CHAPTER 48

Task-Specific Architectures for Flexible Systems

T. R. Johnson, J. W. Smith, and B. Chandrasekaran, The Ohio State University

Abstract

This paper describes our research which is an attempt to retain as many of the advantages as possible of both task-specific architectures and more general problem-solving architectures like Soar. It investigates how task-specific architectures can be constructed in the Soar framework and integrated and used in a flexible manner. The results of our investigation are a preliminary step towards unification of general and task-specific problem solving theories and architectures.

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* (Chandrasekaran, 1986; Clancey, 1985; Marcus, 1988) 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 that is best exemplified by the work on Soar (Laird, et al., 1987). Soar has features which make it

In P. S. Rosenbloom, J. E. Laird, & A. Newell (Eds.), The Soar Papers: Research on Integrated Intelligence (pp. 1004-1026) Cambridge, Mass: MIT Press.

attractive because it allows 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 (TSA's) 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 since this approach had its genesis at our Laboratory where it has been further developed and applied for a number of problems; however the ideas are applicable to other task-specific approaches as well.

2 Flexible Systems: A Definition

At a high level of abstraction, a flexible system is a system that can make reasonable progress on any problem instance it is given. If additional knowledge is required to make progress, then the system should work to acquire that knowledge. To be more specific we consider a flexible system to be a problem-solving system that:

1. Can engage in complex problem solving about the potential actions that can be used to solve a problem.

2. Can engage in complex problem solving about what action to take to solve a problem (given a set of potential actions).

3. Allows easy modification of its knowledge through incorporation of new knowledge or changes to existing knowledge.

Note that a flexible system, by this definition, is not necessarily robust. Without appropriate knowledge a flexible system can still be brittle; however, a flexible system has the potential to be robust and adaptive.

3 Limitations of Inflexible Systems

An inflexible system is forced to apply the same reasoning method, or sequence of operations, for every problem instance—the system cannot adapt its reasoning method to the problem being solved. This has several consequences. First, inflexibility can lead to a brittle system. A system is said to be brittle if it cannot behave appropriately when there are small variations in the task or task environment. There are several kinds of inappropriate behavior. First, the system might break, i.e., abnormally halt. For example, suppose that a diagnostic system has a subgoal to determine the confidence of hypotheses by returning *-1*, *0*, or *1*. The subgoal could produce a value outside the prescribed range, such as *high*. Unless the system has a way of handling the unexpected value, the system could break when it attempts to use the value. In this case, *high* might cause an arithmetic function to fail.

Second, the system might get an incorrect answer. For instance, if the diagnostic system described above only considered hypotheses with a confidence of *1*, it would ignore any hypotheses that had nonstandard confidence values, like *high*. This could result in selecting an incorrect hypothesis for the final result.

The third consequence of an inflexible system is that the system might behave in a nonoptimal or unusual way. For example, the diagnostic system might be forced to pursue one hypothesis while holding off the exploration of a more likely hypothesis. Another example is when a system continues to pursue subgoals even though the topmost goal has already been met.

4 Why are Flexible Systems Needed?

Clearly, it is undesirable for a system to behave in the ways described above. However, you might still wonder why we should expect or want systems to work outside the narrow domain in which they were originally designed. After all, we use programs every day

that are designed to work only in narrow domains or solve specific types of problems (e.g., word processors, spreadsheets, and databases). Why should we expect AI systems to be any different?

There are two reasons. First, general intelligence is not defined as the ability to solve a single, bounded problem. An intelligent agent must be ready to solve any problem thrown at it over the course of its existence. An important defining feature of an intelligent system is its ability to dynamically adapt to solve the problem at hand. If we want to build intelligent systems then we should expect no less of them.

Second, most problems being tackled by AI systems are so ill-defined that the input and output for a system cannot be precisely specified at design time. Even if it could be precisely specified, the specification might change because: (1) knowledge about the task or domain might grow or change; (2) the task that the system is designed to do might change (thus requiring a different specification); and (3) the knowledge available to solve the task may be different than what was assumed when the system was first built.

Thus, the ill-defined nature of most AI problems combined with the dynamic nature of task environments demand systems that can adapt to new situations. Whether the adaptation needs to be dynamic or through human intervention depends on the system. Dynamic or automatic adaptation is required when a system must be autonomous or if the task is always varying. Human intervention (i.e., modification of the system by reprogramming) is sufficient whenever little autonomy is required or the task is fairly well-defined. However, even if intervention is acceptable, the system must be capable of supporting change. Unfortunately, many AI systems cannot accommodate these changes. The redesigned tools described in this paper allow both kinds of adaptation

5 The Generic Task Paradigm

Generic tasks (GT's) (Chandrasekaran, 1986) provide an implementation-independent vocabulary for describing problem-solving systems. They allow systems to be described in terms of goals and knowledge that directly relate to the task, instead of implementational formalisms such as rules, frames, or LISP. 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 after (Laird, et al., 1987).

A 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. For example, the GT for hierarchical classification used in CSRL (Bylander and Mittal, 1986) is specified as:

Type of Goal: Classify a (possibly complex) description of a situation as a member of one or more categories. An instance of this goal is the classification of a medical case description as one or more diseases.

Problem-solving Method: This is a hierarchical classification method that works by evaluating and refining hypotheses about the situation. If a hypothesis can be established (i.e., it has a high likelihood of being true), then it is refined into hypotheses of greater detail. CSRL stops when it has run out of refinements to explore, or it has ruled-out every branch of the hierarchy. Here the method is described using a main procedure (*Classify*) which calls on a recursive sub-procedure (*E-R*).

```
Classify(root-hypothesis)
    E-R(root-hypothesis)
end Classify

E-R(hypothesis)
    Determine-certainty(hypothesis)
    If High-certainty(hypothesis) then
        H ← Refinements(hypothesis)
        for each hypothesis, h, in H do
            E-R(h)
        end for
    end if
end E-R
```

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.

Note that a GT is a theory about how to do a general task. It specifies the task, a method for doing the task, and the knowledge required by the method, but it does not specify any details of implementation. The GT specification mentions what knowledge is required, but doesn't say how the knowledge should be represented or computed. Likewise, the problem-solving method specifies subgoals and how the subgoals work together, but it does not specify how the subgoals should be achieved. Hence, a GT analysis of a task can be used to describe systems that are implemented in many different architectures or languages.

To build a system, we must specify how all the knowledge required by a GT's method is to be computed. Instead of moving directly to an implementation formalism, the generic task paradigm can be applied to the subgoals of a GT problem-solving method. For example, we can specify a GT for the task of testing a hypothesis. This recursive application of the GT paradigm leads to a hierarchical goal/subgoal decomposition of a system called the *task structure* (Chandrasekaran, 1989). The task structure shows how a task is solved in terms of other tasks. One possible task structure for a diagnostic system is shown in Figure 1. The task of diagnosis has been decomposed into two subtasks: *classify findings*

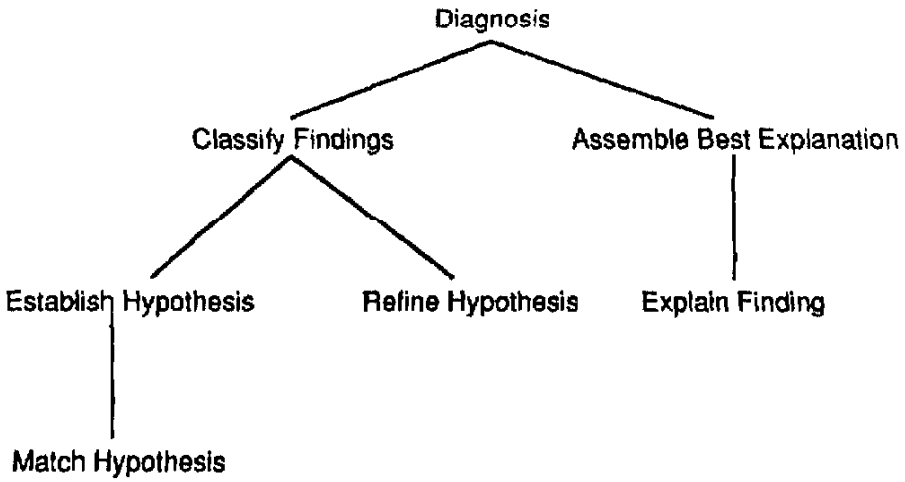

Figure 1: Task structure for a diagnostic system.

and *assemble best explanation. Classify findings*, in turn, has been further decomposed into two additional tasks: *establish hypothesis* and *refine hypothesis*.

6 Task-Specific Architectures

A TSA implements a GT by providing a task-specific inference engine and knowledge-base representation language. The inference engine implements a GT's problem-solving method. The knowledge base provides primitives for encoding the domain specific knowledge needed to instantiate the method. The combination of the encoding of the domain knowledge in the knowledge base and the method that can use it is called a *problem solver*. Thus, a system built using CSRL is called a *hierarchical classification problem solver* or simply a *classifier*.

In addition to CSRL, the GT paradigm has been used to design TSA's for object synthesis by plan selection and refinement (Brown and Chandrasekaran, 1989), assembly of explanatory hypotheses (abduction) (Josephson, et al., 1987), and pattern-directed hypoth-

esis matching (Johnson, et al., 1989). Other researchers have used related approaches to design tools for these and other tasks. Some examples are Clancey's Heracles (Clancey, 1985), McDermott's role-limiting methods (McDermott, 1988), and Kingsland's criteria tables (Kingsland III and Lindberg, 1986).

A problem-solving system often consists of multiple problem solvers that work together, such as a diagnostic system with a classifier, several hypothesis matchers (one for each hypothesis in the classification hierarchy) and an abducer (a problem solver that finds a best explanation). Problem solvers are often related as goal/subgoal pairs. For example, *determine-certainty* is a subgoal of the *E-R* procedure, that can be achieved by using a hypothesis matcher.

Although the GT paradigm naturally leads to TSA's, it is important to realize that a GT analysis does not require the construction of a TSA. Once a GT specification is done for a system, the system can be implemented in whatever language or tools the system builder has available.

6.1 Advantages

TSA's offer a number of advantages over general architectures:

- They provide implementation-independent abstractions for describing and building systems. Thus, a system can be said to be doing classification regardless of whether it is written in a production system or in lisp.
- They provide a theory of how to solve a wide range of problems.
- Deciding when to use a tool is facilitated because the knowledge operationally demanded by the method is explicit in the definition of the tool.

- Knowledge-acquisition is facilitated because the representational primitives of the knowledge base directly correspond to the kinds of domain knowledge that must be gathered.

- Explanations based on a run-time trace can be couched in terms of the method and knowledge being used to apply it.

6.2 Inflexibility in TSA systems

Standard implementations of TSA systems tend to be inflexible for three reasons. First, each tool is usually based on a relatively fixed method. This leads to the following limitations:

- The sequence of operations (i.e., method subgoals) is largely predefined. (The systems do not engage in complex problem solving about the operations or about which to select.)

- Very few modifications can be made to change the operations or their sequence.

Second, a fixed knowledge-representation language makes it difficult to add knowledge that was not deemed necessary by the tool designer.

Third, complex operations are frequently done by directly calling on another problem solver. This has several implications:

- These calls are often "hard-wired" or fixed, thus the system cannot select, at run-time, from among a number of problem solvers.

- Problem solvers cannot easily access the knowledge encoded in other problem solvers. For example, a classification problem solver cannot easily access the knowledge in an abductive problem solver.

- A single problem solver always has control of the problem-solving system. For example, if a classification problem solver calls a hypothesis matching problem solver, the latter problem solver has complete control of the system. The classification problem solver cannot make any control decisions until the hypothesis matcher relinquishes control.

Of course, a TSA system can also contain incorrect domain knowledge that can lead to inflexibility. This, however, is true of any system, even a flexible one. The difference between a flexible system and an inflexible system is that the flexible system should be capable of correcting the incorrect knowledge once detected.

7 The Problem-Space Computational Model[1]

Soar is based on the problem-space computational model (PSCM) (Newell, et al., 1991), a methodology designed to support flexible problem solving. In the PSCM, all problem solving is defined as search for a goal state in a problem space. A problem space consists of an initial state, a goal state, and operators that modify the states. For example, Figure 2 illustrates part of the problem space for a simple blocks-world problem. The initial state is shown at the top of the diagram. The desired state is highlighted at the bottom of the diagram. The operators define the edges of the problem space. Each edge represents the application of an operator to a state.

A problem space has two kinds of knowledge: *task knowledge* and *search-control knowledge*. The task knowledge consists of the initial state, desired state, and operators. Task knowledge defines the problem space, i.e., the shape of the tree and which states in the tree are initial and desired states. Search-control knowledge is knowledge about which operator to take from a given state. Thus search-control knowledge affects the search

[1] This section has benefitted from discussions with attendees of the Ninth Soar Workshop held at The Ohio State University, May 10-12, 1991. The inclusion of preferences and learning as elements of the problem-space computational model is based on these discussions.

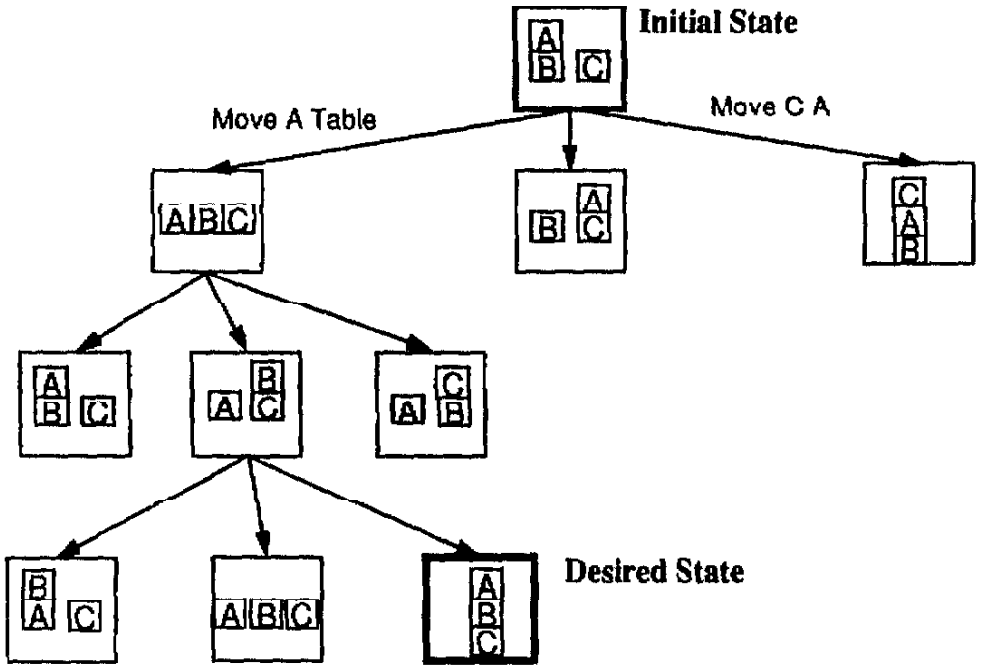

Figure 2: Part of a problem space for a simple blocks-world problem.

through the problem space for a desired state. Note that search-control knowledge affects the efficiency of problem solving, but not the correctness. The correctness of the solution is dependent on the task knowledge that defines the desired state.

Figure 3 shows the steps involved in solving a problem using a problem space. Each of the four boxes represents a PSCM function. The arrows show how the functions are related. To begin, *formulate-task* selects a problem space, the initial state, and the set of desired states. *Select-operator* then selects an operator to apply to the state and *apply-operator* applies it to produce a new state. *Terminate-task* checks to see if the new state is one of the desired states or if success is not possible. If either is true, *terminate-task* halts; otherwise, control loops back to *select-operator* so that an operator can be selected to apply to the new state.

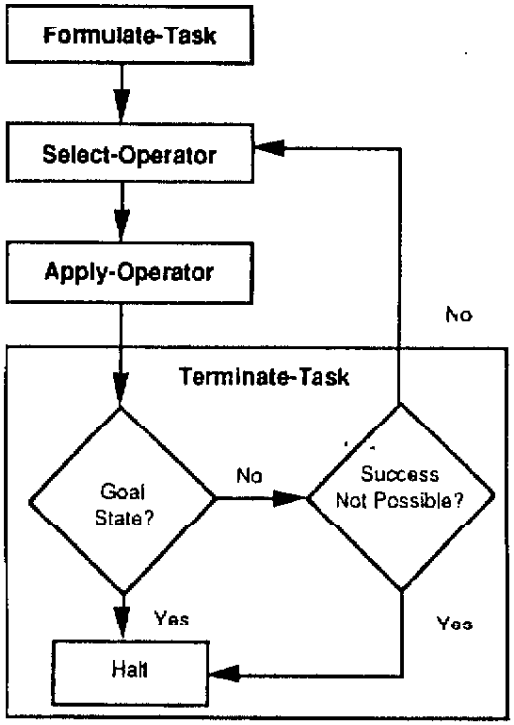

Figure 3: PSCM function flowchart.

To talk about and write down the search-control knowledge that a problem space has we need a way to represent this knowledge. At the PSCM search-control knowledge is described using preferences. A preference represents knowledge about the desirability of objects. There are several different kinds of preferences. Each kind is described below along with an example illustrating how it could be applied to the selection of an operator.

- *acceptable*—Good enough to be considered. If an operator has an acceptable preference, then it can be applied to the current state.

- *reject*—Not to be selected. If an operator has a reject preference, it will not be applied to the current state.

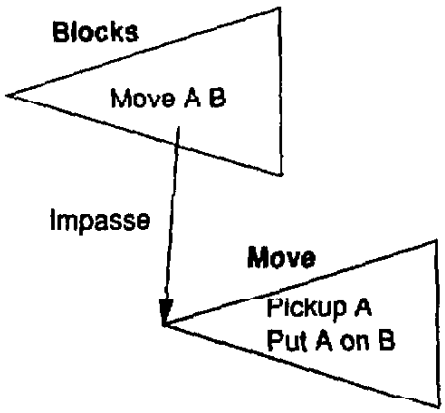

Figure 4: An impasse to acquire knowledge.

- *best*—The object is better than all other objects. If O_1 has a best preference, then it is better than all operators.
- *better*—O_1 is better than O_2.
- *indifferent*—O_1 is indifferent to O_2. This means that it does not matter which operator is selected.
- *worse*—O_1 is worse than O_2.
- *worst*—O_1 is worst (worse than all others).

A problem space must have knowledge to implement each of the PSCM functions. If a problem space does not have the knowledge to implement a function, then an impasse occurs—no further problem solving can be done until the impasse is resolved. Impasses are resolved by formulating a subgoal to acquire the missing knowledge. The subgoal is set up as a task to be solved by another problem space. For example, Figure 4 shows what happens when a problem space does not have the knowledge needed to apply an operator. The

top problem space, *Blocks*, is attempting to apply *Move A B*, but lacks the knowledge needed to do it. A subgoal is automatically formulated to acquire this knowledge. A second problem space, *Move*, is selected to achieve the subgoal. *Move* has knowledge about how to apply the *move* operator. It does this by applying two operators in sequence: *Pick-up A* and *Put A B*. This results in successful application of the operator (*Move A B*), and hence the impasse is resolved.

Several types of impasses can occur:

- **Tie:** Arrises when preferences do not distinguish between two or more objects with acceptable preferences. For example, if two operators have acceptable preferences and there are no other preferences, then a tie impasse occurs, because the knowledge encoded in the preferences is insufficient for selecting a single operator.

- **Conflict:** Arises when there are conflicting preferences. For example, if O_1 is better than O_2 and O_2 is better than O_1, a conflict impasse occurs.

- **No-change:** Arrises if a problem space, state, or operator cannot be selected, or if the current operator cannot be applied.

Whenever an impasse is resolved (such as that shown in Figure 4) learning takes place, resulting in a transfer of knowledge from the subspace to the superspace. For example, in Figure 4 the knowledge about how to move blocks is transferred from the *Move* space to the *Blocks* space. After learning, the *Blocks* space has acquired the following knowledge:

To *Move x to y*:

If *x* and *y* are clear then *Pickup x* and then *Put x on y*.

Note that inductive generalization has occurred. The system induces that all clear blocks can be moved in the same manner as *A* and *B*.

7.1 Advantages and Limitations

The main advantage of the PSCM is that it provides the potential for flexible behavior. Complex problem solving can be used to determine potential actions as well as to select an action from a list of potential actions. The separation of task and search-control knowledge makes it easy to modify the behavior of the system by adding new operators or new search-control knowledge.

Another advantage is the transfer of knowledge achieved through learning. This allows complex problem solving to be avoided. Without learning, a problem-space system would be forced to forever deliberate—impasses could not be avoided.

The main limitation of the PSCM (with respect to building knowledge systems) is that it does not supply a content theory for high-level tasks. For example, the PSCM says little about how to do classification, or solve diagnosis problems.

8 Task-Specific Architectures for Flexible Systems

To produce TSA's that can be used to build flexible systems, we have followed two main steps. First, the GT of interest is reformulated to be as flexible as possible. This means that any unnecessary constraints on the knowledge and method are removed. Second, the GT is implemented as a TSA in Soar. The ability of the TSA to produce flexible systems depends on both of these steps. The first step increases the flexibility of a single GT; the second step provides an implementation that retains this flexibility and allows multiple TSA's to work together.

8.1 A New Language for Describing GT's

We have found it useful to use the language of problem spaces to describe reformulated GT's. A problem space description allows us to specify the method in terms of po-

ential subgoals and preferences on the order of the subgoals without forcing us to specify any unnecessary commitments. Once the GT is properly reformulated and described as a problem space, it can be implemented in Soar in a straightforward manner.

Note that the use of the problem-space language is useful regardless of the means used to implement a GT. The problem-space language implies only that a GT be viewed as having subgoals, knowledge about how to achieve the subgoals, and knowledge about how to order the subgoals. In order to specify flexible GT's, a language of this type must be used—a procedural language forces us to add too many constraints to the specification.

Originally, GT's were specified by giving the type of goal, the problem-solving method, and the kinds of knowledge. This is what we did earlier in the paper when we presented the GT used for CSRL. Using the problem-space language, we now specify a GT by describing knowledge of: the initial state schema; the desired state schema; the operators that apply to states; and how to select from a set of potential operators (*search-control knowledge*).

Initial State Schema: A description of the initial *knowledge state* for the task. A knowledge state is a description of state in terms of it's knowledge content, not it's representation. The schema characterizes all possible initial states (or inputs) for the task. The state can contain knowledge about a real world state, an agent's internal knowledge, or a combination of both.

Desired State Schema: A description of the desired knowledge state for the task. Just as with the initial state, the desired state schema characterizes all possible desired states for the task.

Operators: The operators specify the subgoals of the GT method. An operator modifies a knowledge state to produce a new knowledge state. The modification consists of any number of deletions and additions to the knowledge in the state. If knowledge is added, the operator description must include a specification for that knowledge. Operators can have preconditions and/or enabling conditions. The

Table 1: How elements of the standard GT language map into the PSCM language.

Standard GT Elements	GTPS Elements
Type of Goal	Initial State Schema Desired State Schema
Problem-Solving Method	Operators Search-Control Knowledge
Kinds of Knowledge	The other GTPS elements define the kinds of knowledge needed to use the GT.

preconditions must be satisfied before the operator can be used. The enabling conditions indicate when an operator is to be considered.

Search Control Knowledge: Knowledge about how to prefer operators based on the current state and competing operators. Search control along with operator preconditions and enabling conditions serve to sequence operators. Only search control that is absolutely essential needs to be specified for a GT. Thus, if a GT has nothing to say about how a particular set of subgoals (operators) should be ordered, then nothing needs to said about them. The search control language allows statements of the form: O_1 is better then O_2; O_2 is worse than O_3; O_4 is best; O_5 is worst; and O_6 and O_7 are equivalent.

There are three important points about this GT description language. First, every feature of the old GT specification language is present in the new one. Table 1 shows how each element of the old description maps into the new description. Second, the new description is given entirely in terms of knowledge. In the past, we were often forced to use symbol-level terms when describing the problem-solving method. Third, the new description of the problem-solving method (in terms of operators and search control) allows us to describe the method without giving an overconstrained procedural description of the steps needed to do the task.

To Illustrate these three points, consider the following reformulation of the hierarchical classification GT used in CSRL.

Initial State Schema: The initial state contains knowledge about the data being classified. Possibly knowledge of an initial hypothesis for the data.

Desired State Schema: The desired state contains knowledge about which category the data best describes.

Operators: The classify problem-space uses 3 operators:

suggest-initial-hypotheses: Add one or more initial hypotheses to the state. Hence, the system needs to know of a general class that the data falls into. This should only be used if the state has no hypotheses. Each hypothesis in the state must have an indication of whether it has been tested and refined.

establish *hyp*: Determine whether the hypothesis, *hyp*, should be confirmed or rejected. This is to be used for any hypothesis in the state that is not yet confirmed or rejected.

generate-refinements *hyp*: Generate (add to the state) those hypotheses that should be considered as a refinement of *hyp*. This is used for all of the confirmed hypotheses in the state.

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

Search Control Knowledge: No search control is specified. The only sequencing of operators is that imposed by the preconditions of the operators. If *generate-refinements* and *establish* operators are applicable at the same time, additional knowledge must be used to select one; however, this knowledge is not an essential part of the GT so it is not specified. This allows the designer of a system to add system specific search control.

In a GT system, like the system shown in Figure 1, there might be alternative GT's

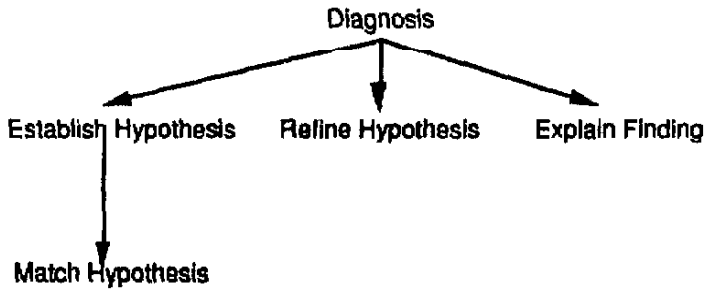

Figure 5: An integrated task-structure for diagnosis.

for implementing the same subgoal. For example, the *establish* subgoal might be done using the match GT or a simulation GT. In these cases, each of the GT's can be specified and we can describe the knowledge needed to select between the GT's. This knowledge is associated with the system, not the GT.

The new GT language also allows us to describe tightly integrated systems. For example, the diagnostic system in Figure 1 completely separates the classification GT from the best explanation GT. Figure 5 shows a new task structure for diagnosis. The new task structure mixes the goals of the classification GT and the best explanation GT. This means that the system can mix classification problem solving with best explanation problem solving.

8.2 Implementing TSA's in Soar

The implementation of TSA's in Soar follows directly from the GT specification. Each GT is implemented as a problem space with the specified operators. A representation for both the states and the operators must be selected. All of the knowledge to propose and select operators and alternative GT's is encoded in productions. Knowledge of the desired state is encoded as a production that tests the current state of the space to see if it is a member of the set of desired states.

So far we have implemented TSA's in Soar for classification, abduction (finding a

best explanation), and hypothesis matching (Johnson, 1991; Johnson and Smith, 1991). To use a tool, the system builder must specify how all the operators are implemented, as well as, any additional preferences for the ordering of the operators. To assist tool users, we have provided several default techniques for implementing common GT subgoals.

We have also used these tools to implement an integrated system, RedSoar (Johnson, et al., 1991), which has a task-structure similar to the diagnostic system described in Figure 5. The flexibility of the Soar implementation allows us to easily experiment with different control strategies as well as strategies that mix subgoals from several GT's.

9 Discussion

It might seem strange to build task-specific architectures in a general architecture. After all, TSA's were motivated by the problems of general architectures. Actually, building TSA's in Soar is not so different: first, TSA's were a reaction against the single level view of problem solving systems often provided by general architectures, not against the utility of a general architecture. The Soar versions of our TSA's retain the multilevel task-dependant view. Soar is not just a collection of rules or problem-spaces. A Soar program has specific goals, and representations. Second, TSA's have always been built on top of some kind of general architecture, be it Lisp, an object system, or some other kind of programming language. Soar is just as valid an implementation medium as these other languages are. Third, just as TSA's are good for building systems that solve a particular kind of task, Soar is good at building flexible systems. If TSA's are so useful because they provide special support for a task, then certainly we should not ignore general architectures that provide support for constructing flexible TSA's.

The new TSA's combine the advantages of the GT approach with the advantages of the Soar architecture. Knowledge acquisition, ease of use, and explanation are all facilitated because subgoals of the problem-solving method and the kinds of knowledge needed to use the method are explicitly represented in the GT description and 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 the TSA's easy to understand and use.

The new TSA-based systems overcome many of the limitations 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 (i.e., GT's) 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 such as lookahead to see which is better. Automatic goal termination provides an integration functionality not available in previous GT architectures. In general, the integration capabilities of the new tools are greatly enhanced. Because of preferences and the additive nature of productions, new knowledge can be added to integrate multiple tools without modifying existing control knowledge.

10 Acknowledgments

We thank the members of the Division of Medical Informatics and the Soar Community for their comments and discussion on this paper. This research is supported by National Heart Lung and Blood Institute grant HL-38776, National Library of Medicine grant LM-04298, Defense Advanced Research Projects Agency grant F49620-89-C-0110, and Air Force Office of Scientific Research grant 89-0250.

Bibliography

Brown, D. C., and Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies*. San Mateo, California: Morgan Kaufmann Publishers.

Bylander, T., and Mittal, S. (1986). CSRL: A language for classificatory problem solving. *AI Magazine, VII*(3), 66-77.

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

Chandrasekaran, B. (1989). Task-structures, knowledge acquisition, and learning. *Machine Learning, 4*(339-345), 93-99.

Clancey, W. J. (1985). Heuristic classification. *Artificial Intelligence, 27*(3), 289-350.

Johnson, K. A., Johnson, T. R., Smith, J. W., Jr., DeJongh, M., Fischer, O., Amra, N. K., and Bayazitoglu, A. (1991). RedSoar—A system for red blood cell antibody identification. In *Proceedings of SCAMC 91*. Washington D.C.

Johnson, T. R., Smith, J. W., and Bylander, T. (1989). HYPER—Hypothesis matching using compiled knowledge. In W. E. Hammond (Ed.), *Proceedings of the AAMSI Congress 1989*, (pp. 126-130). San Francisco, California: American Association for Medical Systems and Informatics.

Johnson, T. R., and Smith, J. W. (1991). A framework for opportunistic abductive strategies. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, (pp. 760-764). Chicago: Lawrence Erlbaum Associates.

Johnson, T. R. (1991) *Generic Tasks in the Problem-Space Paradigm: Building Flexible Knowledge Systems While Using Task-Level Constraints*. Ph.D. Dissertation, The Ohio State University.

Josephson, J., Chandrasekaran, B., Smith, J., and Tanner, M. (1987). A mechanism for forming composite explanatory hypotheses. *IEEE Transactions on Systems, Man, and Cybernetics, 17*(3), 445-454.

Kingsland III, L. C., and Lindberg, D. A. B. (1986). The criteria form of knowledge representation in medical artificial intelligence. In *MEDINFO 86*

Laird, J., Congdon, C. B., Altmann, E., and Swedlow, K. (1990). *Soar User's Manual Version 5.2* (Technical Report No. CMU-CS-90-179). Carnegie Mellon.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence, 33*, 1-64.

Marcus, S. (1988). Salt: A knowledge acquisition tool for propose-and-revise systems. In S. Marcus (Eds.), *Automating Knowledge Acquisition for Expert Systems* (pp. 81-123). Boston: Kluwer Academic Publishers.

McDermott, J. (1988). Preliminary steps toward a taxonomy of problem-solving methods. In S. Marcus (Eds.), *Automating Knowledge Acquisition for Expert Systems* (pp. 225-256). Kluwer Academic Publishers.

Newell, A., Yost, G., Laird, J. E., Rosenbloom, P. S., and Altmann, E. (1991). Formulating the problem space computational model. In R. F. Rashid (Eds.), *Carnegie-Mellon Computer Science: A 25-Year Commemorative* Reading, MA: ACM-Press: Addison-Wesley.