

Chandrasekaran, B. & Johnson, T. R. (1993). Generic tasks and task structures: History, critique and new directions. In J.-M. David, J.-P. Krivine, & R. Simmons (Ed.), *Second Generation Expert Systems* (pp. 232-272). Berlin: Springer-Verlag.

Generic Tasks and Task Structures: History, Critique and New Directions*

B. Chandrasekaran¹ and Todd R. Johnson²

¹Department of Computer and Information Science, Laboratory for Artificial Intelligence Research, The Ohio State University, Columbus, OH 43210, USA

²Department of Pathology, Laboratory for Knowledge-Based Medical Systems, The Ohio State University, Columbus OH, 43210, USA

Abstract. We have for several years been working on an approach to knowledge system building that argues for the existence of a close connection between the tasks which the knowledge system is intended to solve, the methods chosen for them and the vocabulary in which knowledge is to be modeled and represented. We trace the historical origins of the idea that we have called *Generic Tasks*, and outline their evolution and accomplishments based on them. We then critique their original implementations from the perspective of flexible integration. We follow this with an outline of our current generalization of the view in the form of a theory of *task structures*. We describe the architectural implications of this view and outline some research directions.

1 Introduction

We and our colleagues have been working for about a decade on an approach to the construction of knowledge systems that can be best characterized as *task-oriented*. In this approach, the nature of the task that is set for the knowledge-based system (KBS) takes primacy in analysis. The properties of the underlying general architecture on which the knowledge system is implemented are considered relatively unimportant. Our work has gone through much evolution, and in fact, recently, the evolution has been many-branched, i.e., different researchers are taking somewhat different paths.

Our work has had three phases: initial work on MDX and associated systems during which we developed our understanding of the importance of a task-oriented perspective; a second phase in which the task-level emphasis was made explicit and a specific version of it, the Generic Task (GT) approach, was developed in some detail; and the third, current, phase, in which the GT approach is evolving into the broader view that we call *Task Structures* which places greater emphasis on multiplicity of methods for tasks and flexibility in invoking them. The current phase has also introduced new points of view regarding the relationship of the task-oriented view to general architectures.

* Portions of this paper appear as part of Chandrasekaran, B., Johnson, T.R., Smith, J.W.: Task-structure analysis for knowledge modeling. Communications of the ACM, (September) 1992.

In writing this paper, we have assumed a general familiarity on the part of the readers with our work on the first two phases, which is well-documented in the literature. What we plan to do is to quickly review the history of the first two phases, then spend more time on our recent concerns and ideas.

2 Generic Tasks: A Historical Overview

2.1 Diagnosis as Classification

In the late 70's, inspired by the work of the Stanford group on Mycin and related diagnostic systems, we began to investigate medical diagnostic problem solving. Specifically, at a seminar during 1977, at the suggestion of Jack Smith, our medical collaborator, we examined some solution protocols from Harvey and Bordley [35], a book of medical case analyses. Fernando Gomez, a member of our group, proposed that the reasoning involved in the cases that we looked at could be viewed as a form of classification reasoning, i.e., one in which a set of data describing the case (symptoms or observations) was classified into one or more disease classes. (This was some time before Clancey published his analysis of Mycin [23] as a heuristic classification problem solver.) Gomez and Chandrasekaran [32] also proposed that the diagnostic knowledge could be organized as a classification hierarchy of disease concepts, and implemented as a community of specialists, with one specialist for each diagnostic concept. The specialist would have procedural knowledge about how to establish the concept, and if the concept is established, control would be transferred to the child concept. The idea of concepts as specialists has many reverberations from the past and the present: Wittgenstein's ideas [74] on the active nature of concepts, Minsky's view of a Society of Mind [51], and object-oriented programming are some examples that come to mind.

2.2 Diagnosis as Classification + Intelligent Database

Our medical diagnostic system, MDX [19] [18], was based on the classification perspective. Sanjay Mittal, who took on the task of implementing MDX soon saw the need for another type of reasoning to augment classification. Classification specialists had knowledge that helped map from the case data to confidence values in various hypotheses, but the types of case data that the classification specialists could handle were often different from the observations that described the case. The observations had to be normalized or otherwise converted to a form that the classification specialists could handle. Otherwise the procedures in each of the classification specialists would have to contain many combinations to account explicitly for each form in which the data could appear as observations. To give a simple example, a classification specialist in the medical domain might need information about exposure to anesthetics, while the case data might not have any information about this, but instead might record that the patient recently underwent major surgery. The latter information can be used to infer exposure to anesthetics.

Mittal [18] decided to create an "intelligent database" that would use domain knowledge to make inferences from available observations about the presence or absence of information that the classificatory specialists could use. Separating the database in this

way enabled us to keep the classificatory activity perspicuous. Mittal implemented the database in the same specialist-community style as the classification module. The database had specialists that specialized in different types of medical data. These specialists were organized in a way that reflected the hierarchical and other relations between the types of data. Inferential knowledge about data was represented at different levels of abstraction in the various data specialists. Once we modularized the database this way, we could identify more examples of reasoning that could be supported by the database, e.g., spatial and temporal reasoning about data [18, 20, 52, 53]. Spatial reasoning was needed for reasoning about radiographic information, e.g., to infer obstructions in the bile duct from descriptions of shapes and location of masses in x-rays. Temporal reasoning was needed to answer questions about temporal relations between data; for example, to decide if some quantity was rapidly rising or not based on a sequence of its values.

About the time we were developing this view of diagnosis as classification, Clancey was making his own analysis of Mycin as a *heuristic classification* system. He decomposed heuristic classification into three subtasks: data abstraction, heuristic match (of data to categories), and (category) refinement. While our classification problem solver was explicitly performing both concept-establishment and concept-refinement, and our data retrieval module was performing several other types of inferences in addition to data abstraction, MDX and Clancey's Neomycin system were talking about the same kinds of subtasks and at approximately the same level of abstraction. (However, the order in which subtasks were invoked was quite different in the Neomycin and MDX implementations. The similarities are at a fairly abstract level of the types of subtasks.)

2.3 Diagnosis as Abduction

As the range of diagnostic problems we considered widened, we started to look at diagnostic problems for which classification was not enough. MDX could handle multiple malfunctions as long as the malfunctions were more or less independent. The kind of diagnosis that, for example, Internist [50] and Abel [57] could perform, where the diagnoses involved selecting subsets of hypotheses from among relevant hypotheses, was beyond MDX's capability. In [32], we recognized the need for an Overview problem solver which could perform an evaluation of how well the hypotheses were explaining the data, but we made no suggestion about how the problem solving would proceed.

At about this time, John Josephson joined our group. His philosophical interest was in using abductive reasoning, i.e., reasoning to the "best explanation," as the central strategy in arriving at increasingly reliable knowledge in spite of uncertainty. Pople [60] had earlier shown that the diagnostic problem had a strong abductive character. Our diagnostic approach was ready for the addition of the abductive view. It turned out that we could view the classification problem solver as a module that generated a number of highly plausible hypotheses along with an account of what each hypothesis could explain. We added a new problem solver, an *abductive assembler*, that would select the best subset of these hypotheses by reasoning about which subset explained the data most satisfactorily. Again, following the paradigm already set, we looked at abductive assembly as a distinct type of problem solving, and started identifying the types of knowledge needed and the kinds of strategies that were available to perform the

task. Abduction evolved, over the next few years, into one of the most active areas of research within our Laboratory. It was during this time that Josephson, Smith, and a number of graduate students developed a complex abductive system called RED. (The results of almost a decade of our Lab's research on abduction are summarized in [45].)

2.4 Hypothesis Assessment by Hierarchical Matching

Diagnosis as classification requires as a subtask assessing the concepts in the classification hierarchy, or, more precisely, requires determining how likely a given classificatory concept is, given the case data. Mycin had popularized a form of certainty factor calculus for performing this task, and a number of probabilistic techniques had been used in traditional pattern recognition for this subtask. In MDX this was accomplished by a form of template matching against the concept description. The technique [9, 21] combined qualitative measures of the presence of data ("high," "medium," etc.) to reach conclusions about hypotheses with a qualitative measure of certainty attached to the conclusions ("likely," "unlikely," etc.). The essence of the matching was hierarchical, feature-based pattern matching. This technique side-stepped the need for assigning numerical values and combining them when the data didn't have such numerical precision and when the task only called for qualitative evaluation. We saw that this type of problem solving had the potential for wide use, not only as part of diagnosis, but wherever hypotheses or concepts of any sort had to be matched against a situation description. Later this technique was generalized to *structured matching*, in which one choice out of a small number of choices is made by hierarchically matching features.

2.5 Routine Design: Plan Selection, Instantiation and Refinement

Around 1981, David Brown took on as his thesis topic design as a possible application area for KBS's. Brown and Chandrasekaran analyzed the problem solving activity of an expert designer of a mechanical device called an air-cylinder and identified the design activity as a form of routine design [5]. We saw that, similar to our analysis of classification, the designer's knowledge could be decomposed into a number of design concepts that were hierarchically organized, reflecting the hierarchical structure of the object that was being designed. Again, this form of problem-solving activity had characteristic types of knowledge (for example, *plans*), problem-solving subtasks (*plan selection*, *plan refinement*), and problem-solving strategies (*top-down design*).

2.6 Generic Tasks: The Initial Formulation

By 1983, we had gained experience with several tasks. In particular we were beginning to learn how to decompose a complex task into component tasks. Chandrasekaran had started formulating the notion of a *Generic Task* [11, 13]. The main intuition was that classification, data retrieval, plan selection and refinement, state abstraction and abductive assembly all were in some sense re-usable subtasks. These subtasks were proposed as especially useful as components in other more complex problem-solving tasks. In our work on diagnosis we had shown how classification, data retrieval and abductive assembly came together. The work on design also used a form of intelligent data man-

ager in conjunction with the planner. We viewed these generically useful components as building blocks and called them Generic Tasks (GT's). We knew that we did not have an exhaustive list of GT's yet, but we were confident that we had identified some important ones and that they were illustrative of the kind of generic components that we should be looking for.

Note the distinction between tasks, such as diagnosis and design that are in some sense generic, and Generic Tasks. Diagnosis and design were not in our list of GT's. We viewed them as compositions of GT's of the type that we had identified. One of the criticisms often made about the GT approach [37] is that it would lead to multiple representation of the same knowledge. The example often used involved the use of structural knowledge of a device in both diagnosis and design. If one built task-specific knowledge representations for each of these tasks, the argument went, one would need to replicate the structural knowledge in the two modules. However, as we have shown elsewhere in detail [17], both diagnosis and design use structure-to-behavior simulation as a subtask, and structural information is the knowledge that is needed to carry out this subtask. If we create a module for this type of simulation that can be invoked during design or diagnosis as needed, structural knowledge need only be represented once in this module.

The specialist architecture that we had adopted was particularly helpful in composing the GT's. Message passing between specialists was the glue with which to compose the specialists of different types. However, we soon started to recognize that, in identifying the task-level view so closely with the specialist-style implementational approach, we were mixing implementation-level talk with task-level talk. We began to describe [12] each GT in terms of the input-output description of the task, the method/strategy that was appropriate for it and the knowledge that the method needs.

Generic Task Shells. Since each GT had a clear characterization in terms of types of knowledge needed and some family of parameterizable methods, it was a natural step to propose a shell for each of the GT's that we had identified. Bylander, with the design assistance of Mittal, built the first such language, CSRL [10]. It enabled a knowledge engineer to encode classification hierarchies in the domain of interest and either accept the *Establish-Refine* strategy that was the applicable default strategy for hierarchical classification, or specify local variations on the strategy. When the domain-specific knowledge was encoded and the method was specified using CSRL, the compiler produced a classification problem solver. Brown [4] similarly built DSPL, a language in which to specify domain knowledge and control for building routine design problem solvers. In later versions of CSRL and DSPL, graphical user interfaces of considerable sophistication were added which relieved the system builders of much of the need to do "programming." A separate shell called HYPER [41] was built for the hypothesis assessment task.

We designed PEIRCE as a shell for constructing abductive assembly systems [61]. Sticklen built a data-base shell called IDABLE [38] for constructing intelligent data base modules as data servers for knowledge systems.

This general style of paying attention to the knowledge needs of the task and characteristic strategies was a hallmark of our approach to a number of other problems in knowledge systems. For example, Sembugamoorthy and Chandrasekaran identified a form of functional reasoning as a generic activity that was useful in simulation of devices [65], and we identified the types of knowledge needed to represent knowledge

about how devices worked. Allemang built a generic shell to support the acquisition and use of functional representations.

The GT shells found wide use in chemical engineering, nuclear engineering, medical applications, speech recognition, planning and design. Josephson led an effort to build an integrated toolset which was based on a uniform agent-oriented formalism to implement a specialist architecture [46], while Sticklen and Punch at Michigan State moved to build a Smalltalk-based GT toolset. In addition, a number of organizations over time also implemented their own versions of the shells.

Advantages of GT's. It seemed to us that we were engaged in a new approach to knowledge representation (KR). In traditional KR, knowledge was represented independently of the task. Since there was no theory of tasks to help in identifying types of knowledge, the only terms that were available were very general and task-independent, such as predicates, sets and set membership and subset relations. On the other hand, because of the emphasis on tasks and the role different types of knowledge played in their achievement, our approach to knowledge representation provided a larger vocabulary of task-related terms, and additionally, related the knowledge to how it was going to be used. As we develop an understanding of more such tasks, more knowledge-level terms will be identified for the representation of knowledge.

Bylander and Chandrasekaran outlined [8] how the GT view facilitated knowledge acquisition by providing the vocabulary in terms of which to seek knowledge for the task, and by guiding in the organization and use of the knowledge thus acquired. Chandrasekaran, Tanner, and Josephson [22] similarly showed how the task-view also provided important points of leverage in the generation of explanations of problem solving.

The GT's also appeared to have computational advantages. Goel *et al.* [30] showed how, if the right kind of knowledge was available, hypothesis generation by classification had a computational complexity that was linear in the number of hypotheses. Goel and Bylander similarly analyzed the computational properties of structured matching [28] (see Section 2.4). Bylander *et al.*, showed in a series of papers which culminated in [7] that many of the strategies used in the construction of our abductive assembler (see Section 2.3) had attractive computational properties, explaining how knowledge of the right type can help solve problems in acceptable time even though the general abductive problem was NP-complete.

We now had a system-building style in which we had a general architecture of message-passing modular specialists. Each GT language was built by specializing this architecture for the GT at hand into a task-specific architecture (TSA). This architecture, when instantiated with domain knowledge, produced a problem solver for the corresponding task in that domain. Our attitude to general purpose architectures at this point in time was two-fold. One, they strictly served the role of universal machines in enabling the construction of TSA's. It didn't matter whether the underlying architecture was LISP, Prolog, a rule-interpreter, or whatever. Two, emphasis on general purpose architectures often made people view what were essentially content¹ issues at the task

¹The term "content" is often used in AI discussions to contrast it with "form." The content of a representation is roughly what types of information is carried by the representation, while the form is the syntactic aspects of how the representation is encoded. "Form" also corresponds to a computational architecture in that there is a correspondence between architectures and programming languages. Newell's Knowledge vs. Symbol-level

level as syntactic issues in the programming language corresponding to the underlying architecture.

2.7 GT's as Functional Building Blocks: The Platonic View

In [14], Chandrasekaran outlined a view of GT's as “building blocks” of intelligent systems. We can call it the Platonic View since it proposed that a set of abstract GT's existed which together functionally captured the capability of intelligence in solving problems. The proposal was explicitly *not* claiming that the GT's we had identified were all that were there, but we hoped that with further work we would be able to identify a number of such strategies that were sufficient to cover a large part of problem solving. GT's such as classification, plan instantiation and refinement, and abductive assembly seemed ubiquitous as components in many different problem solving tasks. They all could be defined functionally, thus avoiding the vexing problem of the right general architecture. They all had attractive computational properties. The GT's that we were working with seemed more than accidental, *ad hoc* entities.

2.8 Other Work in the Same Spirit

By 1987, Newell's paper on the Knowledge Level [56] had become more widely known and read in AI. In our group, we realized that we had in fact been working at and arguing for precisely the knowledge-level view of knowledge systems. Our arguments against general purpose architectures, which were the bones of contention in the field, could now be seen as arguments against a premature commitment to symbol-level issues.

From 1987-89, we became aware of a number of efforts outside our Laboratory to investigate knowledge-based problem solving at the Knowledge Level. We have already mentioned Clancey's work on Heuristic Classification and the Heracles shell based on it. KADS work in Europe [3] proposed a number of primitive inference terms to use in describing the problem solving behavior of agents. (Hadzikadic [34] in the US had also made a similar proposal, but it was not widely known.) The KADS work at that time seemed to have somewhat different goals from those of the GT work. For one thing, it offered to provide support for analyzing and describing problem-solving behavior in terms of generic inferences, whereas the GT approach offered direct support for implementation in the form of building blocks that could be used to instantiate problem solvers. There were no proposals in KADS of that period about strategies or methods by which problems were solved. The KADS inference primitives seemed much more fine-grained than GT terms such as classification and plan refinement, i.e., they seemed to be at the level of internal operators that our GT's were using. However, there was

distinction is one attempt to capture this distinction more formally. What is content at one level may be form at another level. Consider an assembly language, LISP that compiles into the assembly language, and a knowledge representation language such as KL-ONE written in LISP. LISP has a content theory of some computational objects, but KL-ONE is a further content theory for which LISP provides the formal substrate. KL-ONE is in turn a formal substrate for a representation of knowledge in some domain in KL-ONE.

no obvious direct mapping from the operators that GT's were using and the inference primitives in KADS. Over time, the KADS view has evolved into a more comprehensive framework, called KADS-2 [72], which has many layers spanning from analysis through strategies to implementation. KADS-2 has a clear role for methods in the spirit of GT's in what they have called "the blue book," a compendium of abstract methods of general usefulness. But there is still an unresolved issue of the status of the inference primitives from KADS-1 which now reside in one of the layers of KADS-2. Their status as a closed and orthogonal set of ontological primitives, how they arise and how they relate to other such sets of primitives are all unclear and need further research. We come back to this issue again in the last section of the paper (Section 5.1).

The role of the structure of tasks in guiding the construction of knowledge systems began to be investigated by many other researchers. McDermott and Marcus [48, 49] wanted to know what role different types of knowledge played in different types of methods. They identified very general methods such as "Propose-and-Refine" for configuration problems, and identified the role knowledge played in the achievement of such methods. In France, David and Krivine [26] worked on TSA's for diagnosis. They made a useful distinction [25] between functional architectures at the knowledge level for a task, and GT's as particular components of functional architectures that have proved to be recurrent and ubiquitous in various knowledge systems performing the task. Gruber and Cohen [33] identified uncertain reasoning as a generic activity and proposed generic methods and representations to handle this problem. Musen's work on Protege [54] was quite close in spirit to DSPL. Steels' work on componential frameworks [66] was to come later, but it also shared the spirit of the GT work (we discuss this approach later in the paper as well). Thus a community was emerging with similar goals, with approaches that shared some ideas and differed on others. Specifically, they shared two features. First, they identified tasks at various levels of abstraction above the implementation language level. Second, they identified types of knowledge and reasoning strategies needed to accomplish such tasks.

Later in the paper, we discuss in some detail the relation of our approach to KADS, the componential framework and role-limiting methods.

2.9 Critiques of the GT Approach

As experience accumulated in our Lab in using the GT approach for a variety of complex problems, a period of assessment began from 1988 onwards. While there was widespread acknowledgment of the utility of the GT approach, there were critiques of aspects of the approach from researchers outside our Laboratory as well. We have already dealt with a recurrent criticism that the GT view might lead to redundancy of knowledge representation. What we now want to discuss is the series of critiques of GT from those participating in its development.

The first set of questions concerned the criteria for a GT: How many GT's are there? What kinds of tasks count as GT's? (See Sections 3.7–3.9 for the answers that we ended up with to these questions.) In one sense, there are just two generic tasks: abduction and planning. All agents have to make sense of the world and build more or less veridical internal models of the environment (the problem of abduction), and using such models plan actions on the world to achieve goals (the task of planning). In this view, GT's such as classification are a means to an end, subtasks of the two major tasks. This sense of generic tasks, which emphasizes goals, is different from the build-

ing block view of GT's that we were promulgating and that emphasized generic capabilities useful for a number of goals.

We also became increasingly aware of a terminological problem: the conflation between a task and a method in the GT way of speaking. Each GT such as classification was both a task (classification had the goal of identifying a class that best characterized a set of observations) and a method (in CSRL, the classification task was performed by the hierarchical classification method). Of course the methods could set up subtasks (CSRL set up the subtask of data abstraction/retrieval). Classification used as a building block for the task of diagnosis was a method for the subtask of hypothesis generation, but it also came with the method of hierarchical classification. In fact one of the proposed strengths of the GT view was that each such task came with a preferred default method. Nevertheless it was becoming clear that we needed to make a clear distinction in our discussions between tasks and methods.

Once we started looking at the methods inside GT's separately, it was clear that the rigidity of the methods for GT's was becoming a problem. It is true that the top-down method of *Establish-Refine* is a good general strategy for hierarchical classification, but there are instances in which a different control strategy is needed. For example, at times a disease at a higher level can only be established if any of its subtypes can be established. A knowledge engineer should be able to specify different strategies.

As a result, we gave CSRL increased flexibility in method specification, ways in which the default strategy can be modified. The theoretical issue was how far flexibility could be pushed within the framework of the GT and the corresponding notion of a task-specific architecture.

There was another aspect to the flexibility issue: flexibility in the way different GT problem solvers were hooked up to solve a complex problem. In building our diagnostic systems, we would typically know at system-building time how these components would be put together: for example, for diagnosis, the abductive assembly system would invoke the classifier for hypothesis generation, the classifier would invoke the appropriate hypothesis assessors for evaluating the various hypotheses, and so on. Later in MDX-2 [67], when knowledge was missing in any of the components, the functional model of the device under diagnosis would be invoked to derive the missing knowledge. However, it was becoming increasingly clear that a more opportunistic, run-time choice of methods could be beneficial in several applications.

"Chunkiness" of some of our GT's was also becoming a problem. We noted that our classification tool, CSRL, was actually two GT's in one: one for the problem of navigating classification hierarchies and the other for the problem of assessing each of the hypotheses selected during navigation. It seemed wise to separate out the assessment GT from CSRL. This was in fact done quite early in our work, during about 83-84. The hypothesis assessment GT, along with its symbolic abstraction method, became known as HYPER [41]. It found extensive use wherever we found the task of symbolic assessment of how well a set of data fits a hypothesis, concept or a plan.

The chunkiness problem was actually quite severe in DSPL. DSPL solved a number of subproblems: plan assessment, plan selection, plan refinement, and failure handling. For each of the subtasks it had a method. When users wanted to adopt a different method for a subtask, DSPL provided escape into LISP for coding the new method. This was of course a good thing, since it avoided the tools becoming representational prisons from which escape was impossible, but the problem is that LISP is not a language that has any theory of problem solving implicit in it. The language into which one escapes should preferably be one which is built on a uniform framework for prob-

lem solving. Thus, it seemed that either we had to provide a larger repertoire of methods, or provide a task-independent problem-solving framework to solve the local problem and get back into the GT tool.

Another problem concerned the multiplicity of ways in which people used the GT's in solving a new problem. In a recent experiment to study how users of GT were using the tools to model real-world problems, a number of GT researchers were asked to design a system to assign employees to various rooms so that a set of constraints are satisfied [1]. Most subjects viewed the problem as an instance of routine design in which plans are selected and instantiated so as to satisfy the constraints. But one GT analyst chose to view the problem as an instance of the assembly problem, i.e., a problem where objects (rooms) are assembled such that a set of constraints are satisfied, and proposed the use of the abductive assembly tool, with the understanding that the intended criterion for assembly was not explanation but coverage of design constraints. We are confronting the issue of what it means for a problem to be a certain type. Is abductive assembly a special type of configuration problem? Is design a particular type of assembly problem? (These issues, by the way, are not unique to the GT approach; they are problems for any general theory of tasks and methods.)

The above considerations were taking their toll on two aspects of our stance. One of them concerned the Platonic View, that GT's were abstract functional building blocks of intelligence, and the other was the associated architectural view that intelligence was a collection of task-specific architectures, each devoted to one of the GT's. Skepticism about this position led to re-examination of our belief that general purpose architectures were just Turing-Universal machines for implementing GT-level TSA's.

It was beginning to seem unlikely that a GT such as routine design (as captured in the tool DSPL) could be a building block: there were too many subtasks and methods rolled into one. What are the elementary tasks and how to tell?

Multiplicity of methods for a given task and emergence of complex methods from more elementary pieces reintroduced the issue of an appropriate general architecture. For handling the multiplicity of methods, we needed an architecture in which methods could be invoked, evaluated and chosen at run-time. If that architecture could also explain how methods that we recognized as GT-like could emerge, and could support writing of new methods in a uniform framework, so much the better.

In response to these questions, the GT approach evolved into a framework that we have called the Task Structure view. In the next section of the paper, we describe this view and its architectural implications.

3 The Task Structure Perspective

In [15] and more elaborately in [16], Chandrasekaran described this form of analysis. Let us first clarify some terminology before describing the task structure approach.

3.1 Tasks

The term "task" has been used in many senses in the field, contributing to some confusion. The term has been used to denote an instance of a problem, a problem class, and both a problem class and an abstract description of a method of solving the problem. Newell and Simon use the term to refer either to a problem instance or a class of

problems of the same type. In Clancey [24], the term task refers to the basic subgoals that can be set by an expert system (e.g., APPLYRULE!, GROUP-AND-DIFFERENTIATE, etc.), i.e., it includes a high level description of the method. In our work on GT's, we also included a method description as part of a Generic Task. Similarly, Wielinga *et al.*, [72] use the term "task" to refer to the sequence of operations (at an appropriate level of abstraction) that a particular system performs. We think it is useful to separate the intended set of problem instances for a knowledge system from the methods used by it to solve them, i.e., separate the task from the method.

In the rest of the paper, we will use the term "task" to refer to a *type* of problem, or equivalently, a set of problem instances with something in common. Thus diagnosis is a task, i.e., a type of problem in which the goal is to identify the causes of a given set of abnormal behaviors of some system. The task can be at different levels of generality. For example, we can have diagnosis, and medical diagnosis, which is a type of diagnosis and so on. How a set of problem instances gets grouped into a type of problem is determined purely on pragmatic grounds of the usefulness of such a grouping. For example, treating diagnosis as a type of problem enables us to study general strategies for that class of problems. Note that we explicitly do not include, as part of the task description, any specification of *how* the task will be accomplished.

3.2 Methods, Operators, Subtasks and Inferences

Methods are ways of accomplishing tasks and may be of many types: they may be computational, or "situated," i.e., involve extracting information from the surrounding physical world. For example, the task of predicting behavior of a device may be solved by a computational method that performs a simulation, or it may be solved by manipulating a physical model of the device and seeing what happens. Within the class of computational methods, a method may be couched as executing a precompiled algorithm, search in a state space, as a connectionist network and so on.

Let us focus on computational methods. Abstractly, such a method can be described in terms of the *operators* it employs, the *objects* that it operates on, and any additional knowledge about how to organize operator application to satisfy the goal. Note that any algorithm can be abstractly characterized in these terms. The problem-space formalism is a general way of formulating such methods. In this model, the goal is described as the desired knowledge state of the problem-solving agent. The initial state is the knowledge state corresponding to what is known at the start of problem solving. Thus the initial state in diagnosis contains knowledge of the symptoms, while the goal state description includes the causes of the symptoms. The operators transform one state into another state. Whenever two or more operators are applicable to a state knowledge that indicates which operator to select is needed. This is called *search-control knowledge* [47] because it indicates how the problem space is searched.

Operators are the *subtasks* of a method. The only difference between the definition of an operator and our definition of a task is that operators also come with preconditions. Nonetheless an operator is a task. It specifies a class of problems to be solved. Thus, we use the terms *operator* and *subtask* interchangeably.

Subtasks can be accomplished either directly or through additional problem solving using a method. A subtask can be accomplished directly if all its preconditions are satisfied and the knowledge to accomplish the subtask is in a form that can be applied

without further problem solving. More commonly, however, subtasks require additional problem solving using a method. A subtask, of course, can have alternative methods associated with it.

It might seem that the problem-space formalism is only appropriate for describing search methods. However, algorithms that do not perform search can still be viewed as special cases where the choice of which operator to apply to which state is completely specified, and all preconditions for operator application are satisfied. Thus the problem-space formalism is sufficiently general to include all algorithms whether they perform search or not.

Often the term “inference step” is used as part of method descriptions, and researchers speak of “inference structures”. This terminology arises from the “reasoning metaphor,” in which agents solve problems by making inferences from available knowledge to generate new knowledge. The agent's goal is to make a series of inferences such that a clause or proposition which meets the informational requirements of the goal is generated as one of the new pieces of knowledge. We can map the reasoning metaphor to the problem-space metaphor as follows. Each state in the problem space corresponds to a certain knowledge state for the agent. Each operator if applicable and applied enables the agent to be in a new knowledge state which typically adds knowledge to previous states. Thus there is a correspondence between “inference rule” in the reasoning view and “operators” in the problem solving view. Corresponding to “operator schemas,” wherein the agent instantiates an operator schema before applying it, we might also have inference rule schemas.

3.3 Example: The Method of Hierarchical Classification

Let us consider how to represent the *Establish-Refine* method for hierarchical classification [32] using the framework described above. Hierarchical classification is used in many diagnosis systems as a way of quickly focusing on possible malfunctions. The initial state of the classification task is a set of data (e.g., manifestations in a diagnosis task) and an initial high level hypothesis (e.g., *liver disease*). The goal state is one containing plausible malfunction hypotheses, i.e., the most detailed hypotheses consistent with the data. The method works by first considering a high-level malfunction category, such as *liver disease*, to determine if the malfunction appears likely given the data at hand. If it appears likely then the malfunction is refined to more specific diseases, *hepatitis* and *cancer*, for example. The more specific malfunctions are then evaluated against the data and any that appear likely are refined. This process continues until no more malfunctions can be refined.

We can specify this method using two subtasks:

evaluate hypothesis
refine hypothesis

The first, *evaluate*, takes some hypothesis (such as a malfunction hypothesis) and assigns a likelihood based on the current case data. A precondition for applying *evaluate* to a hypothesis is that the hypothesis must not have already been evaluated. The second subtask, *refine*, takes a hypothesis as input and produces the refinements for that hypothesis. *Refine* has two preconditions: the hypothesis must be likely and must not have already been refined.

We must also specify the operator proposal knowledge, that is knowledge that determines when an operator should be considered for application to a state. For the hierarchical classification method we are describing, the operators *evaluate* and *refine* should be proposed whenever their preconditions are met.

The initial and goal states and the subtasks define a search space or problem space. Figure 1 illustrates the search space that results when the hierarchical classification method described above is applied to a liver diagnosis problem. The search space is the set of states reachable from the initial state by applying the operators for the method. The figure shows part of the search space of the task, beginning with the initial state, *S1*, containing manifestations indicative of a viral infection (labeled *Data*) and the high level hypothesis *liver disease*. The only operator applicable to this state is *evaluate liver disease*. Application of this operation results in a new state, *S2*, in which *liver disease* is rated likely (in the figure this is noted by setting the hypothesis in bold face). Only one operator is applicable to *S2*, *refine liver disease*, resulting in *S3* which contains the refinements of *liver disease: cancer* and *infection*. At *S3* two operators are applicable: *evaluate cancer* and *evaluate infection*; hence the tree branches to show both possibilities: *evaluate cancer* results in *S4'* in which *cancer* is determined to be unlikely and *evaluate infection* results in *S4''* in which *infection* is rated likely.

In the problem-space framework, problems are solved by searching through a problem space for a path from the initial state to the goal state. Problem-space search is done by enumerating operators applicable to the current state (which at the start of problem solving is the initial state of the task instance), selecting from these a single operator and then applying that operator to the current state. The resulting state then becomes the new current state and the whole process of operation selection and application is repeated until the goal state is reached.

Search-control knowledge guides the search through the problem space. For example, in hierarchical classification the agent might apply a heuristic that it is better to evaluate hypotheses with higher likelihoods than those with low likelihoods, or it might decide that the decision about which *evaluate* operator to apply is not important, hence either operator can be selected. In the task structure, to insure that the method is as general as possible, we specify the minimum amount of domain independent search-control knowledge needed for each method. In the *Establish-Refine* method described above, no search-control knowledge is specified because any such knowledge would unduly constrain the method. For example, if either of the heuristics mentioned above were included in the search-control knowledge for the classification method, it would limit the application of the method to those domains and task instances in which the heuristics apply.

The independent specification of search-control knowledge and subtasks lead to two of the primary advantages of the problem-space approach to specifying methods:

1. We are not forced to specify a particular operator sequence. We can specify the search-control knowledge that is general to all task instances for a method and defer other decisions about operator sequencing to system designers or run-time computation. By doing this, we insure that the method can be applied to as wide a range of task instances as possible. In contrast, early GT work often overconstrained the sequencing of operators, limiting the use of each method to a narrow range of problems.

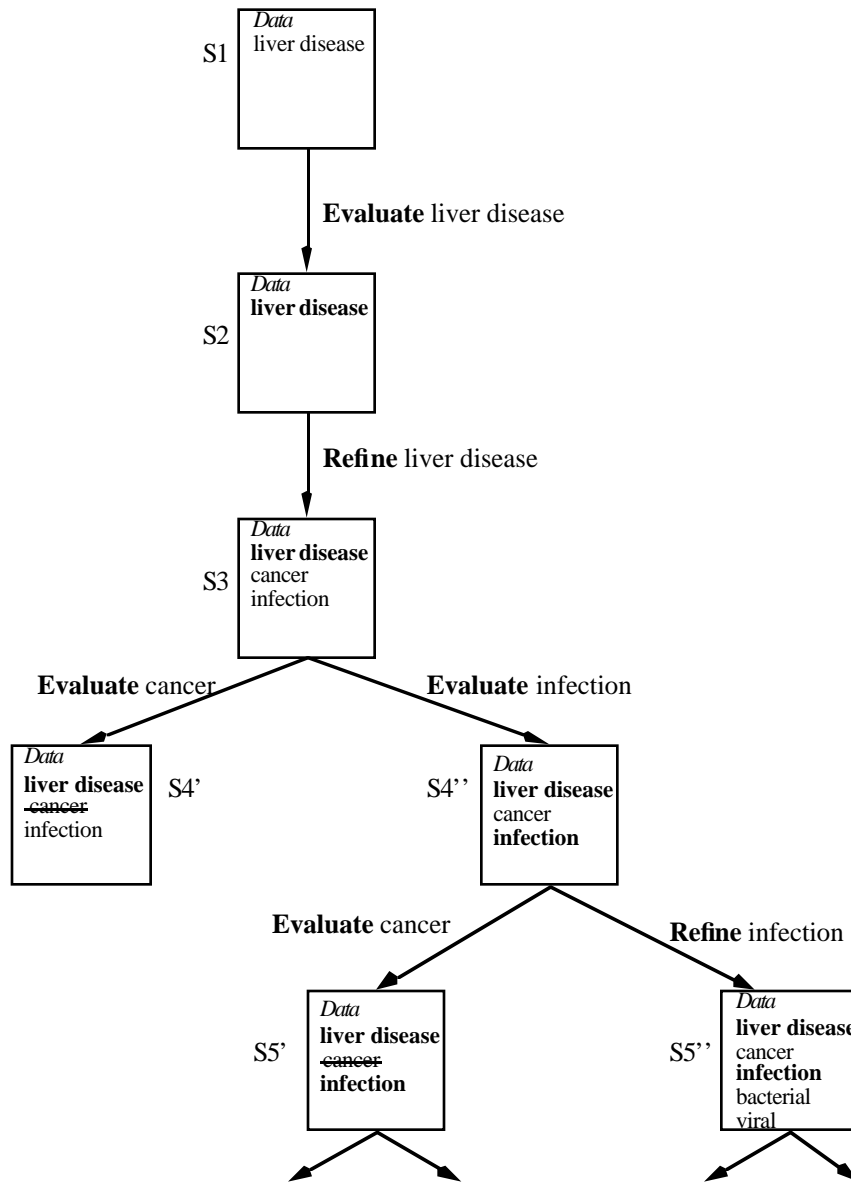


Fig. 1. Part of the search space for the Establish-Refine method as applied to liver disease. The data for this example is indicative of a viral infection. Hypotheses in plain text are unevaluated; those in bold are likely; and those shown with a line through them are unlikely.

2. Search-control knowledge ensures a dynamic or situated system. Each bit of search control knowledge is sensitive to the operators and the current state, hence the precise sequence of operators is determined dynamically at run-time.

3.4 The Task Structure

With the clarification of the terminology of tasks, methods, operators and subtasks behind us, we are now ready to formulate our notion of a Task Structure. The Task Structure is the tree of tasks, methods and subtasks applied recursively until tasks are reached that are in some sense performed directly using available knowledge. Figure 2 graphically represents part of the task structure for diagnosis. A task (as we defined earlier) is a problem type, such as diagnosis. Tasks are represented graphically using circles. A method is a way of accomplishing a task. These are represented graphically using rectangles. In the figure, *Bayesian Explanation*, *Abductive Assembly* and *Cover-and-Differentiate* are identified as methods for doing diagnosis. All of these methods can be classified as abductive methods, hence they appear as a subtype of *Abduction Methods*. In general, a task can be accomplished using any one of several alternative methods, so in the task structure we explicitly identify alternative methods for each task. A method can set up subtasks, which themselves can be accomplished by various methods. For example in the *Diagnosis* task structure *Abductive Assembly* has been decomposed into two subtasks: *Generate Plausible Hypotheses* and *Select Hypotheses*.

In addition to knowledge of the tasks, methods and subtasks, four other kinds of knowledge play important roles in the task structure: 1) subtask preconditions; 2) subtask proposal knowledge, i.e., knowledge about when to attempt a subtask (which can differ from the preconditions); 3) search-control knowledge for sequencing subtasks; and 4) method-selection knowledge for selecting between multiple methods for a task. Method-selection knowledge is associated with a task or task/method combination.

3.5 Examples of Task Structures for Design and Diagnosis

The following descriptions of design and diagnosis illustrate the main points about specifying task structures.

Design. Part of the task structure for design is shown in Figure 3 (this is abstracted from the task structure description in [16]). The top task for the design task structure is, of course, *Design*. The design task can be solved using a family of methods called *Propose-Critique-Modify* (PCM). These methods have the subtasks of proposing partial or complete design solutions, critiquing the proposals by identifying causes of failure if any and modifying proposals to satisfy design goals. Hence the three subtasks shown for PCM: *Propose*, *Critique* and *Modify*. These subtasks can be combined in fairly complex ways, but the following is one straight-forward way in which a PCM method can organize and combine the subtasks.

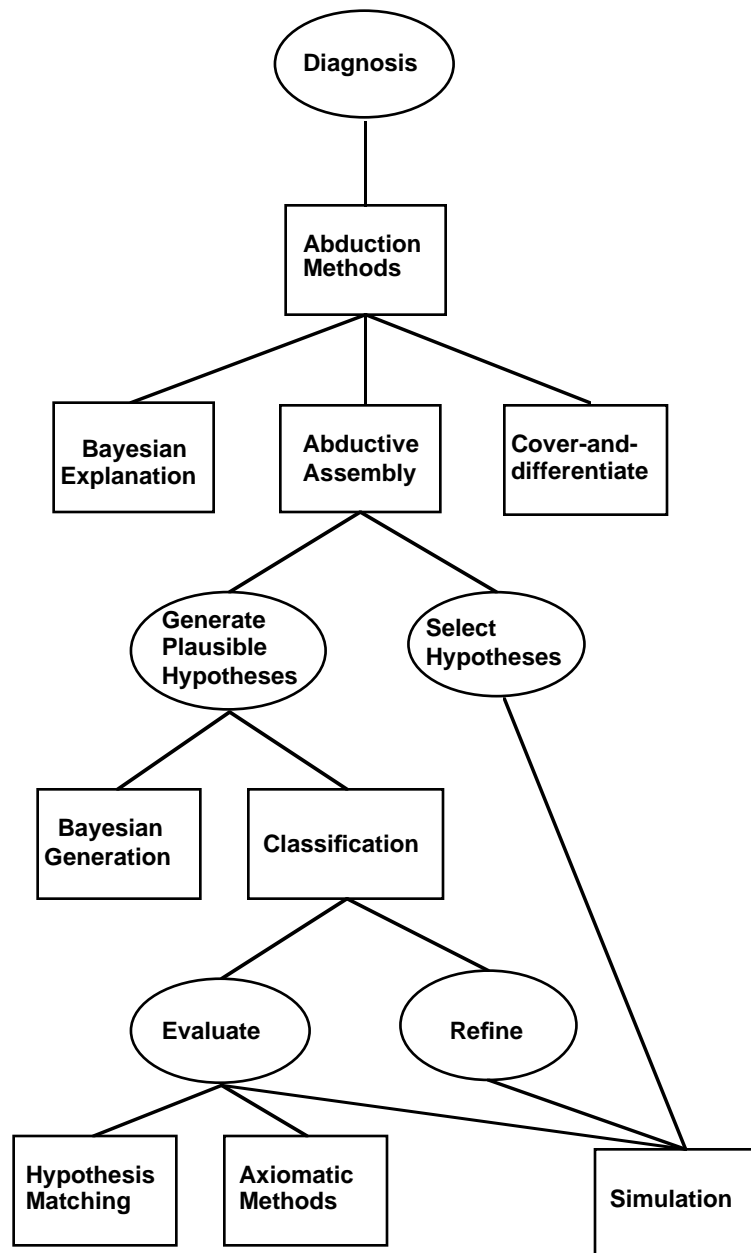


Fig. 2. Part of the task structure for diagnosis. Circles represent tasks; rectangles represent methods. See Section 3.5 for a discussion of the role of simulation.

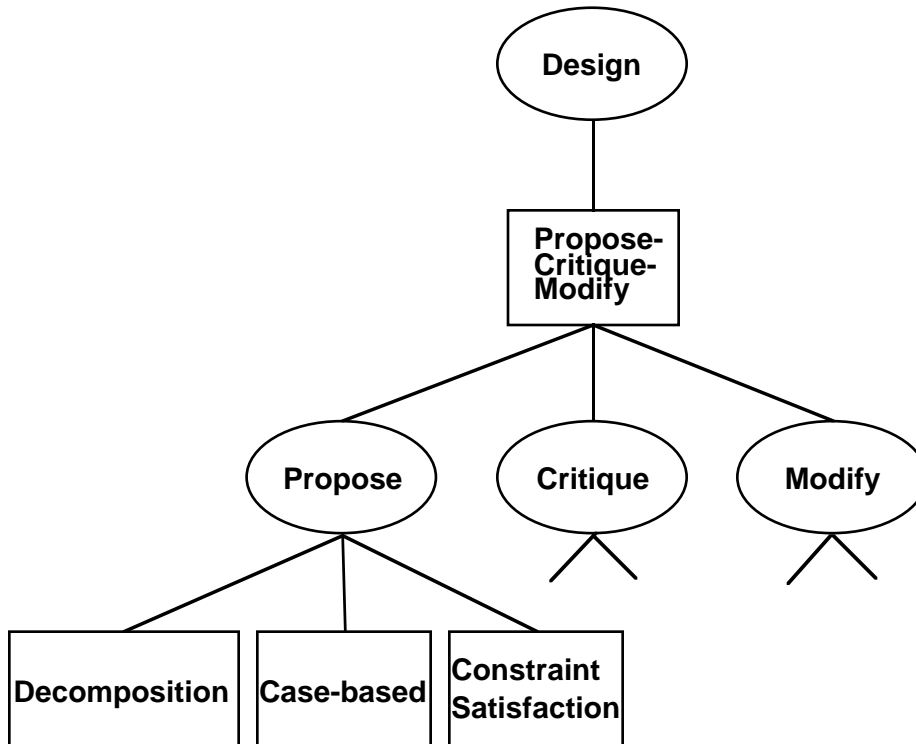


Fig. 3. Part of the task structure for design [16]. Circles represent tasks; rectangles represent methods.

- Step 1.** Given design goal, propose solution. If no proposal, exit with failure.
- Step 2.** Verify proposal. If verified, exit with success.
- Step 3.** If unsuccessful, critique proposal to identify sources of failure. If no useful criticism available, exit with failure.
- Step 4.** Modify proposal; return to 2.

There can be numerous variants on the way the methods in this class work. For example, a solution can be proposed for only a part of the design problem, a part deemed to be crucial. This solution can then be critiqued and modified. This partial solution can generate additional constraints, leading to further design commitments. Thus, subtasks can be scheduled in a fairly complex way, with subgoals from different methods alternating. One could generate all such variations and identify them all as distinct methods, but both the need for descriptive parsimony and the sheer numerousness of the methods argue against doing that.

Each of the PCM subtasks can be achieved using various methods. Three such families of methods are shown for the *Propose* task (see Figure 3): *decomposition*, *case-*

based and constraint satisfaction. In decomposition methods, domain knowledge is used to map subsets of design specifications into a set of smaller design problems. The use of design plans is a special case of the decomposition method. Case-based methods are those retrieving from memory cases with solutions to design problems similar or close to the current problem. Constraint-satisfaction methods use a variety of quantitative and qualitative optimization or constraint-satisfaction techniques.

Diagnosis. Part of the task structure for diagnosis is shown in Figure 2. The diagnosis task can be viewed as an abductive task, the construction of a best explanation (one or more disorders) to explain a set of data (manifestations). The task structure shows three typical subclasses of abductive methods: *Bayesian*, *abductive assembly* [44] and *parsimonious covering* [59]. Bayesian methods require knowledge of prior probabilities of disorders and conditional probabilities between disorders and manifestations. They use this knowledge to estimate posterior probabilities of disorders. Abductive assembly requires knowledge of disorders and the manifestations that they explain. This method works by first generating plausible hypotheses to explain parts of the data and then using these hypotheses to assemble a complete explanation of the data. Parsimonious covering works by stepping through each manifestation, updating the current set of parsimonious explanations as each manifestation is considered. Two subtasks for abductive assembly are shown in the diagram, generate-plausible-hypotheses and select-hypotheses. These tasks can be done using many kinds of methods. Bayesian and classification methods have typically been used to generate plausible hypotheses, so these are shown in the task structure.

Role of Simulation. The task structure for diagnosis also shows that simulation can be used to implement many subtasks. By simulation we mean structure-to-behavior simulation, i.e., determining how some device will behave under changes to its structure by simulating its behavior under those conditions.

Simulation can be used to evaluate a hypothesis because the simulation can reveal whether the hypothesis is possible given the data about the device. Causal refinements of a category can be determined by simulating to determine the possible outcomes of a set of inputs to a device. Simulation plays an important role in many task structures because it is a fairly general method for generating knowledge based on the structure of a device. We did not show the simulation method in the design task structure, but there too it can play an important role, especially for critiquing designs, by generating some of the knowledge needed for the subtask of design verification.

These task structures are based on the methods and subtasks implicit in many expert systems that perform the tasks. Neither of the task structures is meant to be complete; however, both capture a wide range of the methods useful for achieving the respective tasks. As we discover additional methods, these can be added to the structure. Some methods, such as depth-first search, are so general they can be used to solve any problem (see Section 3.7). These methods are not listed in the task structure since they would appear everywhere, cluttering the diagram.

The task structure is meant to be an analytical tool. We do not mean to imply that the implementation of a system must have a one-to-one correspondence to the task structure, but that a system that performs diagnosis or design can be viewed as using some of the methods and subtasks. In particular, the task structure does not fix the order of subtasks or dictate that a single method must be used to achieve each task. It is

also not meant to correspond to a procedure-call hierarchy, although that is one way to directly implement a task structure. The task structure simply provides the vocabulary to use in describing how systems performing the task work. The systems being described might be based on neural networks, production rules, frames, or a task-specific language; this is unimportant for the use and construction of the task structure.

3.6 Direct Versus Derived Knowledge.

The knowledge in a task structure can be available in two forms: it can be directly available or it can be computed by another method. Directly available knowledge is that which is in a form that maps the input of the task to the output. For example, directly available knowledge for *refine* is of the form:

If task is *refine h1* then refinements are *r1, r2, r3...*

For instance:

If task is *refine liver disease* then refinements are *infection* and *cancer*.

No complex computation (i.e., other subtasks) is required to use this knowledge to accomplish the *refine* task; the knowledge is in a form directly applicable to the task. If knowledge is not directly available, it must be derived from existing knowledge or acquired anew from the external environment. In either case, a method must be used to acquire knowledge of the desired form. In the example in Figure 1, *refine* can be accomplished using a method that knows about different refinement dimensions, such as refinements along etiologic and subpart relations. This method can evaluate various dimensions and then select the dimension appropriate for the task instance. For example, using etiology *liver disease* could be refined to *infection* and *cancer*; using anatomic structure *liver disease* could be refined to *central area* and *portal area*. The most appropriate dimension to use depends on the task instance, i.e., the kinds of manifestations available.

Whenever knowledge needed to carry out a method is not directly available, subtasks to acquire the knowledge can be created and set up as new problems. These subtasks are viewed like any other task: they have an initial state and a goal state and can be accomplished by the application of a method consisting of a set of operations. Hence, although a method requires certain kinds of knowledge to be applied to a task, this knowledge does not have to be known before problem solving can begin, but can be dynamically acquired or derived at runtime.

This idea is also closely related to the distinction between “deep” and “shallow” knowledge, sometimes called “deep” and “compiled” knowledge. There is also often another distinction between model-based and rule-based reasoning, where models are supposed to be more general knowledge that describes the principles of the domain, while rules are supposed to refer to relatively *ad hoc* associations between evidence and hypotheses. In [17], we provide an analysis of these terms and develop a notion of “depth” of knowledge that is important for knowledge modeling. We give a brief description of this idea.

Let $K(T,M)$ denote the knowledge needed by method M in performing the task T . If a knowledge system performing T using M has the knowledge $K(T,M)$ directly available in its knowledge base, let us say that the knowledge system has the knowledge in a compiled form. However, suppose some knowledge element k in $K(T,M)$ is missing in the knowledge base, and the task of generating this knowledge is set up as a sub-task. If there exists some other body of knowledge in the knowledge base, say K' , such that by additional problem solving using K' we can generate the knowledge element k , we can say that K' is deep relative to k .

In the *refine* example above we saw that anatomic structure is one of the dimensions along which refinement could be done. In the model-based reasoning approach, structural descriptions of the device under diagnosis are used to generate refinement hypotheses. From the device model, we can generate a list of malfunctions (e.g., one malfunction category can be assigned to the failure of each of the functions of each component; moreover, malfunction categories can correspond to errors in connections between components). The same structural model can be used to generate knowledge needed for the *Evaluate* subtask in Figure 2. The structural model can be simulated for each malfunction, and information about the relation between malfunctions and observations, which is the type of knowledge needed for the methods of the *Evaluate* sub-task, can be generated (See 3.5 for information on simulation). Thus the structural model is a deep model for the methods of classification and hypothesis evaluation that are used in the diagnostic task.

The approach to defining the notion of depth of knowledge in the framework of tasks, methods and knowledge generalizes previous work in the field that has equated structural models with deep models. Under our definition depth is a relative notion, i.e., it is relative to a method for a task, and there is no notion of characterizing knowledge as deep or shallow in some absolute way.

3.7 What Kinds of Methods Belong in the Task Structure?

In the task-structure framework, methods are attached to tasks, rather than identified as independent objects. A criticism is that some methods are applicable to all or many tasks and as such they need not be indexed by tasks at all. The problem here is a shifting sense of the word “task.” The technical definition of “task” that we are using in the Task-Structure framework is that of class of problem instances. It is certainly true that there are very general methods such as *generate and test* that can be used for, say, diagnosis and design. This doesn't make the notion of a method exist independent of tasks. Tasks as classes of problem instances can be partially ordered in a generalization hierarchy. For example, the task of “problem solving” includes all instances of all problems, while diagnosis and design are subtypes of problem solving. Thus both *generate and test* and *classify* are in fact methods associated with tasks. *Generate and test* is associated with the most general class of tasks, while *classify* is associated with a subclass. Of course, all subtypes can inherit the methods associated with the parent. Thus diagnosis can be solved by *generate and test* as well. (The task structure doesn't usually indicate such general methods due to reasons of descriptive economy.) The particular instantiation of *generate and test* for diagnosis would be somewhat more specific, since only diagnostic hypotheses will be generated, and only task-specific tests will be used.

There is another sense in which it is often said that methods are independent of tasks. Thus one might say that *qualitative simulation* as a method is task-independent, since it can occur as part of say both diagnosis and design. But this is again based on a highly restricted use of the word “task.” We saw earlier that *qualitative simulation* is a method that can be used for many subtasks in the task structures for both diagnosis and design. That is, ultimately, methods are always in the service of goals. Methods that appear to be independent of goals are in fact either methods for very general problems or for tasks that are very specific but that appear as subtasks for a number of tasks.

3.8 Task Structure and Domain Knowledge

The Task-Structure approach helps in understanding how domain knowledge comes about to be in certain forms, and in modeling this knowledge.

Methods require characteristic types of knowledge. The Task Structure view associates tasks with methods that accomplish them and the knowledge required to use the methods. The multiple levels of the task structure show how knowledge can be decomposed into bodies of knowledge that are associated with specific tasks. The task structure also highlights the generality and specificity of the knowledge needed for a problem-solving method. That is, it allows methods to be compared based on the required knowledge; hence, we can see how some methods require little domain knowledge (such as depth-first search, which only requires knowledge to recognize a goal state) while others require considerable domain knowledge (such as hierarchical classification, which needs a domain-specific hierarchy of categories).

Normative algorithms for complex problems are less useful than they seem. The Task Structure view should be contrasted with what one might call a “uniform normative algorithm” view of how to solve complex problems such as diagnosis or design. For example, there have been proposals for a general algorithm for diagnosis: “diagnosis from first principles” [63] and Bayesian networks [58] are two examples. The general algorithms, while they guarantee an optimal solution within their respective frameworks, are typically intractable. Engineering of systems to solve the tasks is done by one of two approaches. In one, additional knowledge which constrains the choices and produces tractable behavior is brought to play; however, this knowledge is just thought of as domain-specific knowledge. One way of viewing the methods in the task structure is that they identify types of such constraining knowledge for classes of problems. In the other approach, various forms of heuristic approximations of the general algorithm are used, which of course no longer have the normative properties that are associated with the original algorithm. The general algorithms also do not always make contact with the form in which knowledge is actually available in various real world domains. Thus, the Bayesian framework may be fine for a domain where the needed prior and conditional probabilities (or good approximations to them) are available, but in other domains where the domain knowledge takes other forms, there is often a need for translating from these forms to the probabilistic forms in which knowledge is needed.

The Task-Structure approach on the other hand views the solution of complex problems as arising out of the interaction of many local methods for local tasks. In any domain where there is a record of successful human problem solving, the knowledge in the domain helps to decompose the task into manageable subtasks such that each of the problems can be solved to the degree of precision and accuracy needed for

the domain. It then becomes the task of the AI theorist to develop vocabularies of generic tasks, methods and knowledge. Thus the attention is shifted from the search for uniform algorithms to modeling knowledge and methods by which tasks are decomposed and subtasks are accomplished.

The fact that we do not start with a uniform normative algorithm does not mean that we cannot be precise about the behavior of systems built in the Task-Structure framework. Bylander *et al.* [7] and Goel *et al.* [30] are examples of analyses in which the role of specific types of knowledge in producing good computational properties is studied within the general framework of the task structure view. For example, Goel shows why classification is an attractive method, if knowledge in the form of classification hierarchies is available, and Bylander *et al.* show how knowledge about the existence of certain types of causal links (and non-existence of other types) makes the abductive assembly method tractable.

Domain knowledge evolves in order to match the needs for attractive methods. We can also see how such task structures evolve in real world domains. If classification is a generally effective method for the generate-hypotheses subtask of diagnosis, then over time, the problem solving community develops the knowledge needed to apply it. Thus the medical community has devoted hundreds of years to the development of disease taxonomies, which is the form in which the classification method needs knowledge. Knowledge compilation techniques (see Section 3.6) are also a means by which knowledge in a less direct form is converted into knowledge in a form that is more directly usable by a computationally attractive method. In domains and tasks of importance, domain knowledge tends to evolve over time so that methods with good computational properties can be supported.

Task-structures capture regularities in a body of knowledge. Often AI approaches are categorized as either a good way of solving a problem (ideally a way that is justified as rigorous in some way and hence offered as prescriptive) or a model of how a human agent solves it. We have already described the problems in using such prescriptive methods for building knowledge systems. If the human agent whose problem solving is modeled is a certified expert, then perhaps the model may be expected to perform well in the same task domain. Additionally, the issue of knowledge availability does not arise since the model only uses the knowledge acquired from the expert. However, cognitive modeling in this sense restricts knowledge systems to be models of individual human problem solvers.

The Task-Structure analysis represents a third way between prescriptive techniques for solving a problem and cognitive models of individual agents. The methods in the structure need not model the knowledge of any individual agent but could model the structure of the corpus of human knowledge in the domain. The collective knowledge of a community of problem solvers can transcend the errors and limitations of individual agents and be a stable, robust and convergent body of knowledge. We just discussed how domain knowledge evolves to satisfy the needs of computationally attractive methods.

Since much of this community knowledge is meant to be used by individual agents most of the time, this knowledge has certain interesting properties. Individual agents should be able to use, learn and generate it. Certainly parts of it could be in the form of specialized computational models, but in general the knowledge is qualitative and robust (i.e., insensitive to small errors in data or reasoning). Methods of this type have manageable computational complexity. Complex methods are decomposed into subtasks and methods for them.

The means-ends (or goal-subgoal) character implicit in the task structure (i.e., the methods are the means which in turn set up subgoals) and the relatively low complexity of the methods are properties that reflect the fact that the intended users of the community knowledge are human agents. Of course, when we wish to mechanize problem solving, we don't have to be restricted to these methods, but use of these methods helps to ensure that the domain knowledge needed to execute them is likely to be available.

Task structures retain the knowledge-modeling advantages of the earlier task-specific tools. Since methods are characterized by the knowledge they require, domains can be modeled by tools appropriate for the knowledge that is available in the domain. High-level tools based on this concept, such as CSRL [10], DSPL [6], MUM [33], and MOLE [27], can be viewed as method-specific shells. Much has been written about how they facilitate knowledge modeling, knowledge acquisition, explanation and learning.

Task structures suggest how generation of new knowledge can itself be viewed as a reasoning task. During knowledge modeling appropriate questions can identify sources of deep knowledge for various methods in the task structure.

Task structures suggest how methods of different types can be combined. Quantitative and qualitative knowledge, heuristic and algorithmic knowledge can be appropriately combined for the accomplishment of a task. For example, if a subtask can be accomplished using a known technique, for instance by solving a set of differential equations, that method can be used instead of more traditional AI methods. The method that set up this subtask is concerned with the solution, not how it was arrived at, so the original task can be implemented using a different kind of method or even a different computational architecture. The only requirement is that there is an underlying architecture which can set up goal stacks, invoke methods and unwind the goal stack as methods return solutions.

Task structures have a number of architectural implications. Overdetermination and rigidity in methods are avoided by using the task structure because a complete method does not need to be specified (only the subtasks need to be given and not all of these have to be used to accomplish a task). Furthermore, multiple methods can be used to model domains that do not warrant the selection of a single method for accomplishing a task. Overdetermination and rigidity of implementation can be avoided by dynamically determining methods and subtask sequencing at runtime. We will next discuss some architectural alternatives to achieve this kind of dynamic selection of methods.

3.9 Architectures for Supporting Task Structures

So far we have presented the task structure as an analysis technique. We can, however, also ask about the architectural implications of such an analysis. An obvious possibility is to support each of the methods by a method-specific shell. In fact the earlier generation of GT's can be viewed as method-specific shells (except that the method that a GT supported was the only one for the task). If we want the system to have the flexibility to select methods at run-time, we also need, in addition to method-specific architectures, a capability to invoke methods, assess them and select one to achieve a goal. We will now discuss the TIPS architecture of Punch [62] and the Soar/GT work of Johnson and Smith [43], both of which were attempts at LAIR to achieve flexibility in problem solving behavior.

The germs of the task structure view were contained in the work that William Punch began in 1988 towards building diagnostic systems. MDX2 [67] had shown how causal models could be used to generate missing diagnostic knowledge, but which model to use had been hard-wired at design time. Punch wanted the diagnostic system to invoke different types of models selectively as run-time problem solving needs dictated. He developed the TIPS architecture as a solution to this problem. TIPS had the ability to invoke different methods for a problem-solving goal (the methods had to be appropriately indexed as being relevant to a goal), evaluate them, order them, and select the most appropriate one. (See [16] for the criteria for method selection.) The selected method may set up a subgoal for which the architecture would again invoke and select methods. Thus, during diagnosis, the system might invoke a method that used simulation of a physiological model for a local subgoal in one problem instance, while for another problem instance the heuristic match method might be chosen. TIPS used a simple *sponsor-selector* scheme for partial ordering of the appropriateness of the methods. This scheme called for the knowledge engineer to represent in advance the conditions on the state of problem solving under which a method was appropriate. Punch independently invented several features of the Soar architecture that we will discuss shortly. More recently Herman has built DSPL++ [36] as a flexible architecture for design problem solving in a framework that explicitly follows the design task structure as developed in [16]. DSPL++ supports a number of different methods for the Propose subtask of design, and is extensible, i.e., methods can be added. DSPL++'s techniques for invoking, evaluating and selecting methods were similar to those of TIPS. (Dynamic method selection has also been investigated by [2], [70], and [69].)

This brings us to the Soar phase of our work. Around 1988-89, Jack Smith had become excited by Soar as a general architecture [47]. In an earlier era, we would have reacted to it as just another general purpose architecture, but with the recent awareness of the need for an architecture that can support flexibility, we saw that Soar offered us three capabilities:

1. Unlike earlier general purpose architectures, Soar's problem-space construct was consistent with a key insight of the GT view: the close connection between the task, the method and the knowledge needed to support the method. Each problem space was in fact the agent's way of bringing the task and the knowledge together in one organized entity.
2. Universal subgoaling was a way of achieving great run-time flexibility in the invocation and choice of methods. Soar's preference scheme for ordering choices had many similarities to Punch's sponsor-selector mechanism for doing the same, but was more general.
3. Soar's chunking mechanism could explain how higher level GT's such as DSPL's could come about and be put to general use without their being initially available in that chunked form. That is, the TSA's that we have been proposing could be seen as emergent virtual architectures for classes of problems.

But we also saw that Soar had no content theory of tasks or of methods (except for a collection of weak methods). The GT's and later the methods in the task structures that we identified for diagnosis and design are a content theory of those tasks. Thus it appeared that it would be an interesting and important research direction to see how the GT idea could be worked out in the context of Soar as the underlying architecture.

This is what Todd Johnson did for his thesis [40, 42]. There are several ways of implementing GT's (or generic methods in the task structure) in the Soar framework. First, each of the methods can be implemented directly as a problem space. We can go all the way from a fully procedurally specified method (one all of whose control choices are specified ahead of time) to one whose control behavior is all determined at run-time. For example, in classification, depending upon the search-control knowledge that is made available, a hierarchical control structure can emerge, or an exhaustive search of the hypothesis space can take place.

If the methods are complex, that is, if they can be decomposed into many subtasks, then a finer-grained control behavior can be obtained if the sub-methods are treated as the units for invocation and selection. Extremely flexible interlacing of subtasks from different methods can be achieved. For example, during diagnosis a subgoal in the abductive assembly method can be followed by a subtask in the classification method. Because problem spaces permit implementing methods whose control is not specified in advance but can emerge at run time as a function of available knowledge, a full range of control can be implemented. The complex methods that characterized earlier GT implementations can be obtained as special cases in this framework. In particular, the chunking mechanism of Soar can be used to demonstrate the emergence of the somewhat more compiled virtual architectures such as DSPL.

The role of the chunking mechanism in Soar in producing GT-like architectures as a result of experience is relevant for an issue that we noted in our discussion of the early history of GT's. We said that our attitude to lower-level architectures (rules, frames, logic, etc.) was that they were just implementation alternatives without any intrinsic theoretical interest. Functionally, we argued, once we extracted the knowledge and strategies by an analysis at the task-level, the problem solver can be implemented in any of a numerous array of lower-level architectures. However, the fact that GT-level architectures can emerge from the appropriate lower-level architecture as a result of suitable learning mechanisms means that not all lower-level architectures are equivalent.

If the methods themselves can be written in the same architecture which is used for dynamically selecting methods, and if the learning mechanisms of this architecture can chunk or compile method-specific virtual architectures as a result of problem solving experience over a number of problems of a certain type, then significant unification would have been achieved. The problem-space perspective and the associated computational architecture provides a framework in which this unification is possible. The computational architecture comes with a learning mechanism of the appropriate type for chunking the GT's. This suggests that not all implementational alternatives are equally desirable. The alternatives can be evaluated with respect to how they support problem solving, learning, and dynamic invocation, all in a unified framework.

In the Soar implementation of GT's, we get the best of both the worlds: the task-level leverage of the GT perspective and the flexibility and opportunism of the Soar architecture. But the range of architectures that are useful in practice as technology for knowledge systems still occupies a large space, and there is room for a number of architectures with different degrees of flexibility. The Soar implementation of GT's occupies one niche, where the subgoals in different methods can be combined very flexibly at run time. TIPS and DSPL++ occupy another niche, where methods, with all their subgoals, can be invoked and combined at run time as units. The earlier CSRL/DSPL systems occupy yet a third niche where the designer has enough information to hard-wire what method will be used for which subtask at design time. Unifying

them all is the Task-Structure view that identifies the goals, methods for them and the knowledge needed to implement the methods.

3.10 RedSoar: An Abductive System in Soar

To emphasize the importance and role of the Task Structure for describing systems, as well as to give a sense of how the Soar architecture can support the realization of task structures, let us consider three descriptions of RedSoar, an abductive system that interprets immunohematologic tests in order to identify antibodies present in a patient's blood [39]. In describing a complex knowledge system such as RedSoar, we can use three levels: 1) the Task Structure; 2) a computational level (such as problem spaces); and 3) a symbol level. Figure 4 shows each of these levels for RedSoar. RedSoar uses abductive assembly of antibody hypotheses to construct a best explanation of the test data. In the task, the test data are the manifestations; antibodies are the “disorders” or explanations. RedSoar is directly implemented in Soar's production-rule language and can be described by listing all the rules in the knowledge base, such as those in Figure 4c. About 1000 of these rules constitutes the symbol-level view of RedSoar. However, this description fails to capture the task-level control and knowledge in the system. To do this, RedSoar can be described at a computational level by listing the problem spaces defined by the Soar production rules, as in Figure 4b. That is, we can abstract away from the symbol level production rules to focus on the problem spaces, their initial and desired states and their operators. This level of description is much closer to the task level but would still contain too many details present as artifacts of the implementation (e.g., extra operators that must be used for low-level manipulation of representations). At the Task-Structure level (see Figure 4a), we can simply describe the system as using abductive assembly and then point out how it generates and selects hypotheses: i.e., the methods and knowledge that it uses. RedSoar uses conditional and *a priori* probabilities to generate plausible hypotheses and a scoring function based on explanatory coverage and plausibility ratings to select a hypothesis. As shown in the figure RedSoar also uses two additional subtasks, *Rule-out* and *Confirm* hypotheses. These are domain-specific subtasks. The first allows the system to quickly rule-out clearly absent antibodies. The second lets the system focus on antibodies that are likely present. By describing RedSoar at this level, a comparison can be made between it and other abductive assembly systems by comparing the methods and knowledge used to generate and select hypotheses.

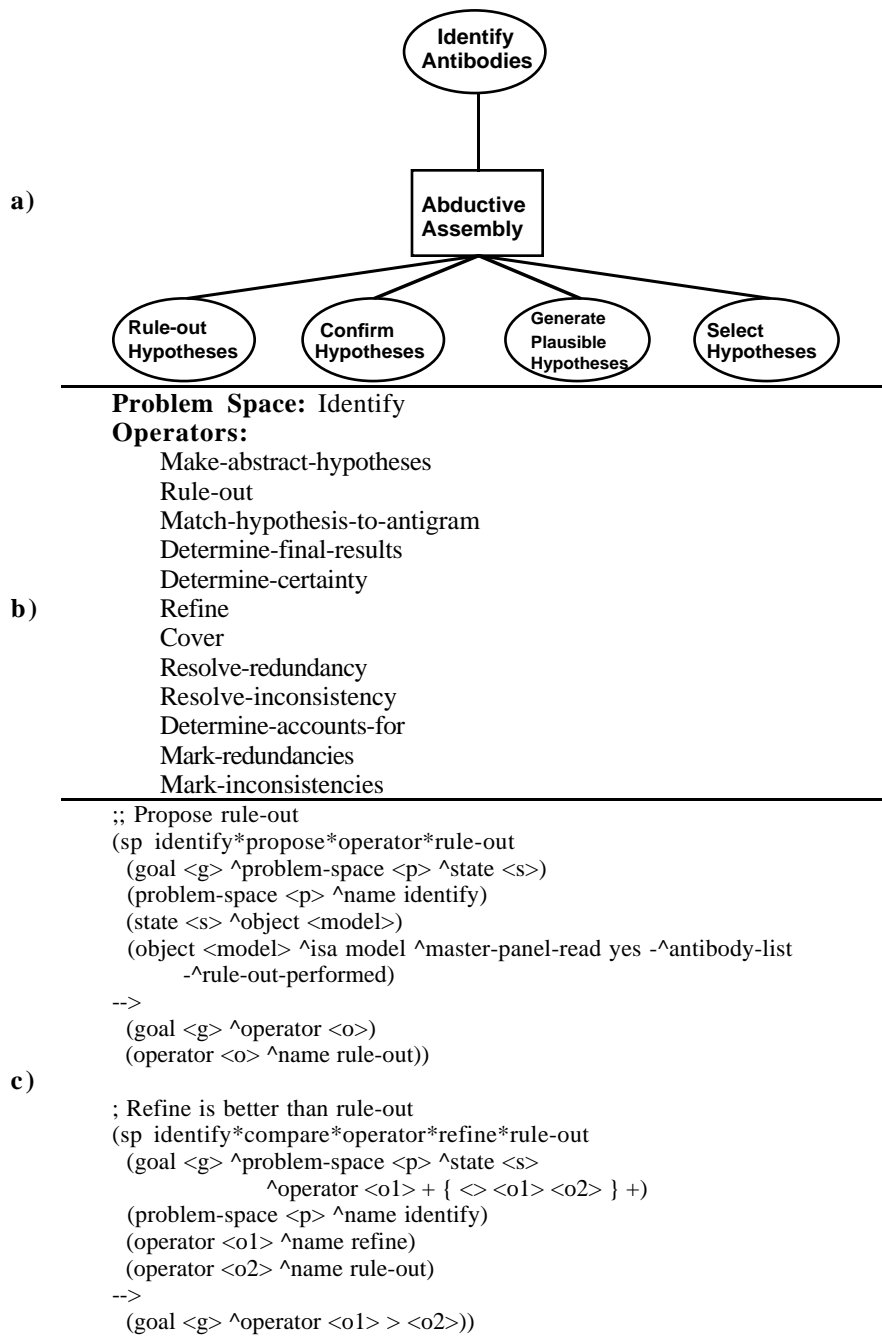


Fig. 4. RedSoar described at three levels: a) the task structure; b) the problem-space level (a computational level; and c) production rules (the symbol level).

4 Comparison to Other Work

4.1 Clancey's Model-Construction Perspective

Recently, Clancey has developed the model-construction perspective wherein systems are viewed as constructing and manipulating models of situations [25]. The knowledge in these systems is modeled using a general graph/set/operator language. By developing primitive model-construction operators, it is possible to compare systems that have very different implementations at an implementation-independent level. The model-construction perspective is related to the KADS approach in that both attempt to offer a uniform language for modeling knowledge.

In Clancey's view, knowledge systems use domain models and inference rules (contained in the knowledge base) to build a situation-specific model (SSM). Specifically, he suggests that SSM's be viewed as graphs whose nodes and links are constructed by the inference steps during problem solving. For example, consider the problem-solving graph produced by a diagnostic system whose problem solving consists of associating subsets of the symptoms with diagnostic hypotheses that explain it, and establishing the causal pathways by which the diagnoses cause the symptoms. Some of the nodes in this graph would be symptoms, some nodes would be pathophysiological states hypothesized by the problem-solving system during problem solving, and finally some nodes would correspond to diagnostic hypotheses. Many of the links would be causal links, (e.g., such and such a pathophysiological link caused such and such a symptom). Links such as "type-of" can also relate some nodes (e.g., after establishing liver disease, the problem solver might establish cirrhosis, which is a type of liver disease).

At the (successful) completion of problem solving, a subgraph of the problem-solving graph would correspond to a model of the specific case: which symptoms were caused by which pathophysiological states which in turn were caused by which disease processes which were types of which diseases, and so on. When the diagnostic job is done, not only do we have the name of a disease, but we also have an explicit record of the relation between the data and the conclusions. This subgraph is what Clancey calls the situation-specific model.

From the modeling perspective, the language in which SSM's are constructed is closely related to tasks. That is, the types of nodes and links that are constructed for a diagnostic problem are likely to be different than those for a planning problem or for a design problem. In a diagnostic task, the SSM's are couched in the language of "causes," "explained-by" and so on, while in a design task, the basic vocabulary would include terms such as "sub-function-of," "achieves-behavior" and so on. Secondly, while the terms are related to the task, the tasks would share parts of the vocabularies. To the extent that both design and diagnosis deal with devices, causal processes in them, their functions and malfunctions, many terms would occur in the SSM's of both diagnostic and design tasks. In fact, the task/method/subtask point of view that we advocate makes this clear on the following way. The task structures for design and diagnosis both have simulation as a possible subtask; and a common method for simulation is tracing of causal paths from a causal network. Thus we would expect that the problem solving activities which construct SSM's for diagnostic and design problems would, when they are engaging in the simulation subtask, be using identical terms for that portion of the SSM.

The SSM viewpoint can be sharpened by the analysis provided by the Task Structure framework. We can decompose the SSM built by a problem solver into a hierarchy of SSM's mirroring the task/method/subtask hierarchy. Thus for a diagnostic problem, the task at the highest level can be described as the construction of a problem-specific SSM of a certain type: specifically, a model in which the abnormal observations of a malfunctioning systems are linked to diagnostic hypotheses by "caused-by" or "explained-by" links, with perhaps additional causal links between hypotheses themselves. At this level we can take Clancey's proposal as a notational variant of the goal state specification in the problem space point of view. As the problem solver uses the knowledge to consider and select methods and starts executing the selected one, SSM's specific to the method chosen will be constructed. For example, if the abductive assembly method is chosen, the SSM will consist of initial hypotheses, their plausibilities, assessment of various hypothesis combinations and so on. The subtasks set up by this method will have their own characteristic SSM's, and this process will recurse, similar to the task/method/subtask recursion. The hierarchical structure of the SSM's that emerges from the Task-Structure viewpoint makes SSM's easier to comprehend and use during the problem-solving process itself. In fact, we can regard the SSM's as a well-organized working memory, which retains the structure of invocation of methods and subtasks. In this sense the hierarchical SSM's induced by the Task-Structure methodology preserve the character of an informal "proof" that Clancey attributes to SSM's, since they reflect how the agent decomposed a task and how the methods accomplish the task instance.

Let us consider the example of a problem solver engaging in the task of behavior prediction using the method of causal process simulation. There is a mapping between the following three ways of talking about the situation:

1. In the domain causal process C , the state $S1$ is causing state $S2$ because of some action $A1$.
2. The problem solver, armed with this knowledge, can transit from a problem state $P(S1)$ to $P(S2)$ by applying the operator "Action $A1$." Here $P(S1)$ stands for the problem state in which the agent's knowledge includes the information that the process C is in state $S1$.
3. The SSM has nodes $S1$ and $S2$, linked by the link "caused-by Action $A1$."

However, not all of the operators in 2 have to be "domain operators" as in 1. Some of them could be "mental" operators, such as abstraction, generalization, specialization, aggregation, and so on. In this sense, SSM's that problem solvers construct reflect both domain terms and additional abstraction terms. What is important for our perspective is that there is a close connection between the tasks, methods and the terms in the corresponding SSM's because these terms are reflected in the knowledge that is needed to implement the methods.

4.2 The KADS Approach to Knowledge Modeling

KADS [72] is a Europe-based research program for knowledge modeling which is becoming quite influential. KADS-2, the most recent version of the approach, identifies four layers:

1. *Domain layer.* In this layer the domain objects and their relations are modeled. For example, in diagnosis, presumably the device components and their connectivity relations would be included in this model. It is often stated that the domain layer is task-independent, but it is hard to imagine that one would know what entities and what relationships should be included in the domain layer without some idea of the range of tasks that we want the knowledge system to perform. Perhaps what is really meant is that the terms in the domain layer are not unique to specific high-level tasks.
2. *Inference layer.* Here the terminological differences begin to become a problem. This layer has a description of the roles played by the objects in the domain layer in the problem-solving process. For example, if “Drug d” is an object in the domain layer, and if “prescribe drug <d>” is a possible action that the problem solver can consider, this layer will make this connection explicit. They have tried to identify generic classes of such problem-solving roles such as “Available-Observation” and “Available-action” (“prescribe drug <d>” is an example of the latter), and a generic taxonomy of what might be called “elementary” goals in our terminology, e.g., “select,” “classify,” “compare,” etc. (They call these a taxonomy of “knowledge sources,” but that term has a different meaning for most researchers in this area. We think it is best to view these as types of primitive goals or primitive operators. See our earlier discussion in Section 3.2 on the duality of a goal or subtask also being an operator.) While the set of terms that they have developed is a useful one, there is no theoretical study of the orthogonality of the terms or their completeness.
3. *Task layer.* To the first degree of approximation, the task layer is really what we have called a method, namely, a series of operators that together help achieve the goal. They additionally specify that each of the operators should be one of the types that we identified as elementary goals in 2 above. There are two reasons why we would be concerned about such a requirement: One is that we would like the operators to be abstract enough that additional problem solving might be used by selecting methods. There is no reason to impose the requirement that the method's sequence of operators at the elementary level be fixed in advance. Second, given our concerns about whether the terms that they propose are orthogonal, primitive or complete, we think the development of such terms should be an on-going empirical enterprise. This is where the task-specificness of our enterprise becomes important, since we think the specification of the goal class gives us leverage in analyzing the methods and developing primitives for them. Perhaps eventually we might converge on a set of terms, but we need to have a clear idea of the role played by the terms in the performance of classes of tasks.
4. *Strategy layer.* This layer exists as a partial answer to one of the concerns that we raised in our discussion of 3 above, namely the lack of flexibility implicit in the fixed set of primitive operators. In the strategy layer, the aim is to represent meta-plans so that different combinations of operators may be generated.

From the knowledge modeling perspective, the only layer where they have made specific commitments for a vocabulary is, as we discussed above, in their proposal for a typology of primitive goal types. As we said, our preference is for these terms to emerge empirically from studies of classes of tasks.

4.3 The Componential Framework

In Steels' componential framework [66] systems are described from three perspectives: 1) the model perspective; 2) the task perspective; and 3) the method perspective. This framework is in many ways similar to the task structure framework that we have described. The model perspective corresponds to the knowledge states of problem spaces in our framework. In both frameworks tasks are decomposed via methods into subtasks to form a task structure. Likewise, both frameworks recognize that the level of the task decomposition (that is, when a particular task should be decomposed into subtasks) varies depending on the purpose of the analysis. There are, however, several differences between the two frameworks. First, Steels' task structure only includes the tasks and subtasks; ours mixes tasks, subtasks and methods. Thus our task structure makes explicit the use of methods including multiple methods for a task. Second, the componential framework makes no commitment to a technique for describing methods, whereas we have selected the problem-space paradigm because of its generality and flexibility.

4.4 Role-limiting Methods²

Role-limiting methods [49] identify particular paths through the task structure. A role-limiting method is described to the lowest level, i.e., all tasks, methods and subtasks are specified until the lowest subtasks can be directly implemented. The specification also includes the order in which the subtasks are accomplished. This differs from other approaches in which a system can be described at high levels of abstraction with lower level details remaining to be specified either later in the design process or dynamically at runtime. The implication of this is that a small variation of a particular role-limiting method must be viewed as a different role-limiting method.

For example, *Cover-and-differentiate* [49] is a role-limiting method for a form of abduction. McDermott defines the method as follows [49]:

1. Determine the events that potentially explain (that is, cover) the symptoms.
2. If there is more than one candidate explanation for any event, identify information that will differentiate the candidates by
 - ruling out one or more of the explanatory connections,
 - ruling out one or more of the candidate explanatory events,
 - providing sufficient support for one of the candidate explanatory events,
 - providing a reason for preferring some of the explanatory connections over others.
3. Get this information (in any order) and apply it (in any order).
4. If step 3 uncovers new symptoms, go to step 1.

Two classes of task knowledge are needed to use the *Cover-and-differentiate* method:

1. Knowledge mapping manifestations to disorders (for Step 1 of the method, generating explanations for a finding).

²This is a modified version of the analysis appearing in [40].

2. Knowledge mapping an explanation to information that can help confirm or disconfirm the explanation (for Step 2 of the method, differentiating between competing explanations).

These two classes indicate the role that domain-dependent knowledge plays in the method.

On one hand, a role-limiting method is a domain-independent abstract method. Thus *Cover-and-differentiate*, as a method, is specified independent of whether it is going to be used for liver diagnosis or automobile diagnosis. On the other hand, the search-control knowledge that describes the behavior of the method, and hence the method's control behavior, is completely specified as part of the method specification. The knowledge engineer who is using the method for his domain has no freedom to change the control behavior to suit his domain. The philosophy is that precisely this form of pre-specification of the control strategy is needed to be able to guide the knowledge engineer on the kinds of knowledge needed to use the method. Of course, if the knowledge engineer can't even specify parts of the control strategy at system-building time, certainly, the approach does not enable the control strategy to emerge at run-time as a result of interactions between any partially specified search-control knowledge and the particulars of the problem instance. Note that the Task-Structure approach allows a full range of options in the specification of control behavior: all control knowledge does not have to be prespecified, and if some aspects of control are specific to some domains, that is fine as well. Nevertheless, the Task-Structure approach provides the same advantage as the role-limited method approach with respect to identify the role knowledge plays in a method. We will demonstrate this by showing that:

1. The role of domain-dependent knowledge is still limited even though search-control knowledge is not specified as part of a method, and
2. Domain-dependent search-control knowledge can also be role-limited.

To illustrate each of these points consider the method described for abductive assembly. This method supplies very little search-control knowledge, but since the input and output of each subtask in the method are known, the knowledge required to use the method can be specified. For example, to implement *generate-plausible-hypotheses* the system must have domain-dependent knowledge mapping a manifestation to a disorder. One direct encoding of this knowledge is a rule that maps from a particular manifestation to hypotheses that explain it. Even if this knowledge is indirectly represented using a method that implements *generate-plausible-hypotheses*, the role the knowledge plays in the abductive method remains the same. Furthermore, by specifying the task structure for the method that implements *generate-plausible-hypotheses* the role of each piece of knowledge used to generate explanations can be understood. Thus, if the subtask's input and output are known, the role and basic form of the knowledge needed to implement the subtask can be identified.

As for the second point, domain-dependent search-control knowledge is limited to the role of differentiating between competing subtasks. The form of the knowledge is a mapping from one or more subtasks to preference information for those subtasks. This knowledge can be divided into three classes:

1. Knowledge that prefers or rejects a single subtask without respect to other applicable subtasks.
2. Knowledge that prefers one or more subtasks over one or more other subtasks.
3. Knowledge that indicates that two or more subtasks are equivalent.

This might seem an overgeneral specification of the knowledge; however, it is no more general than the role of knowledge for differentiating between explanations in *Cover-and-differentiate*. Thus search-control knowledge can be given the same role-limiting status as the knowledge for differentiating explanations.

5 Shifts in Perspective and New Research Directions

The Task-Structure perspective retains many of the advantages of the GT view, but in crucial places, it reflects changes in philosophy. In this section we discuss some of these shifts in perspective.

The major shift in the point of view is about the status of the TSA's associated with the GT's. The notion of a fixed method for a task is abandoned. Furthermore, the fact that there is no finite set of distinct methods for a task implies that there is no finite set of hard-edged conceptual building blocks. But GT-based TSA's still represent a good idea technologically for a range of situations served well by the method incorporated in the TSA's.

In an earlier section, we enumerated a number of questions about GT's that motivated our reexamination: How many GT's are there? What kinds of tasks count as GT's? What are the criteria? In the current perspective, the answer is that there is no finite number of tasks or methods. They exist in some partial ordering of generality and domain-specificity, and any collection of problem instances that have some problem features in common and that is for some reason worth considering as a group is a generic task. Thus, abduction, diagnosis and medical and engineering diagnosis are all Generic Tasks. Abduction is more general than diagnosis (i.e., covers a much larger number of problem instances), and diagnosis is more general than either medical or engineering diagnosis.

The Task-Structure approach retains many of the traditional advantages of the GT perspective. The task-method-subtask characterization still conforms to the injunction not to separate knowledge from its use. The method organizes how knowledge is to be used to achieve the task. To the extent the methods are generic, useful characterizations can be made of the knowledge and strategic requirement of the methods, just as in the case of the traditional GT's. Leverage for knowledge acquisition and explanation that GT's provided is retained. In fact, because the task structure separates the task from the method, how the method helps achieve the task can be explicitly included in the explanation capability, as is done in Tanner [68]. The generic methods that are identified will play the role that generic mechanisms play in mechanical engineering: organized collections of inferences for generic types of goals. Just like generic mechanisms are not building blocks in a basic theoretical sense (that is, there is no way to argue that in principle all mechanical devices can be decomposed into a set of generic mechanisms), they still play extremely useful roles as technological building blocks. Generic methods—and method specific architectures (MSA's) indexed by tasks that they are good for—will continue to play this role in knowledge-based systems as well.

The analogy of the GT primitives to Schank's conceptual dependency primitives [64] is interesting. Schank proposed about 15 primitives in terms of which all the action verbs in natural languages can be expressed. Wilks [73] also proposed a similar idea, but the number of primitives in his scheme was in the order of hundreds. In practice, however, few natural language systems use the particular set of primitives from either system as canonical objects. There is quite a range of variation in the primitives that do get used. But what has survived is the important idea that it is useful to group verbs in such a way that inferences common to a set of verbs be shared by abstract verbs that stand for their common action quality. Similarly, in the GT view, in spite of the twists and turns about what the primitives are and at what level of genericness should they be represented, what is of long-term importance is the idea that tasks and methods provide the organizing principle for knowledge-based problem solving.

Some might argue that in moving away from a set of primitive "generic" tasks and associated preferred methods we have weakened the enterprise, i.e., that the Task-Structure approach provides fewer guidelines to researchers and system designers. To the contrary the Task-Structure approach builds upon and strengthens the GT enterprise by clearly separating analysis from implementation and by providing a framework for encompassing a multitude of task-method combinations. The task structure concept just provides the organizing principles for analyzing and describing systems, i.e., that there are tasks, methods and subtasks. In this role it is purposefully designed to be extremely general, even content-free. The content is provided by identifying and fleshing out useful task structures for important tasks, such as those we described earlier for design and diagnosis. It is these "instantiated" task structures that provide strong guidance to researchers, system designers and system users.

5.1 Research Directions

Many researchers have been associated with the GT view over time. The research directions that are currently being pursued by them cover a wide spectrum. The research issues can be broadly categorized as architectural and content issues.

Architectural Issues. The dimension in which architectural approaches differ is that of generality and flexibility. As a practical matter, there is a need for different combinations of generality and task-specificness for system-building technologies. There are many technological niches and different researchers are pursuing different ones. At one end of the spectrum, Sticklen continues to experiment with TSA's at the level of granularity of the original GT's. Punch continues to experiment with architectures that treat methods as the units to invoke. Herman [36] has similarly proposed an architecture for design which can invoke, assess and select methods for various subtasks of the design task as described in [16]. At the other end of the spectrum, Johnson and Smith are continuing their investigation into using Soar and problem spaces as the architectural medium to realize task-specific problem solvers. In this approach, TSA's (or MSA's) are constructed in the uniform framework of problem spaces. Whenever a subtask needs a new method that is not supported by an MSA, the problem-space framework is available to code up the new method.

Knowledge and Method Sharing. There are several aspects to the research on the content issues related to task structures. In one, a number of researchers are using the Task-Structure viewpoint to abstractly analyze complex tasks. We have already discussed task analyses of design and diagnosis. Goel and Chandrasekaran [31] provide a task structure for case-based design. Narayanan and Chandrasekaran [55] provide such an analysis for visual reasoning in the task of prediction of behavior of physical objects.

This type of analysis directly leads to a potentially revolutionary technological possibility: method- and knowledge-sharing. Each TSA embodies a class of strategies to solve a type of problem in addition to providing primitives in which to encode needed knowledge. In this sense each TSA is a means to share the corresponding problem-solving method. The task structure makes explicit that it is the methods that are being abstracted and made available for sharing. As research on various diagnosis and design problems is pursued around the world by numerous researchers, task structures which abstract methods and subtasks from the individual efforts and solutions can be constructed, knowledge and strategies for them identified, knowledge representation and acquisition tools for them constructed, and their computational properties analyzed. Task structures and methods in them may begin to play the role in knowledge engineering that identification and analysis of generic mechanisms played and continues to play in other engineering disciplines.

Explanation and Learning. There are a number of issues related to learning that intersect with the task structure perspective. Earlier we said that it was conceptually and practically important to have architectures which unified problem solving and learning, and pointed out that earlier GT-like architectures can be viewed as learned virtual architectures for classes of problems. Goel's work on Router [29], a system that finds routes between locations, is an instance of research in this perspective. It starts with a method that is model-based, but gradually its reasoning shifts from model-based to case-based.

There is another aspect to learning that relates, not to the learning mechanisms that architectures have, but to the content issues in learning. In [15], Chandrasekaran outlined a research agenda for explanation-based learning (EBL) in the Task-Structure framework. EBL, broadly construed, is a method of learning by constructing an explanation of why some solution was correct or incorrect, and using the explanation to refine the representation of the concept that is being learned. Weintraub [71] uses this idea to build a corrective learning component to his gait diagnosis system. When its answer to a problem is incorrect, the system attempts to identify which part of the knowledge it used and which task in the task structure may be at fault and also attempts to change it. The approach uses the fact that the task-specific view gives advantages by providing appropriate vocabularies in which explanation can be couched (Chandrasekaran, 1989). We identify three types of explanation relating to knowledge-based systems. These are: (1) trace of run-time, data-dependent problem-solving behavior, i.e., explaining the data in the current situation was used to arrive at a decision; (2) relating specific decisions to the control strategy used by the system; and (3) justifying particular pieces of knowledge by relating them to more general domain knowledge. When an error is made, Type 1 and Type 2 explanations can be constructed for the incorrect answer. These explanations together can be used to identify possible knowledge elements that could have been responsible for the error, and methods in the task structure that the knowledge elements are associated with. In this way the explanation capa-

bility associated with the task-specific view helps solve some aspects of the *credit assignment problem* for learning. Combining this ability to narrow down the error candidates with simulation of the underlying qualitative model of the gait, Weintraub was able to both locate the source of the error and correct it. The potential of the task structure for helping solve the credit-assignment is significant.

Content Theory of Tasks, Inferences and Methods. Another important content issue is somewhat more foundational for the entire family of approaches at the task level. There is a central theoretical issue that dogs all of this type of work: GT's, task-structures, KADS, Clancey's SSM's (situation-specific models) and so on. That issue can be formulated as a series of questions about the content theory of tasks (or inferences): What are the task-level terms? In what sense are they primitives? How do they relate to one another? Is there any sense in which we can determine and agree on a set of terms that are not idiosyncratic to each research paradigm, but in fact represent some universally sharable set of analytic primitives?

Let us take some examples. KADS proposes a list of inferences as primitives in terms of which they wish to analyze the problem solving behavior of experts. That is, all information processing verbs in a natural language description of the behavior of the expert will be mapped into equivalence classes represented by these inference primitives. Similarly, while the GT view does not propose an exhaustive list of terms as KADS does, still for problems of diagnosis and design, a person trained in the GT view would be armed with terms like "classify," "assess hypothesis," "instantiate plan," and would decompose the behavior into modules like the above.

Earlier we discussed Clancey's proposal that what problem solvers do is to construct situation-specific models (SSM's) of some system in the world. He proposes that all SSM's can be expressed in terms of sets, memberships and relations. The set-relation view of the SSM is no doubt a good starting point, but it doesn't completely specify the content theory of inference operators for the task. We need to know what kinds of objects and relational operators are going to be in the vocabulary for the SSM for a given task. The set-theoretic view, or the point that what is being constructed is an SSM, does not by itself specify that in diagnosis the kinds of sets we are interested in are malfunctions, symptoms, and the kinds of relations are explanatory causal relations; or that in design, the objects of interests are components and devices, the relationships of interest are functional and physical connections. Further, we pointed out earlier the need for inferential operators (in addition to operators that stand for relations between domain objects) that arise from the strategies used in the methods. These operators are not part of the vocabulary of SSM's for the design task.

Thus whether we are following the GT/TS, KADS or SSM framework, we need a principled theory of inference operators, and how they connect to world models that we construct and the tasks that we perform. The hope that there may be small number of such operators which are in some sense primitives is present in several frameworks. But the proposals on the table for such primitives are either not complete, or not rich enough, or if nominally complete, not sufficiently well-motivated to explain why those terms and not others, or even the sense in which they are primitives at all.

Aristotle, and Kant following him, proposed rather similar content theories of fundamental categories of thought and experience: quality, quantity, order, space, causation, etc. It appears to us that when we think about the world, when we solve problems, we are trying to build models which are ultimately couched in these basic categories. Taking seriously the notion of problem solving as model building, we can say

that our inference operators arise ultimately from the primitives anchored on the Kantian categories of thought and experience. Theories of generic tasks and methods will eventually have to be grounded in such a categorical content theory of our thought. Until such a thoroughgoing content theory is available, we will have to be content with pragmatic criteria as the basis for selecting from among alternative proposals for knowledge primitives.

6 Concluding Remarks

Among the many ideas that we have covered in this paper, the most important one is the emphasis on content issues in knowledge for problem solving. Clearly some of the content of knowledge is so specific to the individual or a domain that it cannot be of much interest for the theory or technology of artificial intelligence. The content issues that we have emphasized are the regularities in knowledge use that cut across individual instances of the problems, and become applicable to classes of problems. Task-level analyses are ways in which we exploit these regularities. How much of these content characterizations are part of the definition of intelligence, and how much are just analyses of knowledge that the agent has acquired, is a philosophical issue that has to do with one's prejudices about how much of intelligence has to do with form and how much with content. But from the viewpoint of creating knowledge technologies, the scientific issue is elucidation of these regularities in knowledge and its use. The original notion of Generic Tasks helped nudge the field into a consideration of content issues a decade or so ago. The notion of Task Structures, in our view, provides an analysis tool for a more sophisticated marriage of form and content in knowledge systems.

7 Acknowledgment

The work on generic tasks and task-specific architectures has been done jointly over the years with a number of colleagues to whom we offer heartfelt thanks. This paper has benefited from comments by Jack W. Smith, Tom Bylander, Susan and John Josephson, Ashok Goel, Dean Allemang, Jon Sticklen, Bill Punch and Jean-Marc David. Chandrasekaran's work on GT's has been supported over the years by AFOSR and DARPA. The preparation of this paper in particular was supported by DARPA under AFOSR contract F-49620-89-C-0110. Todd Johnson's work on this paper was supported by National Heart Lung and Blood Institute grant HL-38776, and National Library of Medicine grant LM-04298.

References

1. Allemang, D., Rothenfluh, T.E.: Acquiring knowledge of knowledge acquisition: a self-study of generic tasks. *Current Developments in Knowledge Acquisition, Proc. of the Sixth European Knowledge Acquisition Workshop—EKAW 92*, (ed. Wetter, T., Acthoff, K.D., Gaines, B.R., Linster, M. & Schmalhofer, F.), 353-372, Springer-Verlag, Berlin, 1992

2. Benjamins, R.V., Abu-Hanna, A., Jansweijer, W.N.H.: Dynamic method selection in diagnostic reasoning. 12th Avignon International Congress on Artificial Intelligence, 155-164, 1992
3. Breuker, J., Wielinga, B.: Models of expertise in knowledge acquisition. Topics in Expert System Design, (ed. Guida, G. & Tasso, C.), 265-295, Elsevier Science Publishers B. V., North-Holland, 1989
4. Brown, D.C.: Expert Systems for Design Problem-Solving Using Design Refinement with Plan Selection and Redesign. Ph.D. Thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, Oh, 1984
5. Brown, D.C., Chandrasekaran, B.: Expert Systems for a Class of Mechanical Design Activity. Knowledge Engineering in Computer-Aided Design, (ed. Gero, J.S.), 259-282, North-Holland, New York, 1985
6. Brown, D.C., Chandrasekaran, B.: Design Problem Solving: Knowledge Structures and Control Strategies. Morgan Kaufmann Publishers, San Mateo, California, 1989
7. Bylander, T., Allemang, D., Tanner, M.C., Josephson, J.R.: The computational complexity of abduction. Artificial Intelligence, 49(1991):25-60, 1991
8. Bylander, T., Chandrasekaran, B.: Generic Tasks for knowledge-based reasoning: The "right" level of abstraction for knowledge acquisition. Int. J. Man-Machine Studies, 26:231-243, 1987
9. Bylander, T., Johnson, T.R., Goel, A.: Structured matching: A task-specific technique for making decisions. Knowledge Acquisition, 3(1):1-20, 1991
10. Bylander, T., Mittal, S.: CSRL: A language for classificatory problem solving. AI Magazine, VII(3):66-77, 1986
11. Chandrasekaran, B.: Towards a Taxonomy of Problem Solving Types. AI Magazine, 4(1):9-17, 1983
12. Chandrasekaran, B.: Generic tasks in expert system design and their role in explanation of problem solving. Proceedings of the National Academy of Sciences/Office of Naval Research Workshop on AI and Distributed Problem Solving, National Academy of Sciences, Washington, D.C., 1985
13. Chandrasekaran, B.: Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. IEEE Expert, 1(3):23-30, 1986
14. Chandrasekaran, B.: Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, (ed. McDermott, J.), 1183-1192, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987
15. Chandrasekaran, B.: Task-structures, knowledge acquisition, and learning. Machine Learning, 4:93-99, 1989
16. Chandrasekaran, B.: Design Problem Solving: A Task Analysis. AI Magazine, 11(4):59-71, 1990
17. Chandrasekaran, B.: Models versus rules, deep versus compiled, content versus form: some distinctions in knowledge systems research. IEEE Expert, April:75-79, 1991
18. Chandrasekaran, B., Mittal, S.: Conceptual representation of medical knowledge for diagnosis by computer: MDX and related systems. Advances in Computers, (ed. Yovits, M.), 217-293, Academic Press, 1983
19. Chandrasekaran, B., Mittal, S., Gomez, F., Smith, J.W.: An approach to medical diagnosis based on conceptual structures. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, 134-142, IJCAI, Tokyo, Japan, 1979

20. Chandrasekaran, B., Mittal, S., Smith, J.W.: RADEX—Toward a Computer-Based Radiology Consultant. Pattern Recognition in Practice, (ed. Kanal & Gelsema), 463-474, North Holland Publishing Co., 1980
21. Chandrasekaran, B., Mittal, S., Smith, J.W.: Reasoning with uncertain knowledge: The MDX approach. Proceedings of the 1982 Congress of the American Medical Informatics Association, (ed. Lindberg, D.A.B.), 335-339, Masson Publishing, U.S.A, 1982
22. Chandrasekaran, B., Tanner, M., Josephson, J.: Explaining Control Strategies in Problem Solving. IEEE Expert, 4(1): pp. 9-24., 1989
23. Clancey, W.J.: Heuristic classification. Artificial Intelligence, 27(3):289-350, 1985
24. Clancey, W.J.: From GUIDON to NEOMYCIN and HERACLES in twenty short lessons: ORN final report 1979-1985. AI Magazine, 7(3):40-60, 1986
25. David, J.-M.: Functional architectures and the Generic Task approach. Knowledge Engineering Review, 3(3):212-215, 1988
26. David, J.M., Krivine, J.P.: Diva: An expert system for vibration-based monitoring of large rotating machinery. Technical Report, Laboratoires de Marcoussis, France, 1988
27. Eshelman, L.: MOLE: A knowledge-acquisition tool for cover-and-differentiate systems. Automating Knowledge Acquisition for Expert Systems, (ed. Marcus, S.), 37-80, Kluwer Academic Publishers, 1988
28. Goel, A., Bylander, T.: Computational Feasibility of Structured Matching. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11(12):1312-1316., 1989
29. Goel, A., Callantine, T.: An Experience-Based Approach to Navigational Path Planning. Proceedings of the IEEE/Robotics Society of Japan International Conference on Robotics and Systems, 705-710, IEEE Press, 1992
30. Goel, A., Soundararajan, N., Chandrasekaran, B.: Complexity in Classificatory Reasoning. Proc. Sixth National Conference on Artificial Intelligence, 421-425, Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987
31. Goel, A.K., Chandrasekaran, B.: Case-based design: a task analysis. Artificial Intelligence in Engineering, (ed. Tong, C. & Sriram, D.), 165-183, Academic Press, NY, 1992
32. Gomez, F., Chandrasekaran, B.: Knowledge organization and distribution for medical diagnosis. IEEE Trans. Systems, Man and Cybernetics, 11(1):34-42, 1981
33. Gruber, T., Cohen, P.: Design for acquisition: Principles of knowledge system design to facilitate knowledge acquisition. International Journal of Man-Machine Studies, 26(2):143-159, 1987
34. Hadzikadic, M., Yun, D.: Characterization of application domains for the expert system technology. AAAI Workshop of High Level Tools for Knowledge-Based Systems, Laboratory for AI Research, The Ohio State University, Columbus, 1986
35. Harvey, A.M.: Differential Diagnosis, The Interpretation of Clinical Evidence. W. B. Saunders, 1972
36. Herman, D.J.: An Extensible, Task-Specific Shell for Routine Design Problem Solving. Ph.D. Thesis, Department of Computer and Information Science, The Ohio State University, Columbus, Oh, 1992
37. Iwasaki, Y., Keller, R., Feigenbaum, E.: Generic tasks or wide-ranging knowledge bases? The Knowledge Engineering Review, 3(3):215-216, 1988

38. Johnson, K., Sticklen, J., Smith, J.W.: IDABLE—Application of an intelligent data base to medical systems. Working Notes of the 1988 AAAI Spring Symposium on Artificial Intelligence in Medicine, 43-44, AAAI, Stanford, Ca., 1988
39. Johnson, K.A., Johnson, T.R., Smith, J.W., Jr., DeJongh, M., Fischer, O., Amra, N.K., Bayazitoglu, A.: RedSoar—A system for red blood cell antibody identification. Proceedings of the Fifteenth Annual Symposium on Computer Applications in Medical Care, 664-668, McGraw Hill, Washington D.C., 1991
40. Johnson, T.R.: Generic Tasks in the Problem-Space Paradigm: Building Flexible Knowledge Systems While Using Task-Level Constraints. Ph.D. Thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, Oh, 1991
41. Johnson, T.R., Smith, J.W., Bylander, T.: HYPER—Hypothesis matching using compiled knowledge. Proceedings of the AAMSI Congress 1989, (ed. Hammond, W.E.), 126-130, American Association for Medical Systems and Informatics, San Francisco, California, 1989
42. Johnson, T.R., Smith, J.W., Chandrasekaran, B.: Generic tasks and Soar. Working Notes of the AAAI-89 Spring Symposium on Knowledge System Development Tools and Languages, 25-28, AAAI, Stanford University, 1989
43. Johnson, T.R., Smith, J.W., Chandrasekaran, B.: Task-specific architectures for flexible systems. The Soar Papers: Research on Integrated Intelligence, (ed. Rosenbloom, P.S., Laird, J.E. & Newell, A.), The MIT Press, In press
44. Josephson, J., Chandrasekaran, B., Smith, J., Tanner, M.: A mechanism for forming composite explanatory hypotheses. IEEE Transactions on Systems, Man, and Cybernetics, 17(3):445-454, 1987
45. Josephson, J., Josephson, S.: Abduction: Computation, Philosophy, Technology. Cambridge University Press, In Press
46. Josephson, J., Smetters, D., Fox, R., Oblinger, D., Welch, A., Northrup, G.: Integrated Generic Task Toolset—Fafner Release 1.0: Introduction and User's Guide. Technical Report, Laboratory for AI Research, The Ohio State University, Columbus, Oh, 1989
47. Laird, J.E., Newell, A., Rosenbloom, P.S.: SOAR: An architecture for general intelligence. Artificial Intelligence, 33:1-64, 1987
48. Marcus, S.: Salt: A knowledge acquisition tool for propose-and-revise systems. Automating Knowledge Acquisition for Expert Systems, (ed. Marcus, S.), 81-123, Kluwer Academic Publishers, Boston, 1988
49. McDermott, J.: Preliminary steps toward a taxonomy of problem-solving methods. Automating Knowledge Acquisition for Expert Systems, (ed. Marcus, S.), 225-256, Kluwer Academic Publishers, 1988
50. Miller, R.A., Pople, H.E., Jr., Myers, J.D.: Internist I, An Experimental Computer-Based Diagnostic Consultant for General Internal Medicine. The New England Journal of Medicine, 307:468-476, 1982
51. Minsky, M.: The Society of the Mind. Simon and Schuster, 1985
52. Mittal, S.: Event-Based Organization of Temporal Data Bases. Proceedings of the Fourth Biennial Conference of the Canadian Society for Computational Studies of Intelligence, 164-171, CSCSI, Toronto, Ontario, 1982
53. Mittal, S., Chandrasekaran, B.: Patrec: A knowledge-directed database for a diagnostic expert system. IEEE Computer, 17(9):51-58, 1984

54. Musen, M.A.: Generation of Model-Based Knowledge-Acquisition Tools for Clinical-Trial Advice Systems. PhD Thesis, Stanford, 1988
55. Narayanan, N.H., Chandrasekaran, B.: Reasoning visually about spatial interactions. Proc. 12th IJCAI, 360-365, Morgan Kaufman, Mountain View, CA, 1991
56. Newell, A.: The Knowledge Level. *AI Magazine*, (Summer):1-19, 1981
57. Patil, R.S.: Causal Representation of Patient Illness for Electrolyte and Acid-base Diagnosis. Ph.D. Thesis, Massachusetts Institute of Technology, 1981
58. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufman, 1988
59. Peng, Y., Reggia, J.A.: Abductive Inference Models for Diagnostic Problem-Solving. Springer-Verlag, New York, 1990
60. Pople, H.: On the mechanization of abductive logic. Proc. of the International Joint Conference on Artificial Intelligence, 147-152, IJCAI, 1973
61. Punch III, W.F., Tanner, M.C., Josephson, J.R., Smith, J.W.: Peirce: A tool for experimenting with abduction. *IEEE Expert*, 5(5):34-44, 1990
62. Punch, W.F.: A Diagnosis System Using a Task Integrated Problem Solver Architecture (TIPS), Including Causal Reasoning. Ph.D. Thesis, Department of Computer and Information Science, The Ohio State University, Columbus, Oh, 1989
63. Reiter, R.A.: A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57-95, 1987
64. Schank, R.C.: Conceptual dependency: a theory of natural language understanding. *Cognitive Psychology*, 3:552-631, 1972
65. Sembugamoorthy, V., Chandrasekaran, B.: A Representation for the Functioning of Devices that Supports Compilation of Expert Problem Solving Structures. Experience, Memory and Reasoning, (ed. Kolodner, J.L. & Riesbeck, C.K.), 47-73, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986
66. Steels, L.: Components of expertise. *AI Magazine*, 11(2):28-49, 1990
67. Sticklen, J.: MDX2 An Integrated Medical Diagnostic System. Ph.D. Dissertation, The Ohio State University, 1987
68. Tanner, M.C.: Explaining Knowledge Systems: Justifying Diagnostic Conclusions. Ph.D. Thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, Oh, 1989
69. Van Marcke, K.: A generic tutoring environment. Proc. ECAI-90, (ed. Aiello, L.), 655-660, Pitman, London, 1990
70. Vanwelkenhuysen, J., Rademakers, P.: Mapping knowledge-level analysis onto a computational framework. Proc. ECAI-90, (ed. Aiello, L.), 681-686, Pitman, London, 1990
71. Weintraub, M.A.: An Explanation-Based Approach to Assigning Credit. Ph.D. Thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, Oh, 1991
72. Wielinga, B.J., Schreiber, A.T., Breuker, J.A.: KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4:5-53, 1992
73. Wilks, Y.A.: A preferential pattern-seeking semantics for natural language inference. *Artificial Intelligence*, 6:53-74, 1975
74. Wittgenstein, L.: Proposition 560, *Philosophical Investigations*. McMillan, New York, 1953