Design Problem Solving: A Task Analysis[1]

B. Chandrasekaran

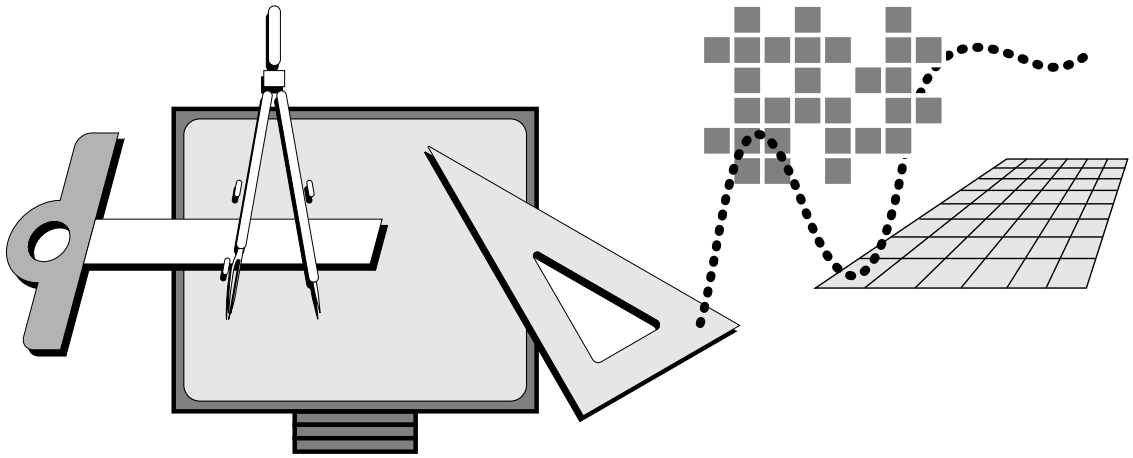
I propose a task structure for design by analyzing a general class of methods that I call propose-critique-modify methods. The task structure is constructed by identifying a range of methods for each task. For each method, the knowledge needed and the subtasks that it sets up are identified. This recursive style of analysis provides a framework in which we can understand a number of particular proposals for design problem solving as specific combinations of tasks, methods, and subtasks. Most of the subtasks are not really specific to design as such. The analysis shows that there is no one ideal method for design, and good design problem solving is a result of recursively selecting methods based on a number of criteria, including knowledge availability. How the task analysis can help in knowledge acquisition and system design is discussed.

Design problem solving is a complex activity involving a number of subtasks and a number of alternative methods potentially available for each subtask. The structure of tasks has been a key concern of recent research in task-oriented methodologies for knowledge-based systems (Chandrasekaran 1986; Clancey 1985; Steels 1990; McDermott 1988). One way to conduct a task analysis is to develop a task structure (Chandrasekaran 1989) that lays out the relation between a task, applicable methods for it, the knowledge requirements for the methods, and the subtasks set up by them. The major goal of this article is to develop a task structure for design as a knowledge-based problem-solving activity.

Design as Search in a Space of Subassemblies

Designing artifacts that are meant to achieve some functions within some constraints is an important class of design with characteristic properties (Goel and Pirolli 1989). I concentrate on this class of design problems in this article.

For sufficiently complex versions of the design problem, a common theme emerges for design as a process: It involves mappings from the space of design specifications to the space



of devices or components (often referred to as mapping from behavior to structure), typically conducted by means of a search or exploration in the space of possible subassemblies of components. This accent on assembly is in fact the origin of the frequent suggestion that design is a synthetic task.

The design problem is formally a search problem in a large space for objects that satisfy multiple constraints. Only a vanishingly small number of objects in this space constitute even satisficing, not to mention optimal, solutions. What is needed to make design practical are strategies that radically shrink the search space.

Set against the view of design as a deliberative problem-solving process is the view of design as an intuitive, almost instantaneous, process where a design solution comes to the mind of the designer. Artistic creations and scientific theories are often said by their creators to have occurred to them in this manner. Even when a plausible solution occurs in this way, the proposal still needs to be evaluated, critiqued, and modified by deliberatively examining alternatives. That is, except in simple cases, deliberative processes are still essential for real-world design.

Functions, Constraints, Components, and Relations

A designer is charged with specifying an artifact that delivers some functions and satisfies some constraints. For each design task, the availability of a (possibly large and generally only implicitly specified) set of primitive components can be assumed. The domain also specifies a repertoire of primitive relations or connections that are possible between components. An electronics engineer, for example, might assume the availability of transistors, capacitors, and other electrical components when designing a waveform generator. Primitive relations in this domain are serial and parallel connections between components.

Of course, design is, in general, recursive: If a certain component that was assumed to be available is in fact not available, the design of this component can be undertaken in the next round. However, the vocabulary of primitive components and relations can be different from that for the original device.

Functions can be expressed as a state or a series of states that we want the device to achieve or avoid under specified conditions. Functions can be explicitly stated as part of problem specifications, or they can be implicit in the designer's understanding of the domain. An example of an implicit function in many engineering devices is safety: For example, a subsystem's role might only be explained as something that prevents the leakage of a potentially hazardous substance, and this function might never be explicitly stated as part of the design specification (Keuneke 1989).

In addition to desired functionalities, design specifications will usually mention a number of constraints.[2] The distinction between functions and constraints is hard to formally pin down; functions are constraints on the behavior or properties of the device. However, it is useful to distinguish functions from other constraints because functions are the primary reason that the device is desired. Design constraints can be on the properties of the artifact (for example, "It should not weigh more than . . ."), the process of making the artifact from its description (manufacturability constraints), the design process itself (for example, "I want a design within a week"), and so on. A computationally effective process of design is to generate a candidate design based on functions and then modify it to meet the constraints.

Definition of the Design Task

Consider the following definition of the design task:

Definition: The design problem is specified by (1) a set of functions (those explicitly stated by the design consumer as well as those implicitly defined by the domain) to be delivered by an artifact and a set of constraints to be satisfied and (2) a technology, that is, a repertoire of components assumed to be available and a vocabulary of relations between components. The constraints might pertain to the design parameters themselves, the process of making the artifact, or the design process. The solution to the design problem consists of a complete specification of a set of components and their relations that together describe an artifact that delivers the functions and satisfies the constraints. The solution is expected to satisfy a set of implicit criteria as well; for example, it is not much more complex or costly than plausible alternatives (ruling out Rube Goldberg devices).

The preceding definition also captures the domain-independent character of design as a generic activity. Planning, programming, and engineering design all share this definition as well as many of the subprocesses to a significant degree. Nevertheless, there are versions of the design problem for which this defini-

tion needs to be modified or extended, as in the following examples: First, at the start of the design process, only a minimal statement of functions and constraints might be available, and additional ones might be developed in parallel with the design process itself. Second, some design problems involve extensive trade-off studies, where a part of the design process is a search for ways in which the functions or the constraints can be relaxed or otherwise modified. Third, tinkering is a time-honored method of invention, where the design space is explored without any specific set of functions in mind. Functions can be identified for structural configurations that arise during exploration. Fourth, the world of primitive objects can be open ended and only implicitly specified. The design framework that I present can be extended to cover these variations.

The Task Structure

Let us say we have a problem-solving task *T*, and let *M* be some method suggested for the task.[3] A method can be described in terms of the operators it uses, the objects it operates on, and any additional knowledge about how to organize operator application to satisfy the goal. At the knowledge level, the method is characterized by the knowledge the agent needs to set up and apply the method. Different methods for the same task might call for knowledge of different types.

To take a simple example, for the task of multiplying two multidigit numbers, the logarithmic method consists of the following series of operations: Extract the logarithm of each of the input numbers, add the two logarithms, and extract the antilogarithm of the sum. (The operators appear in a typewriter-like font.) Their arguments, as well as the results, are the objects of this method.)

Note that one does not typically include (at this level of description of the logarithmic method) specifications about how to extract the logarithm or the antilogarithm or how to do the addition. If the computational model does not provide these capabilities as primitives, the performance of these operations can be set up as subtasks of the method. Thus, given a method, the application of any of the operators of a method can be set up as a subtask. Some of the objects a method needs can be generic to a class of problems in a domain. As an example, consider hierarchical classification using a malfunction hierarchy, a common method of diagnosis. Establish-Hypothesis and Refine-Hypothesis operations are applied to the hypotheses in the hierarchy. These objects are useful for solving many instances of the diagnostic problem in the domain. If the malfunction hypotheses are not directly available, the generation of such hypotheses can be set up as subtasks. A common method for the generation of such objects is compilation from so-called deep knowledge. Structure-function models of the device that is being diagnosed have been proposed and used as deep models to generate malfunction hypotheses (Chandrasekaran, Smith, and Sticklen 1989). There is no finite set of mutually distinct methods for a task because there can be numerous variants on a method. Nevertheless, the term method is a useful shorthand to refer to a set of related proposals about organizing computation.

. . . a task structure. . . lays out the relation between a task, applicable methods for it, the knowledge requirements for the methods, and the subtasks set up by them

Types of Methods

One type of method is of particular importance in knowledge-based systems: methods that can be viewed as a problem space search (Newell 1980). Designer-Soar (Steier 1989) and Air-Cyl (Brown and Chandrasekaran 1989) are examples of design systems that explore search spaces. For example, Air-Cyl can be understood as searching in a space of parameters for the components of an air cylinder by using design plans that propose and modify parameter values.

Another class of methods consists of algorithms that directly produce a solution without any search in a space of alternatives, for example, producing a set of design parameters by numerically solving a set of simultaneous equations. Such algorithms are only available for so-called well-structured problems.[4] Most real-world problems are ill structured, and the role of domain knowledge is to help set up spaces of alternatives and help control the search in these spaces.

A task analysis of this type can be recursively continued until methods whose operators are all directly achievable (within the analysis framework) are reached. In the following task analysis for design, I explicitly indicate as subtasks only those to which I want to draw specific attention. Other operators might exist that also require additional problem solving.

A Task Structure for Design: The Propose-Critique-Modify Family of Methods

The most common top-level family of methods for design can be characterized as propose-critique-modify (PCM) methods. These methods have the subtasks of proposing partial or complete design solutions, verifying proposed solutions, critiquing the proposals by identifying causes of failure if any, and modifying proposals to satisfy design goals. These subtasks can be combined in fairly complex ways, but the following method is one straightforward way in which a PCM method can organize and combine the subtasks.

Example PCM Method:

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, and return to 2.

Although all the PCM methods need to have some way to achieve the iteration in step 4, 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. It is hard to identify a separate method for each such variation. The implications for a design architecture of this open-endedness in the number of methods are discussed in the concluding sections of this article.

In this article, most of the attention is devoted to the proposal subtask because most of the design knowledge as such is used in this subtask. Every task has a *default method*, one that uses compiled knowledge to get a solution without any problem solving. This method is practical only in simple cases. Because this method is potentially applicable to simple versions of all tasks and has no interesting internal structure, I do not explicitly mention it in my discussion.

A task analysis should provide a framework within which various approaches to design can be understood. I use selected examples of existing AI systems to illustrate the ideas, but there is no attempt to provide a survey of all AI work on design.

Methods for Proposing Design Choices

Design proposal methods use domain knowledge to map part or all of the specifications to partial or complete design proposals. Three groups of methods can be identified: (1) problem decomposition–solution composition, (2) retrieval of cases from memory, and (3) constraint satisfaction. First is problem decomposition–solution composition. In this class of 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 decomposition method. Second is the retrieval of cases from memory that correspond to solutions for design problems that are similar or close to the current problem. Third is the family of methods that solve the design problem as a constraint-satisfaction problem and use a variety of quantitative and qualitative optimization or constraint-satisfaction techniques.

Decomposition and case-based methods help reduce the size of the search spaces because the knowledge they use can be viewed as the compilation or chunking of earlier (individual or community) search in the design space. The conversion of a design problem into one amenable to global-optimization algorithms requires substantial a priori knowledge of the structure of the design problem.

Decomposition-Solution Composition. I treat this method in terms of all the features that an information-processing analysis calls for: the types of knowledge and information needed and the inference processes that operate on this form of knowledge.

Knowledge needed is of the form $D \rightarrow D1, D2, \ldots, Dn$, where D is a given design problem, and Dis are smaller subproblems (that is, associated with search spaces smaller than D). A number of alternate decompositions for a problem might be available, in which case a selection needs to be made, with the attendant possibility of backtracking and making another choice. Repeated applications of the decomposition knowledge produce design hierarchies. In well-trodden domains, effective decompositions are known, and little search for decompositions needs to be conducted as part of routine design activity. For example, in automobile design, the overall decomposition has remained largely invariant over several decades.

Decomposition knowledge in design generally arises when the functional specifications can be decomposed into a set of subfunctions (Freeman and Newell 1971). Design decom-

position knowledge can come in the form of part-subpart decomposition if a direct mapping is available between functions and components.

The following are two important subgoals of the decomposition–solution composition method: First is generating specifications for subproblems. The functional and other specifications on *D* need to get translated into specifications for each of the subproblems *D1, . . . , Dn*. Second is gluing the subproblem solutions into a solution to the original design problem.

In most routine design, these subtasks are not explicit: They are either solved by compiled knowledge, or the problem specification already implies a solution to these problems. In general, however, additional problem solving is needed.

How a decomposition–solution composition method might actually organize and use the subgoals is shown in the following example:

Example Decomposition–Solution Recomposition Method:

Step 1. (search the space of decompositions). Choose from among alternative decompositions for the given design problem *D*.

Step 2. Generate specifications for subproblems in the chosen decomposition.

Step 3. Set up each subproblem as a design problem. Solve them in some order determined by control strategies and other domain knowledge (for example, progressive deepening).

Step 4. If subproblems are solved, recompose solutions of subproblems into solution for *D,* and exit.

Step 5. If failure in steps 3 or 4, go to step 1 to make another choice, or relax specifications, and go to step 2.

All the caveats mentioned in connection with the PCM method apply here. Specifically, the control of how subproblems are solved can be variable and more complex than previously indicated. Some of the sources of this complexity are given in the following discussion.

Given a design problem, it might not always be possible to generate all the constraints for its subproblems from the original problem's specifications alone. In many domains, constraint generation for some subproblems alternates with partial design of others, which, in turn, provides additional information for constraints for yet other subproblems. There might be a complex process of commitments and backtracking. In extreme cases, most of design problem solving can consist of search for parameters that make all the subproblems solvable. For example, the propose-and-revise method (Marcus, McDermott, and Wang 1985) involves making commitments to some subparts of the design problem (propose part) and then revising these when some constraints for other parts of the problem are violated.

The most common top-level family of methods for design can be characterized as propose-critique-modify (PCM) methods.

In configuration tasks (Mittal and Frayman 1989), subproblem solutions are given as part of the problem (that is, the desired functions are mapped into a set of key components), and the remaining task is dominated by the subtasks of specification generation and solution recomposition. For components *A* and *B* to be connected, certain preconditions and postconditions might need to be satisfied. If these conditions are not available a priori, they need to be derived from configuration behaviors. Discovering connection conditions and checking whether specific configuration proposals result in desired functional behaviors can often involve simulation as a problem-solving method (for example, Kelly and Steinberg 1982).

There can be complex dependencies between constraints among subproblems. In situations where commitments for *D1* are going to constrain the specifications for *D2, . . . , Dn*, and the commitments for the latter might further specify constraints for *D1,* a strategy that Steier (1989) identified as progressive deepening is natural to emerge. This strategy involves making some commitment for each subproblem at each pass, using these commitments to generate additional specifications, undoing earlier commitments as needed, and repeating this process.

Control Issues. There are two sets of control issues, one dealing with which sets of decompositions to choose (step 1 in the decomposition-recomposition method) and the other with the order in which the subproblems within a given decomposition ought to be attacked (step 3). For the first problem, the decomposition will generally produce an AND or an OR node. All the decompositions in an AND node will need to be solved, but only one of the decompositions for an OR node will need to be solved. Finding the appropriate decomposition requires search in an AND/OR graph. Howev-

er, as a rule, such searches are expensive. In domains where multiple decompositions are possible, but there are no easily formalizable heuristics to choose among them, the machine might be effective in proposing alternatives and the human in evaluating them and making a selection.

In routine design, extensive searches in the spaces of possible decompositions are avoided by limiting the number of possible decompositions at each choice point to one or a few. Design hierarchies come about in those domains where the problem is sufficiently routine that only one decomposition is available to choose at each selection point.

Transformation methods (Balzer 1981) for algorithm synthesis are a type of decomposition method. In this approach, a set of high-level specifications for an algorithm are converted into a series of programming language–level commitments by recursively mapping subsets of specifications into components for which some implementation-level commitments have been made. Each such commitment will typically constrain other implementation commitments. Because of this constraint, search in the space of possible transformations might often be needed. In most implemented transformation systems, humans choose from a set of alternative transformations presented by the design system.

Regarding the order in which subproblems in a given decomposition are to be attacked, the main constraint is the knowledge about dependencies between subproblems that I just discussed. When the subproblems are organized in the form of a design hierarchy, the default control is top down, but actual control can be complicated. For example, a component at the leaf level of the design hierarchy might be the most limiting component and many other components and subsystems can only be designed after this component is chosen. Part of the design process in this case will appear to have a bottom-up flavor. In general, appropriate control strategies come about based on the dependencies between subproblems.

Design Plans. A special case of decomposition knowledge is the *design plan*, representing a precompiled partial solution to a design goal (Rich 1981; Johnson and Soloway 1985; Friedland 1979; Mittal, Dym, and Morjaria 1986; Brown and Chandrasekaran 1989). A design plan specifies a sequence of design actions to take for producing a design or part of a design. Design commitments made by a design plan can be abstract; that is, choices are not made at the level of primitive objects but at the intermediate level of design abstractions, which need to be further refined at the level of primitive objects. For example, in designing an automobile, a design plan might commit to the choice of a diesel engine as the power plant. Although this choice is a design proposal in the sense that a commitment is being made, the diesel engine design itself is not specified in detail at this stage but posed as a subtask to be solved by any of the available methods.

Thus, a design plan *D* can set up other design problems *D1, . . . , Dn* as subproblems, and in this sense, it is decomposition knowledge in a strong form: The process of transforming the main problem goals into goals to be allocated to subproblems and the method of putting solutions to the subproblems back together to obtain a solution to the original design problem are directly encoded in the plan.

Design plans can be indexed in a number of ways. Two possibilities are by design goal (for achieving <goal>, use <plan>) and by component (for designing <part>, use <plan>). Each goal or component can have a small number of alternative plans attached to them, with perhaps some additional knowledge that helps in choosing among them.

Control and inference issues in the use of plans are similar to those in the general case of decomposition: Alternate plans are possible, and in routine design, design plan hierarchies can emerge. The default control strategy can be characterized as "instantiate and expand." That is, the plan's steps specify some of the design parameters as well as calls to other design plans. Choosing an abstract plan and making commitments that are specific to the problem at hand are the instantiation process, and calling other plans for specifying details to portions is the expansion part.

A number of additional pieces of information might be needed or generated as this expansion process is undertaken. Information about dependencies between parts of the plan might need to be generated at run time (for example, discovering that certain parameters of a piston need to be chosen before those of the rod), and some optimizations might be discovered at run time (for example, the same base that was used to attach component *A* can also be used to attach component *B*). Noah (Sacerdoti 1975) is an early example of the run-time generation of dependencies and optimization.

Design Proposal by Case Retrieval. A major source of design proposal knowledge is the *design case*, an instance of successful past design problem solving. Cases can arise from an individual's problem-solving experience or that of an organization such as a design firm or a design community. Cases can be episodic (that is, represent one problem-solving episode) or can represent the result of abstraction and generalization over several episodes. Design plans can be considered fairly abstracted versions of numerous cases.

Sussman (1973) proposes that a design strategy is to choose an already-completed design that satisfies constraints closest to the ones that apply to the current problem and to modify this design for the current constraints. Schank (1982) emphasizes the importance of case-based problem solving in general. Recent work on case-based reasoning in planning and design (Hammond 1989; Goel and Chandrasekaran 1989a) explores this family of methods. In case-based reasoning, almost correct designs are obtained by searching a memory bank of previous cases for a design that solves a problem similar to the current one.

The heart of a case-based design proposal is *matching:* How to choose the design that is closest to the current problem? Clearly, some features of the cases are more important in matching than others. Some notion of prioritizing over goals, or difference ordering in the sense of means-end analysis, might be needed.

Indexing of cases with a rich vocabulary of the features of the case and the goals it satisfies is a key idea in case-based reasoning. Matching and retrieval can be driven by associative processes on these indexes. Much of the work in case-based planning has used domain-specific goals to index cases. For the problem of designing engineering artifacts, the design cases need to be indexed in terms of the output behaviors of interest. For example, Goel and Chandrasekaran (1989b) propose that design cases be indexed using their functions. More generally, they show how cases can be indexed by a causal representation that relates the structure of the device to its function, and how this method of indexing can help in retrieval. Goel (1989) has a proposal for how matching and retrieval can benefit from a principled representation for design goals and states for the device and the substances the device operates with.

Case-based design proposal has a lot in common with the use of analogical reasoning in design. Maher, Zhao, and Gero (1988) propose that analogical reasoning in design is at the heart of design creativity.

. . . cases can be indexed by a causal representation that relates the structure of the device to its function . . .

Design Proposal by Constraint Satisfaction. Under fairly strong assumptions, particular classes of design problems can be formulated as optimization, constraint satisfaction, or algebraic equation-solving problems. What is common to all these formulations is that the solution lies in a space determined by simultaneous constraints, and specific classes of computational algorithms are available to directly locate this space. In particular, when the structure of the design is already specified, but parameters are determined by the specifics of a design problem, numeric or symbolic optimization techniques can be useful for design proposal. Linear, integer, and dynamic programming techniques have been used to solve design problems formulated in this manner.

Some versions of the constraint-satisfaction problem can be solved by constraint propagation. Constraints can be propagated in such a way that the component parameters are chosen to incrementally converge on a set that satisfies all the constraints (Stefik 1981).

Formally, all design can be thought of as constraint satisfaction, and one might be tempted to propose global constraint satisfaction as a universal solution for design. However, unless knowledge is used to reduce the size of the space (for example, by decomposing problems into smaller problems), design by constraint propagation can be computationally intractable. Knowledge such as decomposition can create subproblems with sufficiently small problem spaces where constraint-satisfaction methods can work without excessive search.

Verification

Verification involves checking that the design proposal satisfies functional and other specifications. There are two families of methods for this subtask: First, attributes of interest can be directly calculated or estimated by means of domain-specific algorithms or formulas (for example, the use of algebraic formulas to

calculate total weight or cost or the use of finite-element methods to calculate stress distribution). Direct calculation methods are not of much interest from an AI point of view. Second, behaviors of interest can be derived by simulation. These behaviors can be checked against requirements.

Simulation takes a description of the system structure as input and generates the behaviors of interest as output. The methods used in simulation should mirror the rules by which the behavior of the component assemblages is composed from the component properties. Quantitative simulation methods use equations that directly describe the results of this composition. These equations again are domain specific. For example, differential equations can be used to describe the behavior of a reaction in a reaction vessel. The structural description in a proposed design of a reaction vessel can be translated into parameters of the differential equation and the equation simulated to derive behaviors of interest.

There are generic AI techniques for generating behavior from structure that could be useful for simulation. Qualitative simulation (see Forbus [1988] for a survey of the current state of the art), consolidation (Bylander 1988), and functional simulation (Sticklen 1987) are examples of AI techniques that are available for deriving behaviors given structure. A proposed design can be simulated under various input conditions and the behavior evaluated. All these techniques take a structural description as input and using qualitative descriptions of component behaviors and rules of composition mimic the operation of the device to produce qualitative descriptions of behavior. Qualitative and quantitative simulation can alternate: A qualitative simulation can identify behaviors likely to be in unacceptable ranges, and a more focused quantitative procedure can be used to get more precise values.

Visual Simulations. *Visual simulation* of artifacts is widely used by human designers in verification. Designs are imagined, represented, and communicated pictorially in domains such as architecture and mechanical engineering. (See Goel and Pirolli [1989] for design protocol studies that show the prevalence of images during design.) It is clear that there is a need for pictorial representations and symbolic representations to coexist in design systems. A major use of imaginal representations is in the simulation of design proposals, but they also play a role in making design proposals by analogy with other domains. Little AI research has been done to date on visual representations that have the qualities needed for pictorial reasoning and imagination and also the symbolic properties needed for arbitrary referencing and composition by parts. A beginning in this direction is proposed in Chandrasekaran and Narayanan (1990), and the use of such representations for simulation is discussed in Narayanan and Chandrasekaran (1990).

Critiquing

Critiquing is the subtask in which the causes of a design's failure are analyzed: Parts of the structure are identified as potentially responsible for the unacceptable behavior or constraint violation. Critiquing is really a generalized version of the *diagnostic problem*, that is, a problem of mapping from undesirable behavior to parts of the structure responsible for the behavior. Modification of design can be directed to these candidates. Of course, localization of responsibility for failure does not always work: The entire approach to the design might need to be changed.

What is needed for criticism is information about how the structure of the device contributes to (or is intended to contribute to) the desired overall behavior. An AI method that is commonly used for this subtask is dependency analysis (Stallman and Sussman 1977). This method is applicable if explicit information is available in the form of *dependencies*, that is, knowledge that explicitly relates types of constraint or specification violations to prior design commitments. For example, if the total weight of a proposed design is higher than the weight limit, domain-specific knowledge is usually available that identifies parts whose weights are both sufficiently large and can be adjusted. Dependencies can be discovered by analyzing preconditions and postconditions of design operators. For example, if a certain output value (say, voltage in an electronic device) of a proposed design is excessive, the input to the output stage can give information about which of the components upstream might have contributed to the specific output. This dependency analysis might identify potential candidates, which might be verified by simulation.

Most of the proposals for critiquing that have been made in the case-based reasoning literature use domain-specific critics and are variations on precompiled patterns of relating output behavior to possible changes. Goel's (1989) approach to critiquing a design proposal is based on a functional analysis of the proposed design. If a design proposal is

TASK	METHODS	SUBTASKS
Design	Propose, Critique,Modify family (PCM)	Propose, Verify, Critique, Modify
Propose	Decomposition methods (incl. Design Plans) and Transformation methods	Specification generation for subproblems
		Solution of subproblems generated by decomposition (another set of Design-tasks)
		Composition of subproblem solutions
	Case-based methods	Match and retrieve similar case
	Global constraint-satisfaction methods	
	Numerical optimization methods	
	Numerical or Symbolic constraint propagation methods	
Specification generation for subproblems		
	Constraint propagation incl. constraint posting	Simulation to decide how constraints propagate
Composition of subproblem solutions	Configuration methods	Simulation for prediction behavior of candidate configurations
Verify	Domain-specific calculations or simulation	
	Qualitative simulation, Consolidation	
	Visual simulation	
Critique	Causal behavioral analysis techniques to assign responsibility	
	Dependency-analysis techniques	
Modify	Hill-climbing-like methods which incrementally improve parameters	
	Dependency-based changes	
	Function-to-structure mapping knowledge	
	Add new functions	Design new function. Recompose with candidate design

Table 1. The Task Structure for Design.
For each task, there is a default compiled knowledge method that has domain-specific knowledge to directly achieve it. (This method is not included here.) For subtasks such as critiquing, I only indicate families of generic AI methods, without explicit indication of their subtasks.

endowed with causal indexes that explicitly indicate the relation between structure and intended functions, then it is relatively easy to identify substructures for modification (Goel and Chandrasekaran 1989a).

Modification

Modification as a subtask takes information about the failure of a candidate design as its input and then changes the design to get closer to the specifications. Basically, what is required is changing a functional subpart of the proposed design or adding components to the proposed design to satisfy the design specifications. Depending on how informative failure analysis is and what types of knowledge are available, a number of problem-solving processes are applicable. Some of them are briefly outlined in the following paragraphs.

Modification can be driven by a form of means-end reasoning, where the differences are reduced in order of most to least significant. Especially useful here is knowledge that relates the desired changes in behavior to possible structural changes (Goel 1989).

A related search approach is one where modification is done by some form of hill climbing. In this method, parameters are changed, direction of improvement noted, and additional changes are made in the direc-

tion of maximal increment in some measure of overall performance. This method is especially applicable where the design problem is viewed as a parameter choice problem for a predetermined structure (for example, the Dominic system [Dixon, Simmons, and Cohen 1984]).

Modification is straightforward in dependency-directed methods. Once the dependency point is reached by backtracking, an alternative choice is simply made from the list of finite choices available. Some systems that perform routine design problems have explicit knowledge about what to do under different kinds of failures. This information can be attached to the design plans (DSPL [Brown and Chandrasekaran 1989]).

Criticism can reveal the need to add new functions. If these functions can be modularly added, that is, by creating and integrating separate substructures that deliver the functions, the design of the additional structures can simply be viewed as new design problems to be solved by all the methods available for design. The subtasks of generating specifications for these additional design problems and integrating their solutions were discussed in the section on problem decomposition and solution recomposition.

Discussion of the Task Structure

The task structure for design described in the preceding sections is summarized in table 1.[5] A *task structure* is a description of the task, proposed methods for it, the internal and external subtasks, knowledge required for the methods, and any control strategies for the method. Thus, the task analysis provides a clear road map for knowledge acquisition. How the analysis can be used to integrate the methods and goals is discussed in the following section.

Choice of Methods

How are methods to be chosen for the various tasks? The following is a set of criteria:

Properties of the solution: Some methods can produce answers that are precise, but others might only produce answers that are qualitative. Some of them might produce optimal solutions, and others might produce satisficing ones.

Properties of the solution process: Is the computation pragmatically feasible? How much time does it take? Memory?

Availability of knowledge required for the method to be applied: For example, a method for design verification might require that we have available a description of the behavior of the device as a system of differential equations; if this information is not directly available and if it cannot be generated by additional problem solving, the method cannot be used.

A delineation of the methods and their properties helps us to move away from abstract arguments about ideal methods for design. Each method in a task structure can be evaluated for appropriateness in a given situation by asking questions reflecting these criteria. Although some of this evaluation can take place at problem-solving time, much of it can be done when the knowledge system is designed; this evaluation can be used to guide a knowledge system designer in the choice of methods to implement.

Different types of methods can be used for different subtasks. For example, a design system can use a knowledge-based problem-solving method for the subtask of creating a design but a quantitative method, such as a finite-element method, for the subtask of evaluating the design.

Implications for an Architecture for Design Problem Solving

Because of the multiplicity of possible methods and subtasks for a task, a task-specific architecture that is exclusively for design is not likely to be complete: Even though design is a generic activity, there is no one generic method for it. Further, note that subtasks such as simulation are not particularly specific to design as a task. Thus, if the knowledge for these modules is embedded within a design architecture, either they will be unavailable for other tasks that require simulation as a subtask, or the knowledge for these tasks will need to be replicated. Thus, instead of building monolithic task-specific architectures for such complex tasks, a more useful architectural approach is one that can invoke different methods for different subtasks in a flexible way.

Following the ideas in the work on task-specific architectures, we can support methods by means of special-purpose shells that can help encode knowledge and control problem solving. This approach is an immediate extension of the generic task methodology (Chandrasekaran 1986). These methods can then be combined in a domain-specific manner; that is, methods for subtasks can be selected in advance and included as part of the application system, or methods can be recursively chosen at run time for the tasks

The task structure also makes clear how AI-like methods and other algorithnic or numeric methods can be flexibly combined, much as human designers alternate between problem solving in their heads and formal calculations.

based on the criteria listed in the previous subsection. For the latter approach, a task-independent architecture is needed with the capability of evaluating different methods, choosing one, executing it, setting up subgoals as they arise from the chosen method, and repeating the process. Soar (Rosenbloom, Laird, and Newell 1987), BB1 (Hayes-Roth 1985), and Tips (Punch 1989) are good candidates for such an architecture. This approach combines the advantages of task-specific architectures and the flexibility of run-time choice of methods. The DSPL++ work of Herman (1990) is an attempt at precisely this approach.

Using method-specific knowledge and strategy representations within a general architecture that helps select methods and set up subgoals is a good first step in adding flexibility to the advantages of the task-specific architecture view. However, it can also have limitations. For many real-world problems, switching between methods can result in control that is too large grained. Consider my earlier description of a PCM method. The method description calls for a specific sequence of how the operators of propose, and so on, are to be applied. Numerous variants of the method, with complex sequencing of the various operators, can be appropriate in different domains. It would be a hopeless task to try to support all these variants by method-specific architectures or shells. It is much better in the long run to let the task-method-subtask analysis guide us in the identification of the needed task-specific knowledge and let a flexible general architecture determine the actual sequence of operator application by using additional domain-specific knowledge. The subtasks can then be flexibly combined in response to problem-solving needs, achieving a much finer-grained control behavior. (See Johnson, Chandrasekaran, and Smith [1989] for realizing generic task ideas in Soar.)

The task structure also makes clear how AI-like methods and other algorithmic or numeric methods can be flexibly combined, much as human designers alternate between problem solving in their heads and formal calculations. For example, a designer might need to make sure that the maximum current in a proposed circuit is less than the limits for its components, and at this point, s/he might set up current and voltage equations and solve them. If s/he finds that the current in one branch of the circuit is more than the permitted limit, s/he might go back to critiquing the design to look for possible places to change it. The task structure view that I outlined shows how computer-based design systems can also similarly engage in a flexible integration of problem solving and other forms of algorithmic activity. The key is that the top-level control is goal oriented and can set up subgoals and choose methods that are appropriate to the subgoal. If the appropriate method for a subtask is a numeric algorithm, this method can be invoked and executed, at which point control reverts to the top level for pursuing other goals.

Concluding Remarks

Over the last several years, a number of working systems have come to be able to perform some version of the design task in some domain. These design proposals do not always bring out what is common among the different design tasks. There have also been attempts to develop formal first-principle algorithms for design that are meant to cover all types of design. Such general algorithms are, however, computationally intractable and are not particularly helpful in identifying the sources of power and tractability in human design problem solving in most domains.

The view elaborated here is that there is a generic vocabulary of tasks and methods that are part of design and that design problems in different domains simply differ in the mixture of subtasks and methods. Expertise, that is, methods, and knowledge and control strategies for them, emerge over a period in

different domains to help tractably solve the task in a given domain. Thus, the key to understanding real-world design computationally is not in a uniform algorithm for design but in the structure of the task, showing how the tasks, methods, subtasks, and domain knowledge are related. The analysis also clarifies the relationship between task-specific architectures and more general-purpose architectures for knowledge systems.

Acknowledgments

Many ideas from my collaborations with the following individuals found their way into this article: David C. Brown and Ashok Goel on design problem solving and Tom Bylander, John Josephson, Todd Johnson, Jack W. Smith, and Jon Sticklen on generic tasks. I am thankful to John Gero, Ashok Goel, Mary Lou Maher, Dale Moberg, and David Steier for useful comments on earlier drafts. The usual caveat holds good that they don't necessarily agree with all of what I am saying in this article. Support from the Air Force Office of Scientific Research (grants 87-0090 and 89-0250) and the Defense Advanced Research Projects Agency (contracts F30602-85-C-0010 and F49620-89-C-0110) is gratefully acknowledged.

References

Balzer, R. 1981. Transformation Implementation: An Example. *IEEE Transactions on Software Engineering* SE-7: 3–14.

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

Bylander, T. C. 1988. A Critique of Qualitative Simulation from a Consolidation Point of View. *IEEE Systems, Man, and Cybernetics* 18(2): 252–268.

Chandrasekaran, B. 1989. Task Structures, Knowledge Acquisition, and Learning. *Machine Learning* 4:339–345.

Chandrasekaran, B. 1986. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design. *IEEE Expert* 1(3): 23–30.

Chandrasekaran B., and Narayanan, N. H. 1990. Integrating Imagery and Visual Representations. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, 670–677. Hillsdale, N.J.: Lawrence Erlbaum.

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

Dixon, J. R.; Simmons, M. K.; and Cohen, P. R. 1984. An Architecture for Application of Artificial Intelligence to Design. In Proceedings of the Twenty-First Design Automation Conference, 634–640. Washington, D.C.: IEEE Computer Society.

Forbus, K. D. 1988. Qualitative Physics: Past, Present, and Future. In *Exploring Artificial Intelligence*, eds. H. E. Shrobe and the American Association for Artificial Intelligence, 239–296. San Mateo, Calif.: Morgan Kaufmann.

Friedland, P. 1979. Knowledge-Based Experimental Design in Molecular Genetics. In Proceedings of the Sixth International Joint Conference on Artificial Intelligence, 285–287. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Goel, A. 1989. Integration of Case-Based Reasoning and Model-Based Reasoning for Adaptive Design Problem Solving, Ph.D. diss., Dept. of Computer and Information Science, The Ohio State Univ.

Goel, A., and Chandrasekaran, B. 1989a. Functional Representation of Designs and Redesign Problem Solving. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 1388–1394. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Goel, A., and Chandrasekaran, B. 1989b. Use of Device Models in Adaptation of Design Cases. In *Proceedings of the DARPA Workshop on Case-Based Reasoning,* ed. K. Hammond, 100–109. San Mateo, Calif.: Morgan Kaufmann.

Goel, V., and Pirolli, P. 1989. Motivating the Notion of Generic Design within Information-Processing Theory: The Design Problem Space. *AI Magazine* 10(1): 18–38.

Hammond, K. 1989. *Case-Based Planning: Viewing Planning as a Memory Task*. Boston: Academic.

Hayes-Roth, B. 1985. A Blackboard Architecture for Control. *Artificial Intelligence* 26:251–321.

Herman, D. J. 1990. DSPL++: A High-Level Language for Building Design Expert Systems with Flexible Use of Multiple Methods. Ph.D. diss., Dept. of Computer and Information Science, The Ohio State Univ. Forthcoming.

Johnson, L., and Soloway, E. 1985. PROUST: Knowledge-Based Program Understanding. *IEEE Transactions on Software Engineering* 11(3): 267–275.

Johnson, T.; Chandrasekaran, B.; and Smith, J. W., Jr. 1989. Generic Tasks and Soar, Working Notes of the AAAI Spring Symposium on Knowledge System Development Tools and Languages, 25–28. Menlo Park, Calif.: American Association for Artificial Intelligence.

Kelly, V. E.; and Steinberg, L. I. 1982. The CRITTER System: Analyzing Digital Circuits by Propagating Behaviors and Specification. In Proceedings of the Second National Conference on Artificial Intelligence, 284–289. Menlo Park, Calif.: American Association for Artificial Intelligence.

Keuneke, A. 1989. Machine Understanding of Devices: Causal Explanations of Diagnostic Conclusions, Ph.D. diss., Dept. of Computer and Information Science, The Ohio State Univ.

McDermott, J. 1988. A Taxonomy of Problem-Solv-

ing Methods. In *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, 225–256. Boston: Kluwer.

Maher, M. L.; Zhao, F.; and Gero, J. S. 1988. Creativity in Humans and Computers. *Knowledge-Based Design in Architecture*, eds. J. S. Gero and T. Oksala, 29–44. Helsinki: Helsinki University of Technology.

Marcus, S.; McDermott, J.; and Wang, T. 1985. Knowledge Acquisition for Constructive Systems. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 637–639. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Mittal, S., and Frayman, F. 1989. Towards a Generic Model of Configuration Tasks. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 1395–1401. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Mittal, S.; Dym, C.; and Morjaria, M. 1986. PRIDE: An Expert System for the Design of Paper Handling Systems. *IEEE Computer* 19(7): 102–114.

Narayanan, N. H., and Chandrasekaran, B. 1990. Qualitative Simulation of Spatial Mechanisms: A Preliminary Report, Technical Report, Laboratory for AI Research, The Ohio State Univ.

Newell, A. 1980. Reasoning, Problem Solving, and Decision Process: The Problem Space as a Fundamental Category. In *Attention and Performance*, volume 8, 693–718. Hillsdale, N.J.: Lawrence Erlbaum.

Punch, W. 1989. A Diagnosis System Using a Task-Integrated Problem Solver Architecture (TIPS), Including Causal Reasoning, Ph.D. diss., Dept. of Computer and Information Science, The Ohio State Univ.

Rich, C. 1981. A Formal Representation for Plans in the Programmer's Apprentice. In Proceedings of the Seventh International Joint Conference on Artificial Intelligence, 1044–1052. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Rosenbloom, P. S.; Laird, J. E.; and Newell, A. 1987. SOAR: An Architecture for General Intelligence. *Artificial Intelligence* 33:1–64.

Sacerdoti, E. D. 1975. A Structure for Plans and Behavior, Technical Report 109, AI Center, SRI, Menlo Park, Calif.

Schank, R. 1982. *Dynamic Memory: A Theory of Learning in Computers and People*. New York: Cambridge University Press.

Stallman, R., and Sussman, G. 1977. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence* 9:135–196.

Steels, L. 1990. Components of Expertise. *AI Magazine* 11(2): 28–49.

Stefik, M. 1981. Planning with Constraints. *Artificial Intelligence* 16:111–140.

Steier, D. 1989. Automating Algorithm Design within an Architecture for General Intelligence, Ph.D. diss., School of Computer Science, Carnegie-Mellon Univ.

Sticklen, J. 1987. MDX2: An Integrated Medical Diagnosis System, Ph.D. diss., Dept. of Computer and Information Science, The Ohio State Univ.

Sussman, G. J. 1973. A Computational Model for Skill Acquisition, Ph.D. diss., Massachusetts Institute of Technology. Also in *A Computational Model for Skill Acquisition*. 1975. New York: American Elsevier.

Notes

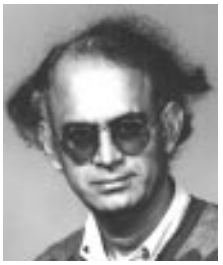
1. This work has evolved over a number of years. Earlier versions have appeared as chapter 2 of Brown and Chandrasekaran (1989) and in *Research in Engineering Design*. 1989. 1:75–86.

2. The constraints that are described as part of the design specification should be distinguished from the term constraint that appears in the description of design methods, such as constraint-directed problem solving.

3. In this article, I use the terms task and goal interchangeably.

4. I subscribe to the view that such algorithms are simply degenerate cases of search where the agent has sufficient knowledge to make the correct choice at each choice point. However, pragmatically speaking, it is best to think of algorithmic methods as a separate type because implementing them does not require search support in general.

5. The task structure described here is inherently incomplete: Additional methods can be identified for any subtask as a result of further research.



B. Chandrasekaran received a B.S. in engineering in 1963 from Madras University, India, and a Ph.D. from the University of Pennsylvania, Philadelphia, in 1967. From 1967 to 1969, he was research scientist with the Philco-Ford Corporation in Blue Bell, Pennsylvania, working on speech and character-recognition machines. He has been at Ohio State University, Columbus, since 1969. He is currently a professor of computer and information science and directs the Laboratory for AI Research. Currently, his major research activities are in knowledge-based reasoning, architecture of mind, and cognitive science. Chandrasekaran is editor in chief of *IEEE Expert* and serves on the editorial boards of numerous international journals. He was an invited speaker at the 1987 International Joint Conference on Artificial Intelligence, held in Milan, and has been awarded the University Distinguished Scholar Award by The Ohio State University. He is a Fellow of IEEE.