Knowledge Acquisition for Constructive Systems

Sandra Marcus, John McDermott, and Tianran Wang

Department of Computer Science

Carnegie-Mellon University

Pittsburgh Pennsylvania 15213

Abstract

Over the past ten years, significant progress has been made in understanding how the knowledge acquisition process for classification systems can be automated. But during this period little attention has been paid to the problem of how to automate the knowledge acquisition process for systems that solve problems by constructing solutions. This paper describes SALT, a tool designed to assist with knowledge acquisition for configuration tasks. SALT[1] assumes a problem-solving strategy involving stages of generate, test, backup, modify, and re-generate. It exploits this problem-solving strategy to guide its interrogation of domain experts and to represent the knowledge they provide in a way that insures it will be brought to bear whenever relevant.[2]

1. Introduction

One of the great things about MYCIN [Shortliffe 76] and other such systems is that their domain knowledge is kept distinct from the knowledge of how and when to apply that knowledge. The separation is achieved by defining a problem-solving strategy for classification [Clancey 84] which completely determines the use to which the domain knowledge will be put. Although more recent work (eg, [Clancey 83], [Neches 84]) has shown that in the early systems the separation left something to be desired, even in its early form, the particular way in which it was achieved had important implications for knowledge acquisition.

In general, a knowledge acquisition tool can provide two kinds of assistance: (1) it can make it easy for someone with a particular expertise to communicate that expertise, and (2) it can somehow organize (or proceduralize) the knowledge that is communicated so that ail of the knowledge that is relevant in a particular situation gets brought to bear. The MYCIN work suggested an approach to knowledge organization that has turned out to be quite powerful. The idea is to use a problem-solving strategy appropriate to some possibly quite narrow problem type to define the roles that the problem-solver's knowledge can play. A number of knowledge acquisition tools that exploit this approach to knowledge organization in classification problem-solvers have been developed (eg, [Davis 82], [Boose84],[Kahn84]).

[1]kNowtedge Acquisition Language

[2]We are grateful to Bob Roche, first domain expert for VT, who provided valuable assessments of feasibility and utility during the development of SALT and to Jeff Stout who consulted on explanation capabilities for SALT-generated expert systems We would also like to thank Allen Newell and Tom Mitchell for comments on an earlier draft of the paper.

The work described in this paper extends this idea to another type of problem-solver; the work is a first attempt to define the roles knowledge can play in configuring electro-mechanical systems. Section 2 identifies the task demands and one strategy for solving the problem. Section 3 describes SALT, a tool for developing and maintaining knowledge-based configurers. Section 4 discusses SALT'S current usefulness and plans for its extension.

2. A problem-solving strategy for configuration tasks

SALT was developed as a knowledge acquisition tool for VT, an elevator system configurer. The input to the configurer was to include functional requirements for the completed configuration, preferences for specific parts and a description of the spatial structure within which the configured system must fit. The system's output was to consist of quantities, descriptions and model numbers of parts selected and a specification of spatial relationships among parts and between parts and structural landmarks.

An isolated selection step might entail consulting a database of part specifications and selecting the least costly part whose specifications match the demands of the current partial configuration. An isolated layout step might consist of defining a distance between two parts as some algebraic function of the space available. What these steps have in common is that they are procedures for arriving at a piece of data needed to describe a configuration (a part type, a distance between two parts) by applying a relatively small class of methods (database lookups, calculations, assignment of constants) to other pieces of data (demands of the current configuration, available space).

What makes this type of configuration task interesting is that steps which appear acceptable in isolation may result in unacceptable configurations when combined. The problem-solving strategy we will focus on allows the configuring to proceed using the step sizes that are produced most naturally by configuration experts. It then employs other domain knowledge for spotting unacceptable configurations and deciding how to backtrack and effect a change that will avoid the original problem. This strategy has the following stages:

1. Generate each piece of the configuration (parts and relationships) using an appropriate method.
2. Identify constraints (ie, limits on configuration values).
3. Compare each constrained value with its constraints.
4. If a constraint is violated, determine what values could be changed to remedy the violation.

5. Choose the least damaging change or change combination that remedies the violation.
6. Make the change and remove any values that depended on the old value.
7. Return to the generate phase.

This strategy defines a problem-solving shell to which more specific domain knowledge can be added. Three roles are specified: (1) a piece of knowledge can indicate how to determine the value of some piece of a configured system, (2) a piece of knowledge can indicate how to spot a constraint violation, and (3) a piece of knowledge can indicate how to remedy a constraint violation.

Use of data-driven procedures in the first stage of the problem-solving shell require knowledge that indicates how to determine the value of some piece of a configured system. Needed are the details of the appropriate method (eg, an algebraic formula for a calculation or parameters for a database lookup) along with any preconditions on the applicability of the method. Other configuration values used in specifying preconditions or methods define what values must already be available during configuration in order to apply this method.

Steps 2 and 3 require knowledge that indicates how to spot a constraint violation. Wherever a criterion exists for deciding whether a configuration value is acceptable, the test must be specified. This requires a comparison of the configuration value to some reference value or set of values using tests such as less than, equal to, member of set.

Steps 4, 5 and 6 define the need for knowledge that indicates how to remedy each potential constraint violation. Specific domain knowledge is needed to identify alternative changes to particular configuration values that alone or in combination might solve the constraint violation. Domain knowledge is also required to weight these potential "fixes" according to their negative effect on the configuration. Once changes are implemented, these define the points to backtrack to.

3. SALT

SALT consists of two subsystems that share a knowledge base. One subsystem interviews the domain expert to elicit the three kinds of knowledge the problem-solver requires and builds up a representation of that knowledge which can be accessed both by itself and by the second SALT subsystem, the rule generator. The rule generator translates this representation into OPS5 rules that are then combined with the shell (the OPS5 interpreter plus some OPS5 rules) to form a configurer. SALT can help with knowledge acquisition because it knows the roles knowledge can play. The roles provide SALT with a way of focussing the experts' attention on the knowledge required to perform a task. Moreover, since a role defines the way in which a piece of knowledge is to be used, SALT can represent the knowledge in a way that insures that whenever it is relevant it will be brought to bear.

3.1. The interview

SALT was designed to be used by a domain expert with no background in AI. In order to avoid natural language issues, the expert is required to use a somewhat structured language. A three page document familiarizes the user with some of the terms and questions that will be used in the interview. Most questions are answered with yes or no, a name, or a selection from a menu. SALT makes it easy to edit the knowledge base as well as enter new knowledge. The user is asked to indicate the kind of knowledge to be entered or modified (method, constraint, or fix).

Describing a new method involves filling in slots in a schema. If two methods presuppose the results of each other, SALT will ask the user to supply an "optimal" estimate for one. SALT will then use the original procedure as a constraint on the estimate and will ask the user how it limits the estimate. For any new method, the user will be asked if there are any constraints on its value. For any new constraint the user will be asked for potential fixes. When the user asks to work on a piece of knowledge previously entered, that knowledge is displayed. In cases where the request is ambiguous, SALT will supply additional information that distinguishes among the possible targets. Examples below are taken from a knowledge base for configuring elevators:

```
 1 METHOD        ENTER A METHOD FOR DETERMINING A VALUE
 2 CONSTRAINT    ENTER A CONSTRAINT ON A VALUE
 3 FIX           ENTER A REMEDY FOR A CONSTRAINT VIOLATION
 4 SYNONYM       ENTER A SYNONYM

  ENTER YOUR COMMAND [EXIT]: 1

  THE VALUE REQUIRING A METHOD? car-buffer-model

THERE ARE MULTIPLE METHODS TO DETERMINE CAR-BUFFER-MODEL

 1  PRECONDITION     SPEED < 200
 2  PRECONDITION     SPEED>= 200

  SPECIFY THE ONE YOU WANT TO WORK ON (0 FOR NEW) [0]: 2
```

The information for the candidate requested is then displayed:

```
METHOD
 1  NAME:                      CAR-BUFFER-MODEL
 2  ENTITY TYPE:               OBJECT
 3  OBJECT NAME:               CAR-BUFFER
 4  OBJECT PROPERTY:           MODEL
 5  CONSTRAINT TYPE:           ACTUAL
 6  PRECONDITION:              SPEED >= 200
 7  METHOD:                    DATABASE LOOKUP
 8  TABLE NAME:                BUFFER
 9  COLUMN OF NEEDED VALUE:    MODEL
10  PARAMETER TEST:            STROKE>=  MINIMUM-STROKE
11  ORDERING COLUMN:           HEIGHT
12  OPTIMAL:                   SMALLEST

  ENTER YOUR COMMAND [EXIT]: 8 Oil-buffer
```

Specifications are entered by typing the number of the line and the new value desired. The display is then refreshed with the change. The parameter test in line 10 states that for the value retrieved the entry in the table column "stroke" must be greater than or equal to the value of "minimum-stroke" which must be supplied by another method. Some lines are dependent on others; if the user had specified calculation for the method, lines 8 through 12 would be replaced by a line asking for the formula. The schema for specifying constraints is very similar to the method schema. An example of a fix schema is shown below:

```
FIX
 1  VIOLATED CONSTRAINT:          MAXIMUM-BUFFER-LOAD
 2  CONSTRAINED VALUE:            BUFFER-LOAD
 3  VALUE TO CHANGE:              BUFFERQUANTITY
 4  CHANGE TYPE:                  INCREASE
 5  STEP TYPE:                    BYSTEP
 6  STEP SIZE:                    1
 7  RATING OF UNDESIRABILITY:     4
 8  REASON FOR UNDESIRABILITY:    CHANGES MINOR EQUIPMENT
                                  SELECTION/SIZING
```

```
ENTER YOUR COMMAND [EXIT]: h 7
```

Requesting help for the rating of undesirability as shown above will display the menu below:

```
 1 NO PROBLEM
 2 INCREASES MAINTENANCE REQUIREMENTS
 3 MAKES INSTALLATION INCONVENIENT
 4 CHANGES MINOR EQUIPMENT SELECTION/SIZING
 5 VIOLATES MINOR EQUIPMENT CONSTRAINT
 6 CHANGES MINOR CONTRACT SPECIFICATIONS
 7 REQUIRES SPECIAL PART DESIGN
 8 CHANGES MAJOR EQUIPMENT SELECTION
 9 CHANGES THE BUILDING DIMENSIONS
10 CHANGES MAJOR CONTRACT SPECIFICATIONS
11 COMPROMISES SYSTEM PERFORMANCE
12 VIOLATES SAFETY CODE

ENTER YOUR COMMAND [EXIT]: Z
```

Typing "7" will substitue that value for rating of undesirability and will fill in the appropriate reason in the next line.

3.2. Rule generation

The rule generator can be revised as the shell is changed to make the problem solving strategy more robust and efficient. Because of this the exact form and numbers of rules produced is not stable. Currently, for every method or constraint schema, SALT writes at least one rule. For every constraint that has at least one potential fix, there is one rule written to make available at runtime all of the information in each fix schema. There are also one or more rules used to assess the ability of fix alternatives to produce a configuration acceptable up to the point of the original violation; these are generated using fix knowledge as well as that from the method and constraint schemas for contributors to the violation. In addition, a single rule is generated that matches synonyms to the schema name. All of these rules are written so that when they fire, they leave a trace describing how it was that they came to be applied and what they did; this information is used for explanation.

4. SALT'S potential

The first task SALT was given was to re-generate VT. The version of VT built by hand took about 3 worker-years to develop and consisted of about 1330 rules. About half of the effort and half of the rules belong to VT's I/O package and the shell that can use both the hand-coded and SALT generated rules. SALT was used to regenerate the remaining half of the rules (the knowledge base). Using the hand-coded VT as the knowledge source, it took about 11 hours to enter half of the knowledge that version had. Our domain expert then spent another 35 hours refining this knowledge and entering the remaining knowledge. The SALT rule generator took 20 VAX-11/780 cpu minutes to produce 636 rules.

SALT was also used successfully to develop a system that could select elevator control cables. This task is done after an elevator system has been configured; the task requires using the layout and functional requirements for the wires to order cable(s) that meet local safety codes and are available in stock. The only change required to SALT was to provide it with a new piece of fix knowledge: the additional cost of repeating a fix.

SALT makes the strong assumption that a task can be structured so that the domain expert can provide a method to establish initial values, a method to define the boundary between acceptable and unacceptable solutions, and a way of changing values, depending on the boundary that was crossed, that will lead to an acceptable solution. Our plan is to test the applicability of SALT on a range of constructive tasks including scheduling tasks. Some scheduling tasks require coming up with initial plans, recognizing unexpected situations that make the plans infeasible and making the minimally disruptive changes that make the plans acceptable once more. SALT appears to be well-suited to building schedulers of this sort.

5. Conclusion

SALT is interesting if its problem-solving strategy is sufficiently specific to strongly constrain the roles that the knowledge it needs can play, while at the same time being sufficiently general to apply to a variety of problems. The fact that SALT'S strategy imposes three roles on the knowledge it uses does, in fact, provide substantial help both in focussing attention on what knowledge needs to be collected and in defining the conditions under which pieces of knowledge are relevant. There is some reason to believe that SALT'S strategy will apply to tasks that are apparently quite diverse, but this has yet to be demonstrated.

References

[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.

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

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

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

[Kahn 84] Kahn, G., S. Nowlan, and J. McDermott.
A foundation for knowledge acquisition.
In *Proceedings of IEEE Workshop on Principles of Knowledge-based Systems.* Denver, Colorado, 1984.

[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.

[Shortfiffe76] Shortliffe, E.
Computer-Based Medical Consultation: Mycin.
Elsevier, 1976.