→ Newell

DRAFT: not for quotation or distribution
April 1986

Making Expert Systems Explicit

John McDermott
Department of Computer Science
Carnegie-Mellon University
Pittsburgh Pennsylvania 15213

To be presented at IFIP Congress '86

Abstract

First generation expert systems have demonstrated that artificial intelligence (AI) techniques can be used to advantage in solving a variety of domain-specific problems which people appear to solve using knowledge gathered during years of experience. But these systems gain their substantial power by accepting very strong limitations on their scope. Work on expert systems during the next several years will surely focus, in part, on ways in which the scope can be broadened without sacrificing power. This paper argues that a promising path to that goal involves first gaining a better understanding of the various problem solving methods that are proving effective in different domains, then exploiting this understanding to develop better knowledge acquisition tools, and finally developing cooperating problem solvers that collectively have much greater scope than any present day expert system.

1. Aspirations expert systems should have

The purpose of this paper is to discuss how the early successes with expert systems can be used as a base on which to build more robust systems. In this paper the label "expert system" will be restricted to a program with the following characterisitics:

- The scope of its task is limited to a clearly bounded domain where almost all of the knowledge required to perform the task is special to that domain (ie, little common sense knowledge is required).
- A single problem-solving method is for the most part adequate to deal with the task.
- At almost every step, the problem-solver must choose among a number of alternative actions.
- Much of the problem solver's knowledge is relevant only infrequently.

Several programs satisfying this definition have been developed, and the abilities of these programs, in some cases at least, has rivaled the ability of human experts. To the extent that this is surprising, the source of the surprise is probably (1) that so much competance can be displayed by a system whose knowledge is so narrow, and (2) that so much compentance can be displayed by a system with just a single problem solving method. The surprise is mitigated, of course, by the penchant of these systems to occasionally fail irreparably.

If we ask what it is that allows an expert system to exhibit irreparable failure, there are, at a

sufficiently high level of abstraction, only two possibilities: (1) inadequacies in its knowledge or (2) inadequacies in its problem solving method (ie, an inability to make appropriate use of knowledge it may or may not have). The first possibility can be immediately ruled out since the repair is simply to add more knowledge. Thus the problem must lie with the fact that an expert system has only a single (not very general) problem solving method -- a method that may very well be totally inappropriate at some of the boundaries of its task. Irreparable failure occurs because the knowledge required for some part of a system's task is incommensurate with its problem solving method.

If the principal vulnerability of current generation expert systems is that each uses only a single problem solving method, one would expect the builders of next generation expert systems to focus primarily on fixing this problem. The ultimate fix, of course, is to create expert systems that invent new methods or adapt existing methods as the need manifests itself. But a likely path to that fix would seem to lead through an expert system which, though it cannot invent problem solving methods, can at least easily accommodate a new method as its developer recognizes the need for it; experience with such systems will help identify the cues that signal a missing method and the process by which a new method can be invented. Further back on the path is a collection of loosely coupled expert systems, each with a single problem solving method, but with enough awareness of each other to know when and how to pass problems on; experience with such systems will help us understand how sets of problem solving methods can interact, how much they have to know about one another, how they can share knowledge, how they can effectively communicate. Finally, at the beginning of the path is the task of identifying the knowledge requirements of a variety of problem solving methods; until this is done, trying to create collections of cooperating expert systems makes little sense.

We are just beginning down this path. Current expert systems research is very much focused on getting clear about what kinds of knowledge are required for various task types. Several researchers have been stressing that to maintain and continue to develop an expert system it is critical to identify the various roles its knowledge will play and to represent its knowledge in a way that does not conflate these roles [Chandra 83, Clancey 83, Neches 84]. But most current generation expert systems still suffer from vagueness with respect to what their methods really are, and as a consequence, it has been very difficult to say anything meaningful about the knowledge requirements of those methods.

A promising approach to making the knowledge requirements of methods explicit is to develop knowledge acquisition tools. A knowledge acquisition tool is a program that knows how to represent the knowledge a problem solving method requires and that knows how to diagnose what knowledge a problem solving method is likely to be missing. Examples include TEIRESIAS [Davis 82], ETS [Boose 84], and MORE [Kahn 85]. There are two ways in which the development of such tools will help identify the knowledge requirements of problem solving methods: (1) The tools will not be able to elicit knowledge unless they understand the roles the knowledge will play. (2) The tools will make the development of expert systems sufficiently easy so that enough data points will be able to be generated to uncover all of the knowledge requirements of a method.

One would expect these two considerations to be sufficient to bring about the creation of a significant number of knowledge acquisition tools. But if they should turn out not to be sufficient, there is fortunately an external consideration which also encourages the creation of such tools. By

now, some of the earliest expert systems have grown to a point where they are becoming difficult to maintain. R1 [McDermott 82], for example, now has over 4000 rules in its knowledge-base, and because its problem solving method is not very well defined, the scope and import of the knowledge in each rule is much less clear than it could otherwise be [Bachant 84]. It is extremely likely that a number of knowledge acquisition tools will be developed just to insure that the systems they can be used to build will remain maintainable.

2. Three data points

Three knowledge acquisition tools being developed at Carnegie-Mellon University give a flavor of what is becoming possible: SALT [Marcus 85], MOLE [Eshelman 86], and SEAR [vandeBrug 86]. For each of these tools, we will discuss:

- The problem solving method it presupposes.
- Knowledge representation expectations that need to be made explicit.
- How the problem solving method helps with the identification of missing knowledge.
- The current status of the tool.
- How systems built with the tool might fail irreparably.

2.1. SALT

The problem solving method SALT presupposes assumes that the task is to create a plan (eg, a design for an electro-mechanical system, a schedule for a flow shop); it further assumes that if a plan is determined to be inadequate (either while it is being developed or while it is being implemented), then the difference between the inadequate plan and an acceptable plan can be used to select a fix. The method presupposed by SALT has the following steps:

1. Tentatively extend or modify a plan (PROPOSE-A-PLAN-EXTENSION) and identify constraints on the extension just formed (IDENTIFY-A-CONSTRAINT)
2. Identify constraint violations; if none, go to 1
3. Suggest potential remedies for a constraint violation (PROPOSE-A-FIX)
4. Select the least costly fix not yet attempted
5. Revise the plan
6. Identify constraint violations due to the revision; if any, go to 4
7. Remove relationships incompatible with the revision
8. Go to 1

To a first approximation, what a problem solving method provides is a partitioning of a knowledge base such that at any given time, any satisfied rule in the active partition can be appropriately applied. The problem solving method imposes a set of roles on its knowledge (thus creating the partitions) and defines the conditions under which each of those roles is relevant to the problem solving endeavor (thus defining the conditions under which a partition is active). SALT assumes that the knowledge bases of the systems it builds can be partitioned into two parts: (1) knowledge of how a plan might be extended or modified (PROPOSE-A-PLAN-EXTENSION) together with knowledge of the constraints that have to be satisfied for the change to be successful (IDENTIFY-A-CONSTRAINT), and (2) knowledge of how a plan might be modified to fix a constraint violation (PROPOSE-A-FIX). PROPOSE-A-PLAN-EXTENSION knowledge guides the processing done in step 1.

IDENTIFY-A-CONSTRAINT knowledge is also introduced in step 1 as the plan is extended and is then interpreted in step 2. PROPOSE-A-FIX knowledge is introduced in step 3 and guides the exploration for and selection of remedies in steps 4 through 7. Step 5 also draws on PROPOSE-A-PLAN-EXTENSION knowledge for revising the plan and IDENTIFY-A-CONSTRAINT knowledge is used in step 6 to test for constraint violations.

A SALT-built system's mistakes are all due to missing knowledge of one of these three types or to inadequacies in its method at the boundaries of its capabilities. Some missing knowledge can be identified statically (ie, just by examining the knowledge base).

- If two rules that extend or modify a plan in incompatible ways can be simultaneously satisfied, then the conditions under which one or both should apply are incompletely specified (PROPOSE-A-PLAN-EXTENSION).
- If the system knows of a constraint but has no knowledge of how to fix it if it is violated, then the system is missing a piece of fix knowledge (PROPOSE-A-FIX).
- If the system has knowledge of a series of inferences such that each uses data produced by the inference before it and the first uses data produced by the last, then one of its pieces of knowledge for extending a plan must be recast as a constraint (IDENTIFY-A-CONSTRAINT), a piece of knowledge must be added to estimate one piece of data in a way that breaks the loop (PROPOSE-A-PLAN-EXTENSION), and one or more pieces of fix knowledge are required (PROPOSE-A-FIX).

Missing knowledge can also be identified dynamically (ie, by analyzing the mistakes the system makes):

- If the plan that the system creates is incomplete, then the system is missing a piece of knowledge that allows an inference to be made (PROPOSE-A-PLAN-EXTENSION).
- If the plan is complete but incorrect, then the system is missing a piece of constraint knowledge (IDENTIFY-A-CONSTRAINT).
- If the plan is complete and correct but suboptimal, then the system is missing a piece of fix knowledge (PROPOSE-A-FIX).

SALT has been used to build VT [Marcus 86], a system that configures elevator systems. Given the functional requirements for the completed configuration, a description of the spatial structure within which the configured system must fit, and possibly preferences for specific components, VT indicates what components should be used to build the system and specifies the spatial relationships among parts and between parts and structural landmarks. VT's knowledge-base (as of March, 1986) consists of over 1400 rules. Of these, 1142 are PROPOSE-A-PLAN-EXTENSION rules, 192 are IDENTIFY-A-CONSTRAINT rules, and 98 are PROPOSE-A-FIX rules. SALT is currently being used to build a system that manages the scheduling of the elevator system building process. It is likely that we will try to use SALT to build a system that can perform part of the computer system configuration task performed by R1.

SALT makes several strong assumptions about the tasks SALT-built systems will perform. SALT assumes, for example, that for any task, a set of potential fixes can be pre-enumerated for each possible constraint violation. We suspect that a configuration task of the sort performed by R1 does not have this property (which is why R1 goes to great lengths to ever avoid violating a constraint) and thus that SALT is not an appropriate tool to use to build a system for that task. A SALT-built system

will, however, always know when it has tried all of the fixes it knows about and thus could pass the problem of constructing an appropriate fix on to some other system if it knew of such a system and if it knew what information that system needed.

2.2. MOLE

The problem solving method MOLE presupposes is a variant of heuristic classification [Clancey 84, Buchanan 84]. It assumes the task is to identify some object on the basis of a set of cues (eg, the cause of a fault on the basis of manifestations of errorful behavior, the component that best fits some need on the basis of characteristics that ordinarily satisfy that need); it further assumes that the set of candidate objects is relatively small (so that the candidates need not be constructed, but can simply be recalled). The method presupposed by MOLE has the following steps:

1. Ask what symptoms need to be explained
2. Determine what hypotheses are possibly relevant (ASSOCIATE-S-WITH-H)
3. Determine what information will differentiate among the hypotheses (ASSOCIATE-S-WITH-H, ASSOCIATE-H-WITH-PC, QUALIFY-AN-ASSOCIATION)
4. Ask for that information
5. Combine the support provided by the new information with the previous support
6. Reject unneeded hypotheses (ASSOCIATE-S-WITH-H)
7. Quit if the hypotheses have been sufficiently differentiated
8. Go to 3

MOLE assumes that the knowledge bases of the systems it builds are not partitioned and contain knowledge of symptom/hypothesis associations (ASSOCIATE-S-WITH-H), knowledge of hypothesis/prior-condition associations (ASSOCIATE-H-WITH-PC), and knowledge of qualifications on associations (QUALIFY-AN-ASSOCIATION). In step 2, ASSOCIATE-S-WITH-H knowledge is brought to bear so that the set of candidate hypotheses will include only those that are viable. In step 3, all domain knowledge that has any bearing on the set of candidate hypotheses is identified. In step 6, ASSOCIATE-S-WITH-H knowledge is again brought to bear to reject unneeded hypotheses.

A MOLE-built system's mistakes are all due to missing knowledge of one of these three types or to inadequacies in its method at the boundaries of its capabilities. Little can be done statically to identify missing knowledge. Missing knowledge can, however, be identified dynamically:

- If an accepted hypothesis should have been rejected and was accepted, in part, because a piece of evidence is associated only with that hypothesis, then the system is missing a piece of knowledge that associates that piece of evidence with another hypothesis (ASSOCIATE-S-WITH-H).
- If an accepted hypothesis should have been rejected and there is no strong evidence against it, then look for negative evidence against that hypothesis (ASSOCIATE-H-WITH-PC).
- If a rejected hypothesis should have been accepted and there is no strong evidence for it, then look for evidence supporting that hypothesis (ASSOCIATE-H-WITH-PC).
- If a rejected hypothesis should have been accepted and there is at least one piece of strongly negative evidence, then look for background conditions qualifying the negative evidence (QUALIFY-AN-ASSOCIATION).
- If two indeterminate hypotheses should have been differentiated and are supported by the same symptom, then look for background conditions that indicate which hypothesis is

the most likely to account for some piece of evidence under a particular set of circumstances (QUALIFY-AN-ASSOCIATION).

MOLE has been used to build MILL, a prototype that can diagnose rolling mill problems. MILL assumes that it can interact with a technician, asking for various pieces of information until it has identified the most likely causes of the faults. The knowledge-base of the MILL prototype consists of approximately 61 rules (as of March, 1986). Of these, 33 rules are ASSOCIATE-S-WITH-H knowledge, 16 are ASSOCIATE-H-WITH-PC knowledge, and 12 are QUALIFY-AN-ASSOCIATION knowledge. MOLE is currently being used to build a system that selects suitable computer system components given a generic description of the computing requirements.

MOLE makes several strong assumptions about the tasks MOLE-built systems will perform. MOLE assumes, for example, that for any task, a set of objects can be pre-enumerated, one or more of which will adequately account for whatever evidence the system collects. There are some examples of tasks for which the method MOLE presupposes is well-suited, but we suspect that a much wider variety of tasks contain one or more subtasks that a MOLE-built system could perform. Thus a MOLE-built system could serve as an identification subsystem for many different kinds of problem solvers, if it knew what information the problem solver could provide it with.

2.3. SEAR

The problem solving method SEAR presupposes, like the method SALT presupposes, assumes the task is to create a plan; but unlike SALT, it assumes that the adequacy of a plan step can almost always be determined at the time the plan step is conceived (ie, that backtracking will ordinarily not be necessary). The method presupposed by SEAR has the following steps:

1. Propose candidate operators (PROPOSE-OPERATOR and REJECT-OPERATOR)
2. Eliminate obviously inferior candidates
3. Vote on the candidates (EVALUATE-OPERATOR)
4. Select one operator
5. Perform the actions associated with that operator (APPLY-OPERATOR)
6. Quit if there is nothing more to do (RECOGNIZE-SUCCESS and RECOGNIZE-FAILURE)
7. Go to 1

SEAR assumes that the knowledge bases of the systems it builds can be partitioned into four parts: (1) knowledge of when a particular operator is possibly relevant (PROPOSE-OPERATOR) and of when a proposed operator is clearly not relevant (REJECT-OPERATOR), (2) knowledge that signals the likely relevance of a proposed operator (EVALUATE-OPERATOR), (3) knowledge of what changes occur when an operator is applied (APPLY-OPERATOR), and (4) knowledge of how to recognize when there's no more to do in the current subtask (RECOGNIZE-SUCCESS and RECOGNIZE-FAILURE). In step 1, PROPOSE-OPERATOR knowledge generates plausible candidate operators and REJECT-OPERATOR knowledge prunes those candidates that are inappropriate in the current circumstances. In step 3, EVALUATE-OPERATOR knowledge allows support for each of the candidates to be accumulated. In step 5, APPLY OPERATOR knowledge effects all of the conditional and unconditional actions implied by the selected operator. In step 6, RECOGNIZE-SUCCESS and RECOGNIZE-FAILURE knowledge enables the system to recognize when it has finished the current subtask.

the most likely to account for some piece of evidence under a particular set of circumstances (QUALIFY-AN-ASSOCIATION).

MOLE has been used to build MILL, a prototype that can diagnose rolling mill problems. MILL assumes that it can interact with a technician, asking for various pieces of information until it has identified the most likely causes of the faults. The knowledge-base of the MILL prototype consists of approximately 61 rules (as of March, 1986). Of these, 33 rules are ASSOCIATE-S-WITH-H knowledge, 16 are ASSOCIATE-H-WITH-PC knowledge, and 12 are QUALIFY-AN-ASSOCIATION knowledge. MOLE is currently being used to build a system that selects suitable computer system components given a generic description of the computing requirements.

MOLE makes several strong assumptions about the tasks MOLE-built systems will perform. MOLE assumes, for example, that for any task, a set of objects can be pre-enumerated, one or more of which will adequately account for whatever evidence the system collects. There are some examples of tasks for which the method MOLE presupposes is well-suited, but we suspect that a much wider variety of tasks contain one or more subtasks that a MOLE-built system could perform. Thus a MOLE-built system could serve as an identification subsystem for many different kinds of problem solvers, if it knew what information the problem solver could provide it with.

2.3. SEAR

The problem solving method SEAR presupposes, like the method SALT presupposes, assumes the task is to create a plan; but unlike SALT, it assumes that the adequacy of a plan step can almost always be determined at the time the plan step is conceived (ie, that backtracking will ordinarily not be necessary). The method presupposed by SEAR has the following steps:

1. Propose candidate operators (PROPOSE-OPERATOR and REJECT-OPERATOR)
2. Eliminate obviously inferior candidates
3. Vote on the candidates (EVALUATE-OPERATOR)
4. Select one operator
5. Perform the actions associated with that operator (APPLY-OPERATOR)
6. Quit if there is nothing more to do (RECOGNIZE-SUCCESS and RECOGNIZE-FAILURE)
7. Go to 1

SEAR assumes that the knowledge bases of the systems it builds can be partitioned into four parts: (1) knowledge of when a particular operator is possibly relevant (PROPOSE-OPERATOR) and of when a proposed operator is clearly not relevant (REJECT-OPERATOR), (2) knowledge that signals the likely relevance of a proposed operator (EVALUATE-OPERATOR), (3) knowledge of what changes occur when an operator is applied (APPLY-OPERATOR), and (4) knowledge of how to recognize when there's no more to do in the current subtask (RECOGNIZE-SUCCESS and RECOGNIZE-FAILURE). In step 1, PROPOSE-OPERATOR knowledge generates plausible candidate operators and REJECT-OPERATOR knowledge prunes those candidates that are inappropriate in the current circumstances. In step 3, EVALUATE-OPERATOR knowledge allows support for each of the candidates to be accumulated. In step 5, APPLY OPERATOR knowledge effects all of the conditional and unconditional actions implied by the selected operator. In step 6, RECOGNIZE-SUCCESS and RECOGNIZE-FAILURE knowledge enables the system to recognize when it has finished the current subtask.

A SEAR-built system's mistakes are all due to missing knowledge of one of these six types or to inadequacies in its method at the boundaries of its capabilities. Missing knowledge can be identified statically:

- If the system knows how to perform an operation, but the system has no knowledge of the circumstances under which that operation is appropriately performed, then it is likely the system is missing pieces of knowledge that identify such circumstances (PROPOSE-OPERATOR).
- If the system knows of a circumstance under which some operation should be performed, but has no knowledge of how to perform the operation, then the system is missing knowledge of how to apply an operator (APPLY-OPERATOR).

Missing knowledge can also be identified dynamically:

- If the system failed to apply some operator at a point in time when the operator should have been applied and if the system was not aware that the operator was possibly appropriate, then the operator needs to be proposed under those circumstances (PROPOSE-OPERATOR).
- If the system applied an inappropriate operator, then either the system is unaware of some of the circumstances that make the operator inappropriate or the votes against this operator and/or for the operator that should have been applied need to be adjusted (REJECT-OPERATOR or EVALUATE-OPERATOR).
- If the system failed to apply some operator at a point in time when the operator should have been applied and if the system was aware that the operator was possibly appropriate, then the votes for this operator and/or against the operator that was applied need to be adjusted (EVALUATE-OPERATOR).
- If the system applied some operator and the current state is not modified as expected, then it is likely that the system is unaware of one of the conditional changes that should result from applying the operator (APPLY-OPERATOR).
- If the system continued to work at a task after it had achieved its goal or after it should have been apparent that the goal was unachievable, then the conditions of success or failure in these circumstances need to be defined (RECOGNIZE-SUCCESS or RECOGNIZE-FAILURE).

SEAR has been used to build DRONE [vandeBrug 86], a system that performs part of the computer system configuration task performed by R1. Given a set of modules, backplanes, boxes, and cabinets, DRONE configures the modules, adding containers as required. DRONE's knowledge-base (as of March, 1986) consists of approximately 200 rules. Of these, 53 are propose-operator rules, 11 are reject-operator rules, 27 are evaluate-operator rules, 76 are apply-operator rules, 21 are recognize-success rules, and 4 are recognize-failure rules. It is likely that we will try to use SEAR to build a system that can perform VT's task.

SEAR makes several strong assumptions about the tasks SEAR-built systems will perform. SEAR assumes, for example, that for any task, it is possible to determine an appropriate next step on the basis of information that is currently available (or that can be collected locally). We suspect that a configuration task of the sort performed by VT does not have this property and thus that SEAR is not an appropriate tool to use to build a system that could perform that task. A SEAR-built system does, however, have ways of identifying when there is ambiguity about what next step is appropriate and thus could pass the problem on to a system that did know how to solve it, if it knew of such a system

and if it knew what information that system needed.

3. A plausible prediction

One can imagine a variety of research programs leading to the development of reasonably intelligent systems. Though the programs will vary to some extent in terms of the issues they address, a more significant difference is which problems are viewed as critical and what strategy is proposed for dealing with those critical problems. This paper identifies three problems as critical and suggests the following strategy for attacking them:

1. Develop a variety of knowledge acquisition tools in order to understand for each the interrelationships among the problem solving method, the knowledge the method requires, and a task.
2. Develop collections of loosely coupled, cooperating expert systems, each of which has only a single problem solving method, but also enough awareness of its own limitations and the strengths of its partners to know when and how to pass problems on; experience with such systems will show how sets of problem solving methods can interact, how much they have to know about one another, how they can share knowledge, how they can effectively communicate.
3. Develop a single system that has a number of problem solving methods at its disposal and that knows enough about the requirements of each method and the ways in which they can interact to determine what method is most appropriate to the problem at hand.

There are at least two other research programs with the same general aims that have been proposed recently which differ from the one described in this paper primarily, I think, by collapsing the three critical problems identified above into a single, less tractable problem. One of these proposals claims that the special methods used by current generation expert systems can be replaced by a single general method that has the power of the special methods without having their limited scope [Rosenbloom 85], implying that the intermediate steps of building special knowledge acquisition tools and groups of cooperating expert systems can be bypassed. The other proposal claims that the special methods that we currently apply only individually will all fall together (ie, it will be obvious when to apply which method) when a huge body of world knowledge, comprised of pieces of knowledge from the most general to the most specific, has been collected [Lenat 86], and thus again that the intermediate steps of building special knowledge acquisition tools and groups of cooperating expert systems can be bypassed.

That these other two research proposals underestimate the magnitude of the task is hinted at by the three data points presented in section 2. Although a significant amount of effort has gone into developing these three tools, we still know very little about their relative scopes, about the assumptions that each makes about tasks, and about the relative importance of the various knowledge requirements of each. By using these tools to build a variety of systems and, in particular, by using different tools to build systems for the same task, we will gradually collect the information required to face the second critical problem.

Acknowledgements

Judith Bachant, Larry Eshelman, and Sandra Marcus contributed much of the analysis presented in section 2 and along with Tom Mitchell and Allen Newell provided much helpful criticism of earlier drafts of this paper.

References

[Bachant 84] Bachant, J. and J. McDermott.
R1 revisted: four years in the trenches.
AI Magazine 5(3), 1984.

[Boose 84] Boose, J.
Personal construct theory and the transfer of human expertise.
In *Proceedings of the National Conference on Artificial Intelligence*. Austin, Texas, 1984.

[Buchanan 84] Buchanan, B. and E. Shortliffe.
Rule-based Systems: the Mycin experiments of the Stanford Heuristic Programming Project.
Addison-Wesley, 1984.

[Chandra 83] Chandrasekaren, B.
Towards a taxonomy of problem solving types.
AI Magazine 4(1), 1983.

[Clancey 83] Clancey, W.
The advantages of abstract control knowledge in expert system design.
In *Proceedings of the National Conference on Artificial Intelligence*. Washington, D.C., 1983.

[Clancey 84] Clancey, W.
Classification problem solving.
In *Proceedings of the National Conference on Artificial Intelligence*. Austin, Texas, 1984.

[Davis 82] Davis, R. and D. Lenat.
Knowledge-Based Systems in Artificial Intelligence.
McGraw-Hill, 1982.

[Eshelman 86] Eshelman, L. and J. McDermott.
MOLE: a knowledge acquisition tool that uses its head.
Technical Report, Carnegie-Mellon University, Department of Computer Science, 1986.

[Kahn 85] Kahn, G., S. Nowlan, and J. McDermott.
Strategies for knowledge acquisition.
IEEE transactions on Pattern Analysis and Machine Intelligence 7(5), 1985.

[Lenat 86] Lenat, D. B., M. Prakash, and M. Shepherd.
CYC: using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks.
AI Magazine 6(4), 1986.

[Marcus 85] Marcus, S., J. McDermott and T. Wang.
Knowledge acquisition for constructive systems.
In *Proceedings of Ninth International Conference on Artificial Intelligence*. Los Angelos, California, 1985.

[Marcus 86] Marcus, S., J. Stout and J. McDermott.
VT: an expert elevator configurer.
Technical Report, Carnegie-Mellon University, Department of Computer Science, 1986.

[McDermott 82] McDermott, J.
R1: a rule-based configurer of computer systems.
Artificial Intelligence 19(1), 1982.

[Neches 84] Neches, R., W. Swartout, and J. Moore.
Enhanced maintenance and explanation of expert systems through explicit models of their development.
In *Proceedings of IEEE Workshop on Principles of Knowledge-based Systems.* Denver, Colorado, 1984.

[Rosenbloom 85] Rosenbloom, P., J. Laird, J. McDermott, A. Newell, and E. Orciuch.
R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture.
IEEE transactions on Pattern Analysis and Machine Intelligence 7(5), 1985.

[vandeBrug 86] van de Brug, A., J. Bachant, J. McDermott.
The Taming of R1.
IEEE Expert 1(3), 1986.