a set of meta-level axioms that constrains how these actions are to be used. The meta-level axioms are written in an expressive meta-level language and are used by a general inference procedure in computing the ideal action to perform at each point in time. The architecture is sufficiently general that a wide variety of traditional control structures can be written as sets of meta-level axioms. While this generality engenders a certain amount of inefficiency in any straightforward implementation, a variety of specialized techniques can be used to avoid inordinate overhead. Finally, experience with meta-level architectures indicates that it makes programs more aesthetic, simpler to build, and easier to modify.

## Acknowledgements

Over the past two years we have benefitted by interactions with numerous people. In particular, Jon Doyle, Pat Hayes and Doug Lenat have had considerable impact on our thinking. Special thanks to Danny Bobrow, Bill Clancey, Lee Erman, and Rick Hayes-Roth for comments on earlier drafts of this paper. Support for this work was provided by ONR contract N00014-81-K-0004.

## References


T. Barnes, R. Joyce, S. Tsuji: "A Simple Income Tax Consultant", Stanford University Heuristic Programming Project, 1982.

J. A. Barnett, L. Erman: "Making Control Decisions in an Expert System is a Problem-Solving Task", USC/ISI Technical Report, April 1982.

C. Bock, W. J. Clancey: "MRS/NEOMYCIN: Representing meta-control in predicate calculus", HPP-82-31, Stanford University Heuristic Programming Project, December 1982.

D. Brown: "MINIMA", Teknowledge Inc., 1982.

K. Brown, R. Kowalski: "Amalgamating Language and Metalanguage in Logic Programming", *Logic Programming*: K. Clark, S. Tarnlund (eds), Academic Press, New York, 1982, pp 153-172.

J. Clayton: "Welcome to the MRS Tutor!!!", HPP-82-33, Stanford University Heuristic Programming Project, November 1982.

R. Davis,: "Meta-Rules: Reasoning about Control", *Artificial Intelligence*, Vol. 15, 1980, pp 179-222.

J. Doyle: "A Truth Maintenance System", *Artificial Intelligence*, Vol. 12, 1979, pp 231-272.

J. Doyle: "A Model for Deliberation, Action, and Introspection", TR-581, M.I.T. Artificial Intelligence Laboratory, May 1980.

L. Erman, V. Lesser: "A Multi-Level Organization for Problem-Solving Using many Diverse, Cooperating Sources of Knowledge", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975, pp 483-490.

H. Gallaire, C. Lasserre: "Metalevel Control for Logic Programs", *Logic Programming*: K. Clark, S. Tarnlund (eds), Academic Press, New York, 1982, pp 173-185.

M. R. Genesereth, R. Greiner, D. E. Smith: "The MRS Dictionary", HPP-80-24, Stanford University Heuristic Programming Project, November, 1982.

M. R. Genesereth: "Diagnosis Using Hierarchical Design Models", *Proceedings of the National Conference on Artificial Intelligence*, August 1982.

M. R. Genesereth: "An Introduction to MRS for AI Experts", HPP-82-27, Stanford University Heuristic Programming Project, November, 1982.

M. R. Genesereth, D. E. Smith: "Meta-Level Architecture", HPP-81-6, Stanford University Heuristic Programming Project, December, 1982.

P. Hayes: "A Logic of Actions", *Machine Intelligence*, Vol 6, American-Elsevier, New York, 1970, pp 533-554.

P. Hayes: "Computation and Deduction", *Proceedings of the Symposium on Mathematical Foundations of Computer Science*, Czechoslovakian Academy of Sciences, 1973, pp 105-117.

K. Konolige: "A First Order Formalization of Knowledge and Action for a Multiagent Planner", Technical Note 232, SRI International Artificial Intelligence Center, December, 1980.

R. Kowalski: "Algorithm = Logic + Control", *Communications of the ACM*, Vol. 22, 1979, pp 424-436.

D. B. Lenat, F. Hayes-Roth, P. Klahr: "Cognitive Economy", HPP-79-15, Stanford University Heuristic Programming Project, June 1979.

J. McCarthy: "Pograms with Common Sense", *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*. H. M. Stationery Office, London, 1960. Reprinted in *Semantic Information Processing*, M. Minsky (Ed), MIT Press, Cambridge, 1968, pp 403-410.

J. McCarthy: "First Order Theories of Individual Concepts and Propositions", in J. Hayes, D. Michie, L. Mikulich (eds), *Machine Intelligence 9*, Ellis Horwood, Chichester, 1979, pp 120-147.

C. Rich: "Inspection Methods in Programming", AI-TR-604, M.I.T. Artificial Intelligence Laboratory, June, 1981.

E. Sandewall: "Ideas about Management of LISP Data Bases", Working Paper 86, M.I.T. Artificial Intelligence Laboratory, January 1975.

N. Singh: "SHAM: A Hierarchical Simulator for Digital Circuits", Stanford University Heuristic Programming Project., 1982.

B. Smith: "Reflection and Semantics in a Procedural Language", TR-272, M.I.T. Artificial Intelligence Laboratory, January 1982.

D. E. Smith, M. R. Genesereth: "Ordering Conjuncts in Problem Solving: Serious Applications of Meta-Level Reasoning I.", HPP-82-9, Stanford University Heuristic Programming Project, April 1982.

D. E. Smith, M. R. Genesereth: "Finding All the Solutions to a Problem: Serious Applications of Meta-Level Reasoning II.", HPP-83-21, Stanford University Heuristic Programming Project, December 1982.

M. Stefik: "Planning and Meta-Planning", *Artificial Intelligence*, Vol. 16, 1981, pp 141-170.

W. VanMelle, A. Scott, J. Bennett, M. Peairs: "EMYCIN Manual", HPP-81-16, Stanford University Heuristic Programming Project, October 1981.

D. Warren: "Efficient Processing of Interactive Relational Database Queries Expressed in Logic", Proceedings of the seventh VLDB conference, 1981.

R. W. Weyhrauch: "Prolegomena to a Theory of Mechanized Formal Reasoning", *Artificial Intelligence*, Vol 13, 1980, pp 133-170.

R. Wilensky: "Meta-Planning: Representing and Using Knowledge about Planning in Problem Solving and Natural Language Understanding", *Cognitive Science*, Vol. 5, 1981, pp 197-234.