ISSUES IN AI SYSTEMS

N90-29904

Generic Task Problem Solvers in Soar

Todd R. Johnson, Jack W. Smith, Jr., B. Chandrasekaran

Laboratory for AI Research (LAIR)
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

1 Introduction

Two trends can be discerned in research in problem solving architectures in the last few years: On one hand, interest in *task-specific architectures*[3, 8, 2] has grown, wherein types of problems of general utility are identified, and special architectures that support the development of problem solving systems for those types of problems are proposed. These architectures help in the acquisition and specification of knowledge by providing inference methods that are appropriate for the type of problem. However, knowledge-based systems which use only one type of problem solving method are very brittle, and adding more types of methods requires a principled approach to integrating them in a flexible way.

Contrasting with this trend is the proposal for a flexible, general architecture contained in the work on Soar[4]. Soar has features which make it attractive for flexible use of all potentially relevant knowledge or methods. But as a theory Soar does not make commitments to specific types of problem solvers or provide guidance for their construction.

In this paper we investigate how task-specific architectures can be constructed in Soar to retain as many of the advantages as possible of both approaches. We will be using examples from the Generic Task (GT) approach for building knowledge-based systems in our discussion since this approach had its genesis at our Laboratory where it has further been developed and applied for a number of problems; however the ideas are applicable to other task-specific approaches as well.

2 The GT Paradigm

The GT paradigm is a theory of types of goals and the problem solving methods needed to achieve each type. By problem solving method we mean the specification of behavior to achieve a goal. The paradigm has three main parts:

1. The problem solving of an intelligent agent can be characterized by generic types of goals. Many problems can be solved using some combination of these types.

2. For each type of goal there are one or more problem solving methods, any one of which can potentially be used to achieve the goal.

3. Each problem solving method requires certain kinds of knowledge of the task in order to execute. These are called the *operational demands* of the method[7].

The term *generic task* refers to the combination of a type of goal with a problem solving method and the kinds of knowledge needed to use the method. The GT for classification by establish-refine (called the E-R GT) is given as:

Type of Goal Classify a (possibly complex) description of a situation as a class in a set of categories. An instance of this goal is the classification of a medical case description as one of a set of diseases.

Problem Solving Method This is a hierarchical classification method that works by creating and testing hypotheses about the plausibility that the description of the situation represents an instance of one or more of the classes.

1. If there are no initial hypotheses about what class the description is an instance of, then try to suggest at least one.
2. Try to confirm or reject any hypothesis that is suggested.
3. If a hypothesis is confirmed, determine the possible refinements of the hypothesis and suggest them.
4. If the goal is not met, go to step 2.

Kinds of Knowledge These consist of a refinement hierarchy, hypotheses about the presence of classes, confirmation/rule-out knowledge for these hypotheses, and knowledge to determine when the goal of classification has been achieved.

In addition to classification by establish-refine, GT's have been created for pattern directed hypothesis matching[5], assembly of explanatory hypotheses[6], and object synthesis by plan selection and refinement[1].

3 GT Systems

A specialized architecture or shell has been constructed for each GT. Each architecture is a combination of an inference engine with a knowledge base. The inference engine is a procedural representation of a GT's problem solving method. The knowledge base provides primitives for encoding the domain specific knowledge needed to instantiate the procedure. We refer to the combination of the encoding of the domain knowledge in the knowledge base and the method that can use it as a problem-solver.

This system building approach offers a number of advantages: First, it is easy to decide when a GT architecture can be used because the knowledge operationally demanded by the method is explicit in the definition of the GT. Second, knowledge acquisition is facilitated because the representational primitives of the knowledge base directly correspond to the kinds of domain knowledge that must be gathered. Third, explanation based on a run-time trace can be couched in terms of the method and knowledge being used to apply it.

4 Problems with GT Systems

Many flexibility problems arise because a GT architecture contains assumptions not present in the original GT problem solving method. For example, our architecture for hierarchical classification assumes that hypotheses are generated from a pre-defined hierarchy. While this is a common way to generate refinements, other ways exist and might be useful in certain domains. Second, the architecture immediately generates refinements for a confirmed hypothesis. An alternative is to test all the hypotheses in the current state before refining any that were confirmed. Third, the architecture assumes that any problem solver it calls will correctly function. We cannot easily modify the architecture to gracefully handle these situations.

Another set of problems involves the integration of multiple GT problem solvers. The simplest kind of integration is when one problem solver calls on another as a direct means to achieve a subgoal. This is easily done using our current architectures by directly invoking the method and domain knowledge needed to achieve a subgoal. However, sometimes we require more interaction between the problem solvers. For example, in our medical diagnosis systems the hypothesis assembly problem solver has knowledge about those diseases that can occur together and those that are mutually exclusive. This knowledge can be used help guide the classification of diseases; however, it is difficult to implement because the classification architecture has no place for representing or using this knowledge. Our only solution was to specially modify both architectures so that they could interact in the desired way.

Finally, new methods are difficult to add to existing problem solvers; each problem solver must be modified to recognize and use a new method. We would like to have the system automatically consider methods based on the type of goal a method is designed to achieve.

5 How can Soar Help?

In Soar, all problem solving is viewed as search for a goal state in a problem space. Operators are used to move from state to state. Knowledge in the form of productions is used to select problem spaces, states, and operators. Productions generate *preferences* for an object (ie. a problem space, state, or operator) that indicate how the object relates to the current situation or other objects. Whenever the directly available knowledge is insufficient to make progress Soar automatically generates a subgoal. Therefore, every decision that needs to be made can become a goal to be achieved by searching a problem space. This is called *universal subgoaling*. In knowledge lean situations Soar can make progress by using an appropriate weak method. The weak methods are not explicitly programmed in Soar, but arise from the knowledge available to solve a problem. If the processing in a subgoal is no longer needed, Soar will automatically terminate the subgoal and resume problem solving in a higher level goal. This is called *automatic goal termination*.

Each of these features directly relate to one or more of the limitations with GT systems. The selection of alternatives via preferences allows new options and knowledge to be easily added to existing systems. Brittleness is decreased because of Soar's ability to automatically create subgoals to overcome failures and its ability to fall back on weak methods. Finally, automatic goal termination eliminates unnecessary computation and provides a more natural flow of control.

6 Mapping GT's to Soar

We have begun to map GT's to the Soar architecture in a straight-forward manner. Each GT is implemented as a problem space; the states represent the developing solution and the operators and operator suggestion rules implement the problem solving method. The required kinds of knowledge can either be represented directly by productions or generated at run-time using additional problem spaces.

To illustrate, we present ER-Soar, an implementation of the E-R GT in Soar. We use a single problem space with three operators: suggest-initial-hypotheses, establish, and generate-refinements.

State Representation The state contains those hypotheses that have been considered and those that are worth immediately considering. Any hypotheses in the state that are refinements of other hypotheses (also in the state) are linked together to form a refinement hierarchy. Each hypothesis also has an indication of whether it has been confirmed, rejected, or not yet judged, and whether it has been refined or not.

Operators The classify problem-space uses 3 operators:

suggest-initial-hypotheses Determine one or more initial hypotheses.

establish <hyp> Determine whether the hypothesis, <hyp>, should be confirmed or rejected.

generate-refinements <hyp> Generate (add to the state) those hypotheses that should be considered as a refinement of <hyp>.

Operator Instantiation A suggest-initial-hypotheses operator is created if there are no hypotheses in the current state. An establish operator is created for any hypothesis in the current state that has not yet been judged. A generate-refinements operator is created for any hypothesis that is confirmed but not refined.

Domain knowledge To use the E-R strategy in a particular domain, knowledge to perform the following functions must be added to the Soar implementation.

- Create the initial state containing one or more initial hypotheses.
- Detect when classification is complete.
- Implement the three operators.

Operator Implementation There are many ways to implement the operators used in the classify space. To make ER-Soar easy to use we have implemented a method for generating refinements from a pre-defined hierarchy and a method for establishing hypotheses based on a confidence value.

6.1 Discussion

ER-Soar combines the advantages of the GT approach with the advantages of the Soar architecture. Knowledge acquisition, ease of use, and explanation are all facilitated in ER-Soar because subgoals of the problem solving method and the kinds of knowledge needed to use the method are explicitly represented in the implementation. The subgoals of the method are directly represented as problem space operators. The kinds of knowledge needed to use the method are either encoded in productions or computed in a subgoal. The same advantages apply to the supplied methods for achieving subgoals. Finally, the implementation mirrors the GT specification quite closely making ER-Soar easy to understand and use.

ER-Soar overcomes many of the problems suffered by previous GT systems. Automatic subgoaling allows unanticipated situations to be detected and handled. If no specific method for handling the situation is available, an appropriate weak method can be used. Whenever a goal needs to be achieved it is done by first suggesting problem-spaces and then selecting one to use. This allows new methods in the form of problem-spaces to be easily added to existing problem solvers. If no specific technique exists to determine which method to use, Soar will try to pick one using a weak method. Automatic goal termination provides an integration functionality not available in previous GT architectures. In general, the integration capabilities of ER-Soar are greatly enhanced. Because of preferences and the additive nature of productions, new knowledge can be added to integrate ER-Soar with other methods without modifying existing control knowledge.

7 Conclusion

ER-Soar illustrates how the advantages of task-specific architectures can be combined with the advantages of a general architecture. The approach used to create ER-Soar is simple and can easily be applied to other task-specific architectures. We are currently using this approach to create Soar versions of the GT's for hypothesis matching and hypothesis assembly. Following this, we will investigate various ways of integrating the three methods.

8 Acknowledgements

This research is supported by National Heart Lung and Blood Institute grant HL-38776, Air Force Office of Scientific Research grant 82-0255, and Defense Advanced Research Projects Agency, RADC Contract F30602-85-C-0010. Dr. Jack W. Smith, Jr. is supported by National Library of Medicine Career Development Award LM-00083. Computer facilities were enhanced by gifts from Xerox Corporation.

References

[1] D. C. Brown and B. Chandrasekaran. Plan selection in design problem-solving. In *Proc. of AISB85 Conference*, Warwick, England, 1985. The Society for AI and the Simulation of Behavior (AISB).

[2] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, 1(3):23–30, 1986.

[3] W. J. Clancey. Heuristic classification. *Artificial Intelligence*, 27(3):289–350, 1985.

[4] Paul S. Rosenbloom John E. Laird, Allen Newell. SOAR: An architecture for general intelligence. *Aritificial Intelligence*, 33:1–64, 1987.

[5] Todd R. Johnson, Jack W. Smith Jr., and Tom Bylander. HYPER: Hypothesis matching using compiled knowledge. Technical report, Lab. for AI Research, CIS Dept., The Ohio State University, Columbus, Ohio, 1988.

[6] J. R. Josephson, B. Chandrasekaran, J. W. Smith, and M. C. Tanner. A mechanism for forming composite explanatory hypotheses. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17(3):445–454, 1987.

[7] John Laird, Paul Rosenbloom, and Allen Newell. *Universal Subgoaling and Chunking.* Kluwer Academic Publishers, Massachusets, 1986.

[8] Marcus, Sandra McDermott, and John McDermott. Salt: A knowledge acquisition tool for propose and revise systems. Technical report, Carnegie-Mellon University, 1987.