Chapter 1
INTRODUCTION

Chris Tong and Duvvuru Sriram

1.1. WHAT THIS BOOK IS ABOUT

What is *design*? Design is the process of constructing a description of an artifact that satisfies a (possibly informal) functional specification, meets certain performance criteria and resource limitations, is realizable in a given target technology, and satisfies criteria such as simplicity, testability, manufacturability, reusability, etc.; the design process itself may also be subject to certain restrictions such as time, human power, cost, etc.

Design problems arise everywhere, and come in many varieties. Some are born spontaneously amidst the circumstances of ordinary human lives: design a dish for dinner that uses last night's leftovers; design some kind of hook-like artifact that will enable me to retrieve a small object that fell down a crack; design a "nice-looking" arrangement of the flowers in a vase. Other design problems are small but commercial in nature: design a paper clip-like device that doesn't leave a mark on the paper; design a lamp whose light can be turned to aim in any particular direction; design an artifact for storing up to twenty pens and pencils, in an easily accessible fashion. Still other design problems are formidable, and their solutions can require the efforts and coordination of hundreds of people: design a space shuttle; design a marketable electric car; design an international trade agreement; etc.

Because design is so ubiquitous, anything generic we can say about the *design process* -- the activities involved in actually solving a design problem -- can have great impact. Even better would be to provide active help to designers.

This book is all about how ideas and methods from Artificial Intelligence can help engineering designers. By "engineering design", we primarily mean the design of *physical artifacts* or *physical processes* of various kinds. In this book, we will see the design of a wide variety of artifacts exemplified, including: cir-

Artificial Intelligence in Engineering Design
Volume I
Design Representation and Models of Routine Design

ISBN 0-12-660561-0

cuits and chips (Volume I, Chapters 2, 8, 12 and Volume II, 2, 8, 9, 10), swinging doors (Volume I, Chapter 6), copying machines (Volume I, Chapter 9 and Volume III, Chapter 6), cantilever beams (Volume I, Chapter 3), space telescopes (Volume II, Chapter 5), air cylinders (Volume I, Chapter 7), protein purifaction processes (Volume I, Chapter 10), fluid-mechanical devices (Volume II, Chapters 4 and 6), new alloys (Volume II, Chapter 7), graphics interfaces (Volume I, Chapter 14), automobile transmissions (Volume I, Chapter 4), spatial layouts (Volume I, Chapter 13), elevators (Volume I, Chapter 11), light-weight load-bearing structures (Volume II, Chapter 11), mechanical linkages (Volume II, Chapter 12), buildings (Volume III, Chapter 12), etc.

What you will not find in this book is anything on AI-assisted software design. On this point, our motivation is twofold: no book can (or should try to) cover everything; and AI and software engineering has already been treated in a number of edited collections (including [15, 30]).

This book is an edited collection of key papers from the field of AI and design. We have aimed at providing a state of the art description of the field that has coverage and depth. Thus, this book should be of use to engineering designers, design tool builders and marketeers, and researchers in AI and design. While a small number of other books have surveyed research on AI and design at a particular institution (e.g., [12, 31]), this book fills a hole in the existing literature because of its breadth.

The book is divided into three volumes, and a number of parts. This first chapter provides a conceptual framework that integrates a number of themes that run through all of the papers. It appears at the beginning of each of the three volumes. Volume I contains Parts I and II, Volume II contains Parts III, IV, and V, and Volume III contains Parts VI through IX.

Part I discusses issues arising in *representing* designs and design information. Parts II and III discuss a variety of models of the design process; Part II discusses models of routine design, while Part III discusses innovative design models. We felt that creative design models, such as they are in 1991, are still at too preliminary a stage to be included here. However, [11] contains an interesting collection of workshop papers on this subject. Parts IV and V talk about the formalization of common sense knowledge (in engineering) that is useful in many design tasks, and the reasoning techniques that accompany this knowledge; Part IV discusses knowledge about physical systems, while Part V gives several examples of formalized geometry knowledge. Part VI discusses techniques for acquiring knowledge to extend or improve a knowledge-based system. Part VII touches on the issue of building a knowledge-based design system; in particular, it presents a number of commercially available tools that may serve as modules within a larger knowledge-based system. Part VIII contains several articles on integrating design with the larger engineering process of which it is a part; in particular, some articles focus on designing for manufacturability. Finally, Part IX contains a report on a workshop in which leaders of the field discussed the state of the art in AI and Design.

Volume II, 2, 8, 9, 10), swing-
ines (Volume I, Chapter 9 and
ume I, Chapter 3), space tele-
Volume I, Chapter 7), protein
10), fluid-mechanical devices
ume II, Chapter 7), graphics in-
nsmissions (Volume I, Chapter
/ators (Volume I, Chapter 11),
II, Chapter 11), mechanical
ume III, Chapter 12), etc.
/thing on AI-assisted software
: no book can (or should try to)
ng has already been treated in a

.
pers from the field of AI and
e art description of the field that
be of use to engineering desig-
researchers in AI and design.
eyed research on AI and design
ook fills a hole in the existing

l a number of parts. This first
egrates a number of themes that
beginning of each of the three
me II contains Parts III, IV, and

designs and design information.
the design process; Part II dis-
III discusses innovative design
h as they are in 1991, are still at
wever, [11] contains an interest-
t. Parts IV and V talk about the
l engineering) that is useful in
hniques that accompany this
physical systems, while Part V
knowledge. Part VI discusses
or improve a knowledge-based
g a knowledge-based design sys-
ercially available tools that may
sed system. Part VIII contains
: larger engineering process of
ocus on designing for manufac-
a workshop in which leaders of
Design.

1.2. WHAT DOES AI HAVE TO OFFER TO ENGINEERING DESIGN?

In order to answer this question, we will first examine the nature of engineering design a little more formally. Then we will briefly summarize some of the major results in AI by viewing AI as a software engineering methodology. Next we will look at what non-AI computer assistance is currently available, and thus what gaps are left that represent opportunities for AI technology. Finally, we outline how the AI software engineering methodology can be applied to the construction of knowledge-based design tools.

1.2.1. Engineering Design: Product and Process

Engineering design involves mapping a specified *function* onto a (description of a) realizable physical *structure* -- the designed artifact. The desired function of the artifact is what it is supposed to do. The artifact's physical structure is the actual physical parts out of which it is made, and the part-whole relationships among them. In order to be realizable, the described physical structure must be capable of being assembled or fabricated. Due to restrictions on the available assembly or fabrication process, the physical structure of the artifact is often required to be expressed in some *target technology*, which delimits the kinds of parts from which it is built. A *correct design* is one whose physical structure correctly implements the specified function.

Why is design usually not a classification task [6], that is, a matter of simply looking up the right structure for a given function in (say) a parts catalog? The main reason is that the mapping between function and structure is not simple. For one thing, the connection between the function and the structure of an artifact may be an indirect one, that involves determining specified behavior (from the specified function), determining actual behavior (of the physical structure), and ensuring that these match. For another, specified functions are often very complex and must be realized using complex organizations of a large number of physical parts; these organizations often are not hierarchical, for the sake of design quality. Finally, additional non-functional constraints or criteria further complicate matters. We will now elaborate on these complications.

Some kinds of artifacts -- for example, napkin holders, coat hangers, and bookcases -- are relatively "inactive" in the sense that nothing is "moving" inside them. In contrast, the design of a *physical system* involves additionally reasoning about the artifact's *behavior*, both external and internal. The external behavior of a system is what it does from the viewpoint of an outside observer. Thus, an (analog) clock has hands that turn regularly. The internal behavior is

based on observing what the *parts* of the system do. Thus, in a clock, we may see gears turning. Behavior need not be so visible: electrical flow, heat transmission, or force transfer are also forms of behavior.

In a physical system, behavior *mediates* function and structure. The *function* is achieved by the *structure behaving* in a certain way. If we just possessed the physical structure of a clock, but had no idea of how it (or its parts) behaved, we would have no way of telling that it achieves the function of telling time.

Not only in a physical system but also in *designing* a physical system, behavior tends to act as intermediary between function and structure. Associated with a specified function is a *specified behavior*; we would be able to tell time if the angle of some physical structure changed in such a way that it was a function of the time. Associated with a physical structure is its *producible behavior*; for example, a gear will *turn*, provided that some rotational force is applied to it. In rough terms then, designing a physical system involves selecting (or refining) a physical structure (or description thereof) in such a way that its producible behavior matches the specified behavior, and thus achieves the desired function. Thus, we could successfully refine the "physical structure whose angle is a function of the hour" as either the hand on an electromechanical clock, or as the shadow cast by a sundial.

Complex functions often require complex implementations. For example, a jet aircraft consists of thousands of physical parts. Parts may *interact* in various ways. Thus the problems of *designing* the parts also interact, which complicates the design process. Such interactions (among physical parts or among the problems of designing those parts) can be classified according to their strength.

For instance, many parts of the aircraft (the wings, the engine, the body, etc.) must, together, behave in such a way that the plane stays airborne; thus the subproblems of designing these parts can be said to *strongly interact* with respect to this specification of global behavior. Non-functional requirements such as global resource limitations or optimization criteria are another source of strong interactions. For example, the total cost of the airplane may have to meet some budget. Or the specification may call for the rate of fuel consumption of the plane to be "fairly low". Not all ways of implementing some function may be equally "good" with respect to some global criterion. The design process must have some means for picking the best (or at least a relatively good) implementation alternative. Good implementations often involve *structure-sharing*, i.e., the mapping of several functions onto the same structure. For example, the part of the phone which we pick up serves multiple functions: we speak to the other person through it; we hear the other person through it; and it breaks the phone connection when placed in the cradle. Important resources such as "total amount of space or area" and "total cost" tend to used more economically through such structure-sharing. On the other side of the coin, allowing structure-sharing complicates both the representation of designs and the process of design.

That neighboring parts must fit together -- both structurally and behaviorally

-- exemplifies a kind of *weak* or *local interaction*. Thus the wings of the plane must be attachable to the body; the required rate of fuel into the engine on the left wing had better match the outgoing rate of fuel from the pump; and so forth. The *design process* must be capable of ensuring that such constraints are met.

1.2.2. Artificial Intelligence as a Software Engineering Methodology

Now that we've briefly examined engineering design, we will equally briefly examine (the most relevant aspects of) Artificial Intelligence (AI).

Problem-solving as search. The late 1950s and the 1960s saw the development of the *search paradigm* within the field of Artificial Intelligence. Books such as "Computers and Thought" [10], which appeared in 1963, were full of descriptions of various weak methods whose power lay in being able to view the solving of a particular kind of problem as search of a space. In the late 1960s, the notion of heuristic search was developed, to account for the need to search large spaces effectively.

Knowledge as power. Nonetheless, most of the problems considered in those early days were what are now commonly called "toy problems". As the 1970s began, many practitioners in the field were concerned that the weak methods, though *general*, would never be *powerful* enough to solve real problems (e.g., medical diagnosis or computer configuration) effectively; the search spaces would just be too large. Their main criticisms of the earlier work were that solving the toy examples required relatively little knowledge about the domain, and that the weak methods required knowledge to be used in very restrictive and often very weak ways. (For example, in state space search, if knowledge about the domain is to be used, it must be expressed as either operators or evaluation functions, or else in the choice of the state space representation.) Solving real problems requires extensive knowledge. The "weak method" critics took an engineering approach, being primarily concerned with *acquiring* all the relevant knowledge and *engineering* it into some usable form. Less emphasis was placed on conforming the final program to fit some general problem-solving schema (e.g., heuristic search); more concern was placed on just getting a system that worked, and moreover, that would produce (measurably) "expert level" results. Thus was born the "expert systems" paradigm.

Evolution of new programming paradigms. Several list-processing languages were developed in the late 1950s and early 1960s, most notably, LISP. The

simple correspondence between searching a space for an acceptable solution and picking an appropriate item in a list made the marriage of AI (as it was at that time) and list-processing languages a natural one. Various dialects of LISP evolved, and the developers of the main dialects began evolving programming environments whose features made LISP programming more user-friendly (e.g., procedural enrichments of a language that was originally purely functional; debuggers; file packages; windows, menus, and list structure editors).

At the same time as the "expert systems" paradigm was developing, a new wave of programming languages (often referred to as "AI languages") was arriving. Like the evolution of expert systems, this development phase seemed to be motivated by the need for less general (but more powerful) languages than LISP. Many of these languages were (part of) Ph.D. theses (e.g., MICROPLANNER [42, 47] and Guy Steele's constraint language [35]). Often these languages were built on top of the LISP language, a possibility greatly facilitated because of the way LISP uniformally represents both data and process as lists. Often these languages were never used in anything but the Ph.D. dissertation for which they were developed, because they were overly specialized or they were not portable.

Exploring tradeoffs in generality and power. During the 1970s, at the same time as many researchers were swinging to the "power" end of the "generality-power" tradeoff curve in their explorations, others were striking a middle ground. Some researchers, realizing the limitations of the weak methods, began enriching the set of general building blocks out of which search algorithms could be configured. New component types included: constraint reasoning subsystems, belief revision subsystems, libraries or knowledge bases of various kinds; a variety of strategies for controlling problem-solving, etc. Other programming language designers than those mentioned previously developed new, specialized (but not overly specialized), and portable programming paradigms, including logic programming languages, frame-based and object-oriented languages, and rule-based languages. Rule-based languages such as OPS5 arrived on the scene at an opportune moment. In many cases, their marriage to "expert systems" seemed to work well, because the knowledge acquired from observing the behavior of domain experts often took the simple associational (stimulus-response) form: "IF the problem is of type P, then the solution is of type S."

Synthesis, consolidation and formalization. AI researchers of the late 1950s and the 1960s posed the *thesis*, "Generality is sufficient for problem-solving." 1970s researchers criticized this thesis, claiming the resulting methods were insufficient for solving real problems, and responded with the *antithesis*, "Power is sufficient." However, that antithesis has been critiqued in turn: "Expert systems are too brittle"; "special languages only work for the application for which they were originally developed"; etc.

Since the early 1980s, AI seems to be in a period of *synthesis*. One useful tool for illustrating the kind of synthetic framework that seems to be emerging out of the last few decades of research is depicted in Figure 1-1. Rather than pitting generality against power, or the "search paradigm" against the "expert systems" or "knowledge-based paradigm", the framework unifies by providing three different levels of abstraction for viewing the same "knowledge-based system": the knowledge level; the algorithm level; and the program level.

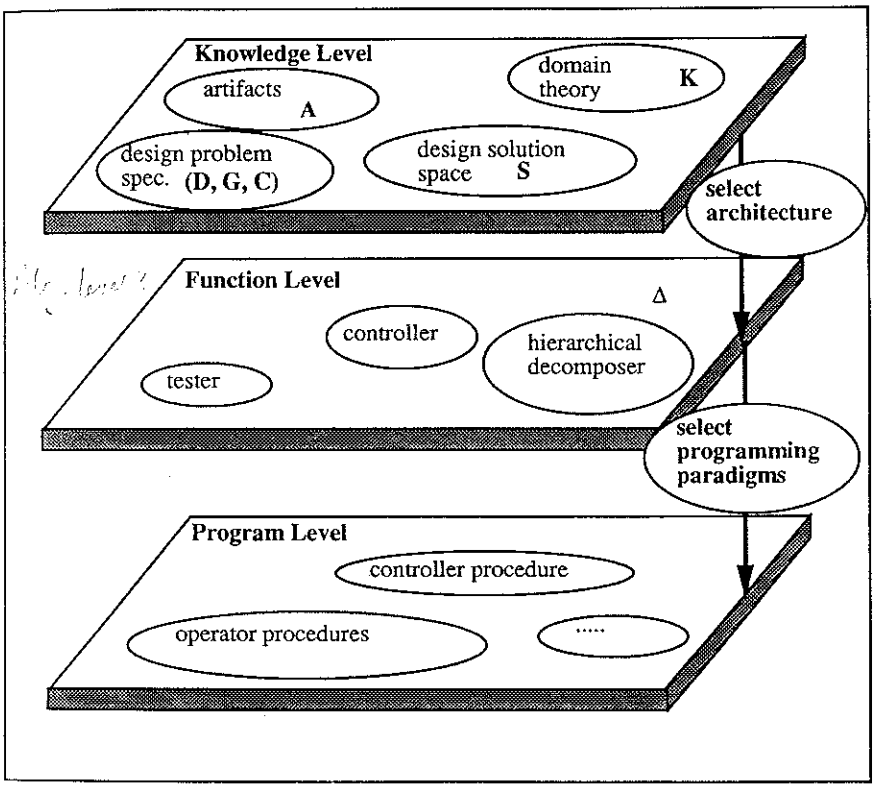

Figure 1-1: Rationally Reconstructed Knowledge-Based System Development

These three levels directly reflect the history of AI as we have just rationally reconstructed it. The "knowledge level" view of a knowledge-based system describes the knowledge that is used by and embedded in that system. The "algorithm level" view describes the system as a search algorithm, configured out of standard component types (e.g., generators, testers, patchers, constraint

propagators, belief revisers, etc.). Finally the "program level" view expresses the system in terms of the elements of existing programming paradigms (rules, objects, procedures, etc.). Within the "algorithm level", a spectrum of search algorithms -- ranging from weak to strong methods -- can be created depending on the choice of component configuration, and the choice of how knowledge (at the knowledge level) is mapped into the search algorithm components. A similar set of choices exists relative to the mapping of the "algorithm level" search algorithms into "program level" knowledge-based systems.

Many of the ideas and insights of this period of synthesis can be viewed as either: stressing the importance of distinguishing these levels (e.g., [6]); introducing criteria for evaluating systems at the different levels (e.g., epistemological adequacy [17] at the knowledge level; (qualitative) heuristic adequacy [17] at the algorithm level; and (quantitative) heuristic adequacy at the program level); fleshing out the primitives at each level (e.g., ATMSs [7] or constraint propagators [36] at the algorithm level); understanding and validating established correspondences between entities at different levels (e.g., between search algorithms and list-processing languages; or expert knowledge and rule-based languages), or on discovering new correspondences.

AI as a software engineering methodology. Viewed as a software engineering methodology, AI works best for developing those knowledge-based systems whose construction is usefully aided by creating explicit knowledge level and function level abstractions. More specifically, the AI methodology works well when:

- the problems addressed by the desired knowledge-based system are ill-structured, and involve large or diverse types of knowledge (when expressed at the knowledge level);
- that knowledge can be incorporated into an *efficient* search algorithm, that can be viewed as a configuration of standard building blocks for search algorithms;
- that search algorithm, in turn, can be implemented as an *efficient* program, using currently available programming paradigms.

1.2.3. Computer-aided Design

1.2.3.1. Opportunities for AI in computer-aided design

In many design areas (e.g., VLSI design or mechanical design), progress in automating the design process passes through a sequence (or partial ordering) of somewhat predictable stages (see Table 1-1). As we see it, design tool developers proceed through the following stages: permitting design capture; automating specific expert tasks; constructing unifying representations and system architectures; modeling and automating the complete design process; automatically controlling the design process; automatically re-using design experience; automatically improving tool performance. The central intuition is that, with the passage of time, design tools play an increasingly more *active* role in the design process. Note that the sequence is not meant to imply that the user is (or should ever be) removed from the design process; instead, the user receives increasingly greater assistance (and a more cooperative and powerful design environment) with time. Table 1-2 lists some particular technological contributions that have been made to design automation by academia and by industry.

Permitting design capture. In the beginning, graphical editors are created that allow the user to enter, visualize, modify, and store new designs, as well as retrieve old designs, in a simple manner. This is such a universal starting point for design automation in any new area that "CAD/CAM" (Computer-Aided Design/Computer-Aided Manufacturing) tends to be used as a synonym for fancy graphical, object-oriented interfaces. The development of these tools is largely aided by techniques drawn from graphics and database management (including such AI-related areas as deductive or object-oriented databases).

Automating the execution of expert tasks. As time passes, tool users become less satisfied with a purely passive use of CAD. CAD tool builders identify specific *analysis* and *synthesis* tasks which have been carefully delimited so as to be automatically executable (e.g., placement, routing, simulation). AI research can make a contribution at this stage; the software engineering methodology mentioned in Section 1.2.2 can facilitate the incremental creation, testing, and development of knowledge-based systems which carry out the more ill-structured analysis and synthesis tasks. (Well-structured tasks are carried out by algorithms.)

Constructing unifying representations and system architectures. A problem of integration arises; the set of available CAD tools is growing increasingly richer, but the tools are diverse, as are the design representation languages they

Table 1-1: Stages in the Evolution of Design Automation

DESIGN AUTOMATION GOAL	PROBLEM	AI ISSUE
Permit design capture	What functions does the user interface provide?	Deductive or object-oriented databases
Build tools for specific tasks	How to automate specialized types of reasoning?	Inference; Expert systems
Integrate tools	How to communicate between tools?	Representation; Architectures
Manage versions	Which task, tool, parameters?	Search space
Model design process	Which model is right for the task?	Taxonomy of tasks and corresponding methods
Find good design fast	How to guide choices?	Control
Improve design system	Where and how to improve?	Machine learning
Reuse design knowledge	How to acquire? How to re-use?	Machine learning, Case-based reasoning

utilize. AI can enter again to contribute ideas about unifying representation languages (e.g., object-oriented languages) that enable the creation of "design toolboxes", and unifying system architectures (e.g., blackboard architectures).

Modeling the design process. Having a single unified environment is good but not sufficient. How can we guarantee that we are making the most of our available tools? AI contributes the notion of the design process as a search through a space of alternative designs; the synthesis tools are used to help generate new points in this space; the analysis tools are used to evaluate the consistency, correctness, and quality of these points; the idea of search is used to guarantee that *systematic progress* is made in the use and re-use of the tools to generate new designs or design versions.

Table 1-2: Increasingly More Sophisticated Technological Contributions From Industry and Academia

Technology	University	Industry	Design Automation Goal
Interactive graphics	Sketchpad (MIT, 1963)	DAC-1 (GM, early 60s)	design capture
Drafting (2D)		Autocad™ ADE™	design capture
Solid modelers (3D) (CSG, BREP)	BUILD (UK) PADL (Rochester) (see [29])	I-IDEAS™ ACIS™	design capture
		MicroStation™	+ specific tools etc.
Solid modelers (super-quadrics, nonmanifold)	ThingWorld [28] Noodles (CMU)		design capture
Physical modelers (spatial + physics)	ThingWorld		design capture + specific tools
Parametric modelers (variational geometry + constraint management)	Work at MIT-CAD Lab PADL-2 (U. Rochester)	DesignView™ (2D) ICONEX™ (2D) PRO/ENGINEER™ (3D) Vellum™	design capture + specific tools
Semantic modeling + geometry (mostly wire frame) + constraint management + layout		ICAD™ WISDOM™ DESIGN++™	design capture + specific tools
Logic synthesis (ECAD) [18, 27]		Logic Synthesizer™	design process model (algorithmic)
Concept generators (routine design)	VEXED DSPL CONGEN	PRIDE (in-house)	design process model
Concept generators (innovative design)	BOGART CADET EDISON KRITIK ALADIN DONTE etc.	ARGO (in-house)	design process model + control
Integrated frameworks (cooperative product development [33])	DICE (MIT, WVU) PACT (Stanford) IBDE (CMU)	PACT (HP, EIT, Lockheed) Falcon™	integrate tools, version management

Controlling the design process. The priced paid for search is efficiency, as the search space is generally quite large. Exhaustive search of the space is usually intractable; however, a search which focuses its attention on restricted but promising *subspaces* of the complete design space may trade away the guarantee of an optimal solution (provided by exhaustive search), in return for an exponential decrease in overall design time.

How can a knowledge-based system control its search of a large design space so that a satisfactory solution is produced in a reasonable amount of time? Good *control heuristics* help.

Control heuristics may either be domain-specific or domain-independent. "Spend much of the available design time optimizing the component that is a bottleneck with respect to the most tightly restricted resource" is an example of a domain-independent heuristic, while "Spend much of the available design time optimizing the datapath" is a domain-specific version of this heuristic that applies to certain situations in the microprocessor design domain. Control heuristics may address different control questions. Some address the question: "Which area of the design should be worked on next?" while others address the question, "What should I do there? How should I refine that area of the design?"

Automatically improving performance and automated reuse of design experience. At this stage in the evolution of design automation in a design area, most of the burden of routine design has been lifted from the end user; this has been accomplished by *reformulating* this burden as one for the knowledge engineers and system programmers. In turn, techniques from machine learning can make life easier for the system builders themselves. In particular, they can build a design tool that is *incomplete* or *inefficient*; the design tool can be augmented by machine learning and case-based reasoning techniques that can extend the knowledge in the system, or use its knowledge with ever greater efficacy.

1.2.3.2. The differing goals of CAD tool and AI researchers

A misunderstanding frequently arises between AI researchers who develop experimental Computer-aided Design (CAD) tools, and traditional CAD tool developers in a particular design area (e.g., VLSI or mechanical design) who specialize in developing new design tools that will be usable in production mode in the near-term future. The CAD tool developers accuse the AI researchers of being too general, and of creating inefficient or toy knowledge-based systems. On the other hand, the AI researchers criticize the traditional CAD tool researchers of creating overly brittle systems.

Confusion arises because these two types of researchers (each of whom is likely to be reading this book) do not share quite the same research goals, and

each tends to judge the other with respect to their own community's values. Traditional CAD tool developers seek to reduce the effort in creating *new designs*. Most AI researchers aim at reducing the effort in developing *new design tools*.

Both research programs are worthy enterprises. The former goal requires the design tools to be powerful. The latter requires the methodology for constructing the tool (e.g., instantiation of a particular shell) to be general, and thus sometimes requires the design tool itself to be an instance of a general form rather than a custom-built tool. This book describes results from both enterprises.

1.2.4. A Methodology for Building a Knowledge-based Design Tool

In Section 1.2.1, we described the problem of design, and mentioned features of the problem that indicate design is generally an *ill-structured problem*. We then described AI as a three-level, software engineering methodology for developing knowledge-based systems for solving ill-structured problems. In the last section, we identified specific design automation tasks where such a methodology can be most usefully applied. We now describe what the general methodology looks like, when restricted to building knowledge-based design systems.

The steps involved in the development of AI tools for engineering design are shown in Table 1-3. The rest of this chapter will go into these steps in greater detail. We indicate which levels are involved in each step (knowledge, function, or program level), and which sections of this chapter will elaborate on that step.

The next few sections flesh out basic ideas relevant to understanding the phases of this methodology. They also relate the later chapters of this book to this methodology.

1.3. FORMALIZING THE DESIGN SPACE AND THE DESIGN KNOWLEDGE BASE

Algorithms can be decomposed into *passive* data structures and *active* access and update operations on these data structures. Similarly, models of design can be partitioned into passive components -- the design space and the knowledge base; and an active component -- the process that (efficiently) navigates through that space, guided by the knowledge in the knowledge base. This section

Table 1-3: Phases of Knowledge-based Tool Construction

PHASE	LEVEL	SECTION
Identify design task	knowledge level	1.5.1
Formalize design space	algorithm level	1.3
Formalize knowledge base	algorithm level	1.3
Configure appropriate model of design process, based on properties of design task and design space	algorithm level, knowledge level	1.4, 1.5.2
Instantiate by acquiring and operationalizing knowledge	knowledge level, algorithm level	1.5.2
Implement	algorithm level, program level	1.5.3
Test (validate and verify)	all levels	covered in individual chapters
Deploy		covered in individual chapters
Improve	all levels	covered in individual chapters

focuses on the nature and organization of design spaces and design knowledge bases, while the next section explores the spectrum of design processes that search such a space.

1.3.1. What Distinguishes a Design Search Space?

In order to characterize a (dynamically generated) search space, we must define the nature of the points in that space, the relationships that can exist between points in the space, and how to generate new points from old ones.

Points in the design space. In a *design space*, the points can be design

specifications[1] or implementations. They can be at varying levels of abstraction. Some points may only correspond to parts of a design (specification or implementation). A single such point P1 might have associated with it:

- its parts: {P11,...,P1n}. In the simplest case, these parts are simple parameters; in general, they can be arbitrarily complex structures.
- constraints on it and its parts.
- information about how its parts are connected.

Chapter 3 in Volume I considers the case where a design can be represented as a *constraint graph*, whose nodes are parameters, and whose arcs represent constraint relationships. Several design operations are easy to implement (in a domain-independent manner), given such a representation: automatic generation of parameter dependencies; evaluation of a constraint network; detection of over- and under-constrained systems of constraints, and the identification and correction of redundant and conflicting constraints. A few commercial tools, such as Cognition's MCAE™ and DesignView™ (see Volume III, Section 4.3.1), incorporate variations of Serrano's constraint management system. Chapter 4 in Volume I goes on to discuss how such a constraint network representation can be used to design automobile transmissions. The application of interval calculus methods to constraint satisfaction problems is treated in Volume I, Chapter 5. These interval methods are used in a mechanical design compiler, which accepts generic schematics and specifications for a wide variety of designs, and returns catalog numbers for optimal implementations of the schematics.

The design space as a whole. Some of the most basic relationships that can exist between points in the design space include:

- P2 is a part of P1.
- P2 is a refinement of P1 (where P1 and P2 are both specifications). P2 consequently may be at a lower level of abstraction than P1.
- P2 is an implementation of P1 (where P1 is a specification for and P2 is a description of an artifact in the target technology).

---

[1]We use the word *specification* to denote a *function* or a *goal* that needs to be realized or satisficed in the final design, e.g., "Design a land vehicle capable of going at least 100 mph over sand."

- P2 is an optimization of P1 (i.e., P2 is better than P1 with respect to some evaluatable criterion).
- P2 is a patch of P1 (i.e., P1 contains a constraint violation that P2 does not).

These points can also be clustered into multiple levels of abstraction; for example, in VLSI design, there might be a system level, a logic level, and a geometric layout level. Figure 1-2 illustrates some of these relationships.

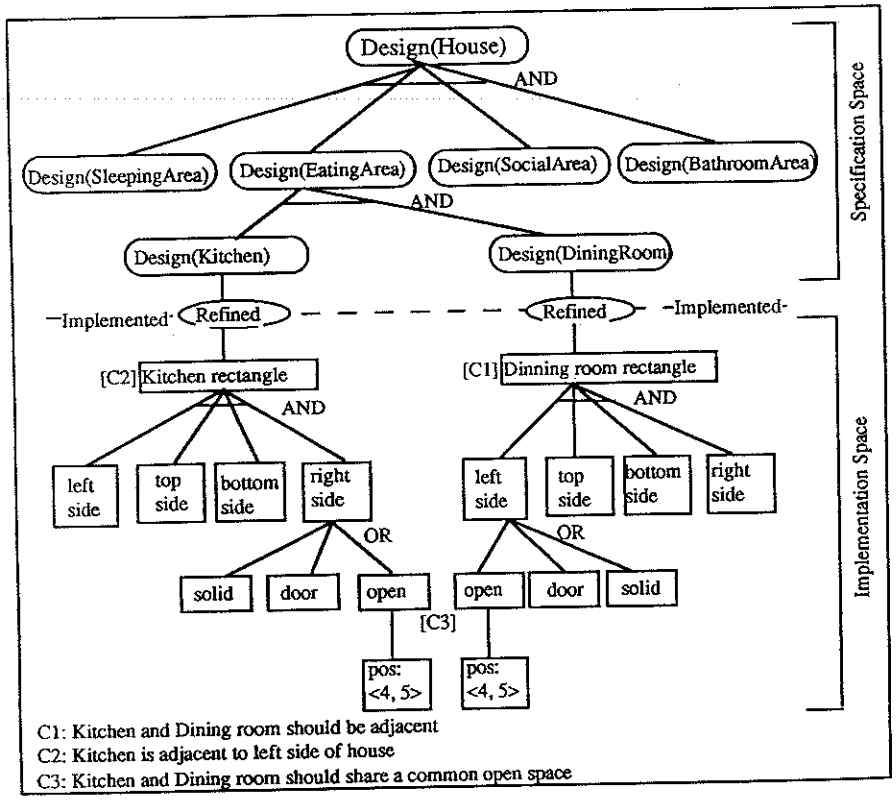

Figure 1-2: The Design Space as an AND/OR Tree

but accepting but slightly narrower view of design space

Dynamically generating the design space. Some of the most basic operations for generating new points in the design space from old ones include:

- refining P1 into P2.
- implementing P1 as P2 in target technology T.
- decomposing P1 into {P11,...,P1n}.
- optimizing P1 into P2 with respect to criteria O.
- patching constraint violation V in P1, yielding P2.

Chapter 2 in Volume I discusses the issues involved in representing all these aspects of a design space. The points are illustrated in the context of VLSI design.

1.3.2. What Distinguishes a Design Knowledge Base?

Often the parts that occur in designs (at any level of abstraction) can be viewed as *instances* of a generic class. For example, microprocessors are usually composed of generic parts such as ALUs, registers, busses, etc.

Such regularity can be exploited by maintaining a knowledge base of *design object classes*, and then viewing designs as configurations of instances of particular classes (e.g., a new microprocessor instance is constructed by creating an instance of ALU5, Datapath3, Bus4, etc. and then connecting these object instances together in a particular fashion). Design objects are also often *parameterized*. A complete instance of such a parameterized object class is created by assigning values to all the parameters.

In the standard object-oriented fashion, such design object classes may be organized hierarchically, thus reaping the standard benefits of inheritance. Design process operations (such as refinement, optimization, etc.) may also be indexed in a class-specific fashion (as methods), and thus, may also be inheritable.

The relation between a design space, a design knowledge base (of the kind just described), and a design process is as follows. A *design process* operation such as refinement, patching, or optimization may generate a new point in the *design space* from one or more old ones; the operation itself may involve creating new instances of design object classes from the *design knowledge base*.

Based on such an object-oriented representation of a design knowledge base, Chapter 2 (Volume I) discusses how to represent parameterized designs, design histories, and task-specific experts. As examples of desirable properties for design representations, it suggests modularity, compactness, flexibility permitted in the design process (e.g. in allowing both top-down and bottom-up design, and concurrent execution of design tasks), and extensibility; it describes how these properties may be achieved.

How does the design process know which design object class(es) should be instantiated to carry out a particular design operation (e.g., refinement of part P1)? One answer is to hardcode the association. For example, a specific *refinement rule* might express the knowledge that whenever a part of type P1 is being refined, it should be decomposed into parts of type {P11,...,P1n}. Or a specific *patching rule* might fix a specific type of constraint violation that commonly occurs in a specific kind of object. The design process models in Part II of this book take this hardcoded approach.

Another answer is to treat this question as a problem that must be solved explicitly by the design process. For example, the process of patching a constraint violation might actually involve solving the problem of *recognizing* that a particular object in the design is an instance of (or similar to) some object in the knowledge base, and then *recognizing* that the specified function of that object has been disabled in some way (by the context of the object). Available patching methods associated with that object class can then be applied (or adapted). Chapter 6 (Volume I) discusses how to organize a design knowledge base so that this kind of "innovative" patching can occur.

1.4. MODELS OF THE DESIGN PROCESS

1.4.1. The Nature of Design Tasks

1.4.1.1. Design task dimensions

Design tasks can be classified along several dimensions, including:

- available methods and knowledge;
- amount of unspecified (physical) structure;
- gap in abstraction levels between specification and implementation;
- complexity of interactions between subproblems; and
- amount and type of knowledge a system user can provide.

Available methods and knowledge. Is an appropriate method and/or sufficient knowledge always available for choosing what task to address next in the design process (e.g., what part to refine, what bug to fix, etc.)? Is knowledge or a method available for executing that next task? If there is more than one way of executing that task, is knowledge or a method available for selecting the alter-

native that will have the (globally) best outcome? The more (appropriate) knowledge and methods are available, the more *routine* the design task is. We will focus our discussion on two basic types of knowledge and methods: *generative* knowledge and methods, for generating new points in the design space; and *control* knowledge and methods, for helping the design process to converge efficiently on an acceptable design solution.

If sufficient knowledge and methods are available for always *directly* (i.e., without problem-solving) generating the next point in the design space and for converging on an acceptable design with little or no search, we will call the task a *routine* design task.

If the available knowledge and methods do allow for fairly rapid generation of an acceptable solution, but only by:

- *indirect* generation of new points in the design space -- i.e., finding a way to generate the next point in the design space involves a problem-solving process; and/or
- *indirect* control of the search, i.e., via problem-solving.

that is -- by itself, the available (directly applicable) knowledge generates unacceptable designs -- we will call the task an *innovative* design task.

Finally, if a problem-solving process is required to construct the design space in the first place, or if the best method available (given our current understanding) is an unguided search through a very large space, we will call the task a *creative* design task.

We will call design process models capable of handling these three types of design tasks routine, innovative, and creative design process models, respectively. We discuss routine design processes in Section 1.4.2, and innovative design processes in Section 1.4.3. We feel that creative design models, such as they are, are still at too preliminary a stage to be included here. However, [11] contains an interesting collection of workshop papers on this subject. Since we have tied creative design to the creation of the proper design space, creative design can also be viewed as a search through a space of design space representations, and thus work on problem reformulation and representation design can be seen as relevant here (see, e.g., [1]).

The terms "routine", "innovative", and "creative design" were introduced in [3], but were used in a somewhat different sense. Note that we use these terms in reference to the *task* and the *process*, but not the *product*. Thus, an innovative design process (e.g., replay of design plans) might not necessarily produce a product that is innovative with respect to the current market.

Amount of unspecified structure. Design maps function into (physical) structure. A design task often provides part of the (physical) structure of the design.

Since the design process involves creating a complete (physical) structure, it is also useful to identify what of the physical structure is left to be determined as a measure of design task complexity [39]. Design tasks are usefully distinguished according to what the *unspecified* structure looks like [40].

In *structure synthesis tasks*, the unspecified structure could potentially be any composition of primitive parts, which may not exist in the knowledge/data base. For example, the specified function might be a boolean function such as `(and (or x y) (not z))`. The physical structure might be any gate network that implements the boolean function; no part of the gate network is given *a priori*.

In *structure configuration tasks*, the unspecified structure is a configuration of parts of pre-determined type, and connectors of pre-determined type. For example, the physical structure might be a house floorplan containing some number of rooms, that can be connected by doors. For a particular floorplanning problem, the number of rooms and the size of the house might be given. In this case, the unspecified structure would be the configuration of rooms and doors, plus the values for room and door parameters.

Finally, in *parameter instantiation tasks*, the unspecified structure is the set of values for the parameters of each part. For example, the physical structure might be the part decomposition for all air cylinders (Volume I, Chapter 7). For a particular air cylinder design problem, the values for particular parameters (e.g., the length of the cylinder) might be given. Then the unspecified structure would be the values for all the remaining parameters.

Gap in abstraction levels between specification and implementation. In the simplest case, the design specification and the design implementation are at the same level of abstraction. This occurs, for example, when the only unspecified structure is parameter values. In other cases, a single level separates the functional specification from the target implementation level. That is, knowledge and methods are available for directly mapping the pieces of the specification into implementations; implementing a boolean function as a gate network is a simple example. In the worst case, the design may have to be driven down through several levels of abstraction before it is completed. For instance, in VLSI design, the initial specification might be of a digital system (e.g., a calculator or a microprocessor), which is first refined into a "logic level" description (a gate network), and then into a "layout level" description (of the actual geometry of the chip).

Complexity of interactions between subproblems. On one extreme (independent subproblems), the subproblems can all be solved independently, the solutions can be composed easily, resulting in an acceptable global design. On the other extreme, the subproblems strongly interact: a special (relatively rare) combination of solutions to the subproblems is required, and combining these solu-

tions into an acceptable global solution may not be easy or quick. Complexity increases when the number of interactions increases or when the type of interaction becomes more complex.

Two major types of design interactions are worth distinguishing. *Compositional interactions* arise when not all choice combinations (for refining or implementing the different parts of the design) are (syntactically) composable. For example, in VLSI design, the output of one part may be "serial", while the input of another may be "parallel"; if the output of the one must feed the input of the other, then the parts are not syntactically composable. Syntactic interactions may be further subdivided into *functional interactions* index(Functional interactions) among parts of a functional decomposition (e.g., in VLSI design, the "serial output/input" interaction) and *physical interactions* among parts of the implementation (e.g., in VLSI design, wire1 and wire3 on the chip must be at least 3 lambda units apart).

Resource interactions arise when different choice combinations lead to different overall usage of one or more global resources (e.g., delay time, power, or area in VLSI design). Different resources "compose" in different ways: e.g., global part counts are related to local part counts by simple addition; global delay time involves critical path analysis; etc.

Each interaction can be represented by a *constraint*. A *local* constraint only constrains a single part; a *semi-local* constraint constrains a relatively small number of parts; and a *global* constraint constrains a relatively large number of parts. Compositional interactions tend to be represented by semi-local constraints (because the syntax rules for correctly composing parts tend to refer to a small number of parts). Resource interactions tend to be represented by global constraints (since the global resource usage tends to be a function of the whole design).

Compositional interactions are typically *weak* interactions; they are usually representable by semi-local constraints. In contrast, resource interactions are typically *strong* interactions, representable by global constraints.

Amount and type of knowledge a system user can provide. In considering the nature of a design task, we will consider human users as knowledge sources, and thus classify the design tasks addressed by a particular knowledge-based design system as "routine" or "innovative" depending on how much knowledge (and method) the system and the user *together* can provide during the overall design process. Thus, even if the design system itself has no directly applicable control knowledge, if the user makes choices at every decision point in a manner that leads to rapid convergence on an acceptable solution, then the task is "routine".

1.4.1.2. Design task decomposition

While sometimes the terms we have just introduced are appropriately applied to the design task as a whole, it is often the case that "the design task" is a collection of (themselves sometimes decomposable) subtasks. Whether a task is considered a "routine design task" really depends on whether the subtasks are all routine and on how strongly the subtasks interact; the same design task may have relatively more and less routine parts to it. A category such as "parameter instantiation task" may be aptly applied to one subtask, and be inappropriate for another. Reference [5] makes some further points about task decomposition and associating different methods with different types of subtasks.

1.4.2. Models of Routine Design

1.4.2.1. Conventional routine design

In many cases, knowledge-based models of design are simply inappropriate, or would constitute overkill; conventional methods suffice for solving the task (or subtask). Some design tasks can be cast as a set of linear constraints C(s) on a set of real-valued variables, plus an objective function O(s) on these variables; for such problems, the methods of *linear programming* apply. Other simple design tasks can be cast as *constraint satisfaction problems* (CSPs) when: only parameter values are left unspecified; each parameter has a discrete, finite range of values; the constraints are unary or binary predicates on these parameters; and there are no optimization criteria. In such a case, the constraint satisfaction methods of [9] apply. Similarly, other types of design tasks are well-fitted to other standard methods (integer programming, multi-objective optimization techniques, AND/OR graph search [26], numerical analysis techniques, etc.). Many of these conventional methods have performance guarantees of various sorts: linear programming and AND/OR graph search are guaranteed to find a global optimum; if the constraint network is a tree, constraint satisfaction methods are guaranteed to run in polynomial time; etc.

1.4.2.2. Knowledge-based routine design

Viewed as a knowledge-based search, a routine design process is comprised of several different types of basic operations: refinement, constraint processing, patching and optimization. *Refinement* and *implementation* operations generate new, and less abstract points in the search space; *constraint processing*

operations prune inconsistent alternatives from consideration by the search; *patching* operations convert incorrect or sub-optimal designs into correct or more nearly optimal designs; *optimization* operations convert sub-optimal designs into designs that are more nearly optimal, with respect to some optimization criterion. Such operations might be stored as rules whose application requires pattern-matching (e.g., as in the VEXED system -- Volume I, Chapter 8); or as plans or procedures that are directly indexed by the type of design part to which they apply (e.g., as in the AIR-CYL system -- Volume I, Chapter 7).

1.4.2.3. Non-iterative, knowledge-based routine design

For some design tasks, sufficient knowledge or methods are available that a single pass (more or less) of top-down refinement -- possibly aided by constraint processing, patching, and directly applicable control knowledge -- is generally sufficient for converging on an acceptable design. This kind of design process model is demonstrated in several systems discussed in this book, including AIR-CYL (Volume I, Chapter 7) and VEXED (Volume I, Chapter 8). In the best case, applying this model requires running time linear in $p*l$, where p is the number of parts in the original specification, and l is the number of levels of abstraction through which each such part must be refined. However, constraint processing can slow things down, particularly if relatively global constraints are being processed [13].

1.4.2.4. Iterative, knowledge-based routine design

In other cases, the same kind of basic operations (refinement, constraint processing, etc.) are involved, but several (but not an exponential number of) iterations are generally required before an acceptable design is found. The need for iteration often arises when *multiple* constraints and objectives must be satisfied. A move in the design space that is good with respect to one constraint or objective may impair the satisfaction of another; tradeoffs may be necessary, and quickly finding a reasonable tradeoff (e.g., something close to a pareto-optimal solution) generally requires extensive domain-specific knowledge.

Several forms of iteration are possible:

- *Chronological backtracking.* A knowledge-poor method that is generally not acceptable for guaranteeing rapid convergence unless the density of solutions in the design space is very high, or the design space is very small. (Note, though, that "very small" need

not mean a space of tens of designs, but -- given the speed of modern-day computing -- could be one containing thousands of designs. See, e.g., Volume I, Chapter 4, where an acceptable design for an automobile transmission is found using chronological backtracking.

- *Knowledge-directed backtracking.* Dependency-directed backtracking possibly aided by advice or heuristics. PRIDE (Volume I, Chapter 9) and VT (Volume I, Chapter 11) both illustrate this kind of iteration.

- *Knowledge-directed hillclimbing.* Iterative optimization or patching of a design until all constraint violations have been repaired, and an acceptable tradeoff has been met among all global optimality criteria (e.g., area, power consumption, delay time, in VLSI design). The knowledge used to select among different possible modifications could be an evaluation function, or a set of domain-specific heuristics (CHIPPE, Volume I, Chapter 12), or the choice could be made by the user (DESIGNER, Volume I, Chapter 14).

- *Knowledge-directed problem re-structuring.* It is not only possible to change the design solution but also the design problem, e.g., by adding new constraints or objectives, or retracting or relaxing old ones. As the original problem poser, the user is often made responsible for such changes [BIOSEP (Volume I, Chapter 10) and WRIGHT (Volume I, Chapter 13)].

In the best case, applying this model requires running time *polynomial* in $p*l$, where p is the number of parts in the original specification, and l is the number of levels of abstraction through which each such part must be refined; i.e., the number of iterations is polynomial in $p*l$. In the worst case, the number of iterations is exponential because whatever knowledge is guiding the search turns out to be inadequate or inappropriate.

1.4.2.5. Routine design systems covered in this volume

Table 1-4 classifies along the dimensions we have been discussing the various routine design systems described in later chapters of this book. Notice that most of these routine design systems address design tasks involving parameter value assignment or structure configuration (but not "from scratch" synthesis of the entire structure).

Table 1-4: Categorization of Systems and Methods for Performing Routine Design

SYSTEM OR METHOD	DESIGN TASK	CHAPTER (VOL. I) OR PAPER	UNSPECIFIED STRUCTURE	DIRECTLY APPLICABLE KNOWLEDGE	SUBPROBLEM INTERACTIONS	ABSTRACTION LEVEL GAP
conventional optimization techniques	many simple tasks	--	parameter values	generative; control	algebraic constraints (global)	0
CSP methods	many simple tasks	Ref. [8]	parameter values	generative; some control	works best for semi-local constraints	0
AIR-CYL	air cylinders	7	parameter values	generative; patching	weak interactions	1
VT	elevators	11	parameter values	generative; knowledge-directed backtracking	strong interactions	0
PRIDE	copier paper paths	9	structure configuration	generative; knowledge-directed backtracking	works best for weak interactions	n
VEXED	circuits	8	entire structure	generative	weak interactions	n
BIOSEP	protein purification processes	10	structure configuration	generative	weak interactions + cost function	n
CHIPPE	VLSI	12	structure configuration	generative; knowledge-directed hillclimbing	weak interactions + global resource budgets	n
WRIGHT	spatial layouts	13	structure configuration	generative; user control	algebraic constraints + evaluation function	1
DESIGNER	graphic interfaces	14	structure configuration	generative; user control	mostly semi-local constraints	1

1.4.3. Models of Innovative Design

In innovative design tasks, routine design is not possible because of *missing design knowledge*. The missing knowledge might either be knowledge for directly generating new points in the design space, or knowledge for directly controlling the design space search. In this section, we will examine four different classes of innovative design. The first three focus (primarily) on missing generative knowledge, while the last deals with missing control knowledge:

- Innovation via case-based reasoning
- Innovation via structural mutation
- Innovation by combining multiple knowledge sources
- Search convergence by explicit planning of the design process

The first three approaches can be used to create innovative *designs*; the last approach involves creating innovative *design plans*, or innovative reformulations of the *design problem*.

1.4.3.1. Missing design knowledge

Why might relevant design knowledge be missing? One reason is that the most naturally acquirable knowledge might not necessarily be in a directly applicable form. This is often so in *case-based reasoning*; old designs and design process traces can be stored away fairly easily (if stored verbatim) in a case database, but then this leaves the problem of how to use these old cases to help solve a new design problem.

A second reason is that it generally is impossible to store the large amount of specific knowledge that would be necessary to adequately deal with all possible design variations (e.g., varying functional specifications, objective criteria, etc.). While some of this knowledge could be generalized, generalization often incurs a price of some sort; e.g., the generalized knowledge is not quite operational and must be made so at run-time; the (overly) generalized knowledge is not quite correct in all the circumstances to which it appears to be applicable; etc. Additionally, some of the knowledge simply is idiosyncractic, and thus not generalizable.

For this reason, deliberate engineering tradeoffs usually must be made in how much directly applicable design knowledge to build into the system, and how much to leave out, letting the system (or the user) cope with the missing knowledge.

A third reason is that human beings themselves may not have the relevant knowledge. Sometimes this is because the "structure to function" mapping is too complex to invert; methods may be available for analyzing the behavior and function of a given device, but not for taking a specified function and directly producing a structure that realizes that function. A case-based approach is often taken for such design tasks.

1.4.3.2. Case-based reasoning

Any case-based model of design must address the following issues:

- design case representation and organization
- design case storage
- design case retrieval
- design case adaptation and reuse

We will now say how three systems described in Volume II -- the BOGART circuit design system (Chapter 2), the ARGO circuit design system (Chapter 3), and the CADET system for designing fluid-mechanical devices (Chapter 4) -- handle these different issues. Chapter 5 (Volume II) analyzes case-based models of design in greater detail.

Design case representation. In BOGART, the stored cases are *design plans*, i.e., the steps used to incrementally refine a functional specification of a circuit into a pass transistor network are recorded *verbatim*. In ARGO, the same design session can yield several design cases, each at a different level of generality. Cases are stored as rules ("macrorules"), wherein the precise conditions for reuse of that case are stated explicitly. In CADET, each case involves four different representations: linguistic descriptions (i.e., <object attribute value> tuples); functional block diagramming; causal graphs; and configuration spaces.

Design case storage. In BOGART, the cases were automatically stored verbatim (when the user so chose) after a session with the VEXED design system (Volume I, Chapter 8). In ARGO, the design plan (a network of design steps and dependencies among them) is partitioned into levels. By dropping away more levels, more general plans are produced. Explanation-based generalization [19] of these design plans is used to determine the conditions under which each of these plans is applicable (which are then cached, along with the corresponding plans). In CADET, the cases were manually entered (since the focus of the CADET research was on case retrieval, and not case storage).

Design case retrieval. Because ARGO stores cases in such a way that the conditions for precise re-use are associated with them, retrieval of *applicable* cases is not an issue; ARGO uses a heuristic to restrict its retrieval to maximally specific cases. In BOGART, the *user* selects a case conceived as being similar to the current problem. In CADET, if no case directly matches the current specification, transformations are applied to the specification of device behavior until it resembles some case in the case database (e.g., some previously design artifact actually produces the desired behavior or something similar to it). In CADET, the specification may also be transformed in such a way that different parts of it correspond to different cases in the case database; all these cases are then retrieved (and the designs are composed).

Design case adaptation and reuse. In ARGO, reuse is trivial; a macrorule that matches is guaranteed to be directly applicable to the matching context. The transformations performed by CADET prior to retrieving a design permit direct use of the designs in the retrieved cases. In a case retrieved by BOGART (a design plan), some steps may apply to the current problem, while other parts may not; *replay* of the design plan is used to determine which steps apply. [23] is worth reading as a framework for case-based models of design such as BOGART, whose *modus operandi* is design plan replay.

Summary. BOGART's main innovation is in its method for design case reuse (via replay); ARGO's is in design case storage (macrorules with conditions of applicability); CADET's contribution is its method for design case retrieval (via transforming the design problem). All of these systems make contributions to the representation and organization of design cases that support their primary contribution.

1.4.3.3. Innovation via structural mutation and analysis

Most directly applicable knowledge for generating new points in the design space (either via refinement or modification) guarantees that something is being held invariant; most commonly, the functionality of the old design is preserved. If functionality-preserving transformations are not available, a weaker approach is to apply transformations that modify the artifact's (physical) structure in some manner, and then analyze the resulting functionality. Such analysis may then suggest further directions for modification until the desired functionality is (re)achieved. Such modifications are also guided by performance criteria and resource limitations.

One such approach is described in Volume II, Chapter 6. Here the problem is

to find a way to simplify a given, modular design (modular in that each structural part implements a different function) by identifying and exploiting structure-sharing opportunities (i.e., ways to make a given structure achieve multiple functions). Here the transformation for modifying the artifact's structure is one that deletes some part of the structure. After a part has been deleted (and hence a function has been unimplemented), other features of the remaining structure are identified that can be perturbed to achieve the currently unimplemented function (while not ceasing to achieve the function(s) they are already implementing). The identified features are then perturbed in the direction of better achieving the unimplemented function. For example, the handle of a mug could be safely deleted if the remaining cylinder were sized and shaped in such a way that it could be grasped by a human hand easily, and were made of a material that was heat-insulating (and hence would not burn the hand) -- e.g., a styrofoam cup. Essential to this approach is knowledge that associates changes in particular physical features of an artifact to the functions these (might) achieve.

If associations between (change of) physical structure and (change of) function are not hardcoded, then they may have to be derived. Qualitative modeling and reasoning of various kinds (e.g., qualitative simulation: see Volume II, Chapter 10) can sometimes be used to derive such associations.

1.4.3.4. Exploiting multiple knowledge sources

We have just described systems that use a case database to generate new designs, and other systems that use associations between structure and function to do the same. For some design tasks, multiple sources of (such indirectly usable) knowledge may be available, and all potentially useful; it might even be the case that solving the design problem *requires* integrating the advice of several knowledge sources.

Chapter 7 (Volume II) describes the ALADIN system, which helps design new aluminum alloys that meet specified properties. ALADIN draws on several sources of expertise to generate new points in the design space:

- a case database of previously designed alloys and their properties.
- if-then rules which associate structural changes (e.g., adding magnesium to the alloy) with functional changes (e.g., the value of the "strength" property increases).
- mathematical models of physical properties.
- statistical methods for interpolation and extrapolation.

1.4.3.5. Planning the design process

In a simple *routine design* scenario, the control questions that must be answered along the way take relatively simple forms: which part of the design to work on next? What to do there (refine, implement, optimize, patch)? Of several possible ways to do that, which to pick? Acquirable control knowledge may be sufficient for answering the control questions as they arise.

However, for several reasons, a design process model can be more complex, thus giving rise to new control questions, and hence to the need for a more complex controller:

- *More methods and knowledge sources.* Innovative design systems can involve a diverse range of activities and draw on many sources of knowledge. For example, the ALADIN system draws on multiple knowledge sources, and consequently must also answer new control questions: which knowledge source to consult next? How to combine the outputs of several knowledge sources? etc.
- *Multiple objectives.* Another source of control problems arises when multiple objectives must be satisfied. New control questions include: With respect to which objective should the design be improved next? Which part of the design should be redesigned to effect the improvement?
- *Expensive design operations.* Operations such as simulation (e.g., VLSI chip simulation) or analysis (e.g., finite element analysis) can be sufficiently costly that their use should be carefully planned.

A global view: Control as planning. To be operational, any control strategy must provide answers to specific, *local* control questions of the kind just described. However, the problem of control has a *global* goal in mind: Utilize knowledge and methods so as to most rapidly converge on an acceptable solution. Hence we can think of the problem of control as a *planning problem*: construct a relatively short design plan whose steps invoke various design methods and draw on design knowledge, and which, when completely executed, results in the creation of an acceptable design.

Stefik [36, 37] and Wilensky [45] gave the name *meta-planning* to this approach to control, since the design process itself is being explicitly represented and reasoned about. Stefik's MOLGEN system represented the design (a plan for a molecular genetics experiment) at multiple levels of abstraction. MOLGEN took a least commitment approach to refining the design through these levels of abstraction. It also used a multi-layered control strategy, explicitly

representing and modifying the design plan. The ALADIN system (Volume II, Chapter 7) uses a very similar approach to managing the navigation through its multiple spaces for designing aluminum alloys.

Control as top-down refinement of design plans. When design operations (such as VLSI simulation) are expensive, one response is to create abstractions of these operations and much more cheaply construct plans for the design process in the space of abstract operations, pick the best abstract plan, and then refine it into an actual design plan (one whose execution would produce complete designs, and accurate analyses). This approach can be viewed as a special kind of meta-planning in which the planning method is top-down refinement (often also called "hierarchical planning"). This approach has been applied to VLSI design in the ADAM system (Volume II, Chapter 8).

But what is the "best" abstract plan? In ADAM, "best" means the one which when executed, creates a design that comes closest to satisfying all of several resource limitations (on area, speed, power, and design time). ADAM uses a single weighted evaluation function of all the resource usages:

```
w1 * area + w2 * speed + w3 * power + w4 * design time

where w1+w2+w3+w4=1
```

to guide its search. ADAM first finds plans that construct designs which are optimal with respect to each of the individual resources; for instance, to do so for "area" would involve setting w1 = 1, and w2 = w3 = w4 =0. Based on the the difference between the costs of the resulting designs and the specified budgets, ADAM uses *linear interpolation* to readjust the weights on the evaluation function. It then replans.

Exploratory design: Control as hillclimbing in the space of problem formulations. The following hypothesis (we will call it the *routine design hypothesis*) is one way of viewing the relationship between an innovative design problem and a routine design problem:

> If the design problem is appropriately structured and contains enough detail (i.e., if we are "looking at the problem right"), then a single pass of a simple routine design process should produce an acceptable design (if one exists).

The control strategy we will next describe, called *exploratory design*, is appropriate for those problems where the initial design problem is *not* appropriately structured or annotated (i.e., it is an *innovative* design problem). We

call this "exploratory design" because our intuition is that human designers handle problems that are complex in novel ways by spending their initial time finding a good way to look at the problem.

Models of routine design involve a search purely in the space of designs. In exploratory design, the problem and the solution co-evolve. Exploratory design hillclimbs in the space of problem formulations (the "outer loop" of the method), getting feedback for adjusting the problem formulation from analyzing how the candidate designs generated so far (by the "inner loop" of routine design) fail to be acceptable.

The DONTE system (Volume II, Chapter 9) performs such hillclimbing in the space of circuit design problem formulations using top-down refinement, constraint processing, and design patching operations in its "inner loop". The kind of problem reformulation operations it performs there are: macro-decision formation, which imposes a hierarchical structure on a relatively flat problem decomposition; budgeting, which adds a new budget constraint to every design component; re-budgeting, which may adjust such constraints in several components; rough design, which assigns estimates of resource usage to various parts of the design; and criticality analysis which (re)assesses how (relatively) difficult the various subproblems are to solve (given their current budgets, etc.).

1.4.3.6. Innovative design systems covered in this volume

Table 1-5 classifies along the dimensions we discussed earlier the various innovative design systems described in later chapters of this book. Notice that most of these innovative design systems address design tasks involving synthesis of the entire structure.

1.4.4. Qualitative Reasoning about Artifacts during Design

The mapping of a knowledge level specification of a design system into am algorithm level search algorithm can draw on formally represented bodies of generally useful "common sense" knowledge and procedures relevant to reasoning about the physical artifacts being designed. We now describe two kinds of such knowledge: knowledge about physical systems; and knowledge about geometry. With respect to codification of "common sense" knowledge, the CYC project [14] represents an alternate and possibly complementary approach to those described here.

Table 1-5: Categorization of Systems and Methods for Performing Innovative Design

SYSTEM OR METHOD	DESIGN TASK	CHAPTER (VOL. II)	UNSPEC. STRUC.	ABSTR. LEVEL GAP	GENERATION PROBLEMS ADDRESSED	CONTROL PROBLEMS ADDRESSED	WHAT IS INNOVATIVE
BOGART	circuits	2	entire structure	1	how to replay retrieved case		design
ARGO	circuits	3	entire structure	1	how to store cases so generation is easy		design
CADET	fluid-mechanical devices	4	entire structure	n	how to identify similar cases		design
FUNCTION SHARING	fluid-mechanical devices	6	none	0	how to identify function-sharing possibilities		design
ALADIN	aluminum alloys	7	entire structure	n spaces	how to use multiple knowledge sources to generate new design		design
ADAM	VLSI	8	entire structure	n		how to find promising design plan	design plan
DONTE	circuits	9	entire structure	n		how to find good problem decomposition, budget allocation, resource usage estimations	design problem reformulation

1.4.4.1. Qualitative reasoning about physical systems during design

Functional specifications for *physical systems* often take the form of stipulating a particular relationship between behavioral parameters, e.g., the output rotation of a rotation transmitter must be 30 times as fast as the input rotation. It is rarely the case that a single part (e.g., a single gear pair) is capable of directly achieving the specified relationship. Instead, a series of interacting components may be needed. This is especially the case when the type of the behavioral parameter changes: e.g., the input is a rotational speed, but the output is a rate of

up-and-down movement. The network of interacting behavioral parameters may necessarily include feedback loops, e.g., when the specified relationship defines a self-regulating device (e.g., a change in one variable should result in a corresponding change in the other).

Williams has proposed a design process model for such problems called *interaction-based invention*:

> Invention involves constructing a topology of interactions that both produces the desired behavior and makes evident a topology of physical devices that implements those interactions [46].

One of the key steps in this process is verifying that the interactions in the constructed interaction topology actually "compose" to produce the specified interaction. Carrying out this step (and satisfying its representational needs, i.e., providing an adequate representation of the causal and temporal features of each interaction) is particularly difficult when the topology is complex (e.g., as in most circuits that contain feedback loops). Chapter 10 (Volume II) discusses how to adequately represent such interactions in complex physical systems (such as analog circuits with feedback loops), and how to qualitatively analyze the global behavior of these systems.

1.4.4.2. Qualitative reasoning about geometry in design

Geometry-constrained synthesis. Many design tasks involve geometry in one way or another in their functional specifications or domain knowledge. In the simplest of cases, the role geometry plays is purely static, placing restrictions on the boundaries of the artifact, points of attachment of parts of the artifact, etc. The WRIGHT system described in Chapter 13 (Volume II) handles a subclass of such spatial placement problems.

The synthesis of small load-bearing structures illustrates a more complex role of geometry: *forces* (i.e., the loads) are positioned at certain points in space; a single structure must be synthesized that is both stable and capable of bearing the loads (and that does not occupy any "obstacle" regions of space). Chapter 11 (Volume II) describes the MOSAIC system, which synthesizes such load-bearing structures using a design process model that performs problem abstraction, problem decomposition, and iterative re-design.

Another geometric complication shows up in *kinematic synthesis*, the synthesis of physical structures that *move* in ways that satisfies certain restrictions on motion in space. Chapter 12 (Volume II) considers the problem of designing linkages (e.g., door hinges, aircraft landing gear, cranes, etc.), given constraints on specific points through which the linkage must pass (perhaps in a particular order), number of straight line segments in the path of motion, etc. In the TLA system, the user selects a linkage from a case database of four-bar linkages,

looking for those that have features resembling the problem specifications. Optimization techniques are then used to adapt the known case to the current problem; user intervention helps such techniques avoid getting stuck in local minima.

Joskowicz (Volume II, Chapter 13) also describes an approach to kinematic synthesis. Mechanisms, retrieved from either a catalog or a case database, are considered during artifact redesign. Retrieved mechanisms should ideally be *kinematically equivalent* to the current design. Joskowicz describes a method for comparing two mechanisms for kinematic equivalence, that involves trying to find a common abstraction of both. This same mechanism comparison technique is used to organize the case database (for the purpose of efficient retrieval) into classes of kinematically equivalent mechanisms.

Geometry-based analysis. That designed artifacts have geometric features means that some of the analysis processes performed during design will involve geometric reasoning, including: static and dynamic analysis of stresses (based on shape), and kinematic simulation of mechanisms.

The conventional approach to analyzing stress is finite element analysis. However, this method requires a grid as an input, and which grid is best varies with the problem. In contrast, Chapter 14 (Volume II) describes an approach to stress analysis that geometrically partitions an object into regions in such a way that the object parts have shapes (e.g., a plate with a hole in it) resembling known cases (e.g., a plate without a hole in it). These known cases have associated (pre-computed) stress analyses, which are then used as part of the stress analysis data for the overall object.

One method for *kinematic simulation* is described in Chapter 13 (Volume II). First, local behaviors are computed from two-dimensional configuration spaces, defined by the objects' degrees of freedom. Global behaviors are then determined by composing pairwise local behaviors.

1.5. BUILDING A KNOWLEDGE-BASED DESIGN TOOL

The actual construction of a new knowledge-based design tool goes through three basic phases:

- Identify the design task
- Configure and instantiate the design process model

- Implement the design process model

1.5.1. Identifying the Design Task

Identifying the design task involves defining the task and classifying it.

1.5.1.1. Knowledge acquisition to define the design task

To define a design task, we must acquire knowledge defining:

- the class of *problems* that can be solved;
- the class of *candidate solutions* that contains a set of acceptable solutions to the problem;
- the *domain theory*, the body of domain-specific knowledge that is accessed in solving such problems, and constrains what is considered to be an acceptable solution.

How can such design knowledge be either easily acquired from domain experts, or otherwise automatically added to the knowledge base?

Graphical interfaces. Chapter 2 (Volume III) discusses the advantages of using graphical interfaces in acquiring design knowledge from experts. In particular, the knowledge is acquired in the form of decision trees. These trees are then mapped into expert rules in OPS5. The complete process is illustrated by acquiring and compiling knowledge from experts for bearing selection.

Knowledge acquisition for specific design process models. Another way to simplify knowledge acquisition is to tailor a particular knowledge acquisition method to a specific design model. For example, the SALT system (Volume I, Chapter 11) specializes in acquiring knowledge for a design system that iteratively modifies a design.

SALT first acquires a graph whose nodes are design inputs, design parameters, or design constraints and whose edges express various relationships between these. SALT then acquires three types of knowledge that are indexed off the graph nodes: knowledge for proposing a design extension (specifying a design parameter), knowledge for identifying a constraint, and knowledge for proposing a fix to a constraint violation. SALT has a schema for each type of

knowledge, and prompts the user with questions whose answers fill in the appropriate schema. SALT also has techniques for analyzing the completeness and consistency of the knowledge base. The SALT system was used to acquire the knowledge in the VT system.

Case-based reasoning. In Section 1.4.3.2, we described case-based reasoning as a particular model of innovative design. Because case-based reasoning involves storage of design cases from previous design system sessions, it represents another way of adding "new" knowledge to the knowledge base.

As mentioned previously, the stored knowledge can range in generality from design plans that are stored verbatim (as in the BOGART system, Volume II, Chapter 2), to automatically generalized knowledge (as in the ARGO system of Volume II, Chapter 3).

1.5.1.2. Classifying a design task

As mentioned earlier, design tasks can be classified along several dimensions, including:

- available methods and knowledge
- gap in abstraction levels between specification and implementation
- amount of unspecified (physical) structure
- complexity of interactions between subproblems; and
- amount and type of knowledge a system user can provide

1.5.2. Configuring and Instantiating the Design Process Model

Classification of a design task identifies important features of that task. Different features suggest different design process models. Tables 1-4 and 1-5 suggest, by example, some of the correspondences.

1.5.3. Implementing the Design Process Model

Once a design process model is determined, the next step is to map the design process model onto the program level (see Figure 1-1). "Maxims" pertinent to carrying out this mapping include:

1. Code in an appropriate programming language, such as C++, LISP, OPS5, KEE™. Most of the papers in Volume I and Volume II, as well as Chapter 7 in Volume III, take this approach.
2. Use a commercial tool that provides some support for design artifact representation; implement appropriate extensions. Chapters 3, 4, 5, and 6 in Volume III follow this path.
3. Develop a domain-independent shell that implements the design process model(s) and instantiate the shell for a particular application.
4. Use a knowledge compiler to generate special-purpose procedures for efficiently processing particular (and generally domain-specific) subtasks of the overall design task.

1.5.3.1. Commercially available tools

There are two kinds of tools available in the commercial market place for civil/mechanical engineering applications (see Table 1-2):

1. **Parametric modelers,** which provide constraint processing capabilities to geometric modelers. An application utilizing a parametric modeler (DesignView™) and a knowledge-based programming tool (NEXPERT™) for designing a product and forming sequence for cold forging is described in Chapter 4 (Volume III). We have included a list of commercial tool vendors in Appendix A at the end of this chapter.
2. **Design representation frameworks,** which provide additional layers over knowledge representation languages. Typically these layers support the following activities:

    - Representation of engineering entities, including composite objects;
    - Geometric modeling;

- Constraint management;
- Access to external programs, such as engineering databases;
- Programming language support (current tools are implemented in LISP); and
- Rule-based inferencing.

Applications implemented in three commercially available tools are described in Volume III, Chapters 3 (ICAD™), 4 (DesignView™ and NEXPERT Object™), 5 (Design++™), and 6 (Concept Modeller™).

1.5.3.2. Domain-independent shells

Domain-independent shells, in addition to representation and programming language support, provide design process models as problem solving strategies. Applications can be built by adding domain-specific knowledge. Many of the routine design systems described in Volume I have evolved into domain-independent shells. These systems view design as:

```
Hierarchical Refinement + Constraint Propagation + ..
```

and provide knowledge editing facilities for inputting design plans, goals, artifacts, and constraints. Table 1-6 summarizes several domain-independent shells, developed in the United States. Several organizations in other countries are attempting to build such tools, e.g., LEOSYS™ , developed by Olivetti Computers, Italy.

1.5.3.3. Knowledge compilers

In principle, *knowledge compilers* can be used to create (at compile time) those components of the design system that are not easily viewable as instantiations of domain-independent "shell" components, and that are not one of the commercially available tools (e.g., parametric modellers or design representation frameworks). Often the compiled components handle particular, domain-specific tasks such as maze routing [32], house floorplanning [44], or synthesis of gear chains [24]. It is also possible to use knowledge compilers to optimize components that originated as shell instantiations.

Some compilers are quite specialized; for example, the ELF system

Table 1-6: Domain-Independent Shells that Implement Hierarchical Refinement and Constraint Propagation

SHELL/ REFERENCE	PREDECESSOR/ DOMAIN	REP. LANGUAGE/ BASE LANG.	MACHINE OR OS	DEPARTMENT/ PLACE
DESCRIBE [20]	PRIDE Paper Handling	LOOPS LISP	XEROX	Only Inhouse
EDESYN [16]	HI-RISE Buildings	FRAMEKIT LISP	Unix	Civil Engrg. CMU
DSPL [4]	AIR-CYL Air Cylinders	LISP	Unix	Comp. Sci. OSU & WPI
EVEXED [38]	VEXED VLSI	STROBE LISP	XEROX	Comp. Sci. Rutgers
DIDS [2]	MICON Computers	C++ C	Unix	EECS Univ. Michigan
CONGEN [34]	ALL-RISE Buildings	C++ C	Unix	Civil Engrg. M.I.T.

[32] specializes in compiling global routers, for varying VLSI technologies. The KBSDE compiler [44] and the constraint compiler of the WRIGHT system (Volume I, Chapter 13) address a different and somewhat broader class of knowledge-based systems for spatial configuration tasks. The DIOGENES compiler [24] addresses the still broader class of heuristic search algorithms. These compilers appear to obey the standard power/generality tradeoff. The models of knowledge compilation also grow progressively weaker as the breadth widens, culminating in such weak (i.e., relatively unrestricted) models as: a transformational model of knowledge compilation [22] or a model of knowledge compilation as formal derivation.

All the compilers just mentioned are research prototypes, and are thus not commercially available. Nonetheless, we mention this technology because of its potential importance in the not too distant future. In the meantime, human programming skills will have to suffice.

1.6. DESIGN AS PART OF A LARGER ENGINEERING PROCESS

It is important to view design in the perspective of the overall engineering process, which involves several phases: market studies, conceptualization, research and development, design, manufacturing, testing, maintenance, and marketing (see Figure 1-3). In this process people from various disciplines interact to produce the product.

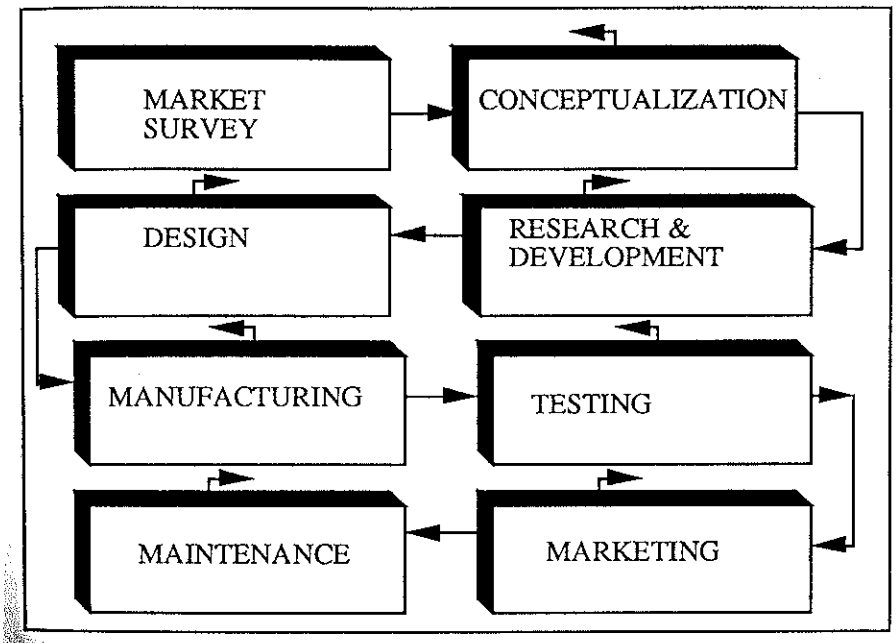

Figure 1-3: Engineering a Product
(Bent arrows indicate that the process is iterative)

In traditional product development, the lack of proper collaboration and integration between various engineering disciplines poses several problems, as expounded by the following Business Week (April 30, 1990, Page 111) clip [see Figure 1-4 for a typical scenario in the AEC industry].

> The present method of product development is like a relay race. The research or marketing department comes up with a product idea and hands it off to design. Design engineers craft a blueprint and a hand-built prototype. Then, they throw the design "over the wall" to manufacturing, where production engineers struggle to bring the blueprint to life. Often this proves so daunting that the blueprint has to be kicked back for revision, and the relay must be run again - and this can happen over and over. Once everything seems set, the purchasing department calls for bids on the necessary materials, parts, and factory equipment -- stuff that can take months or even years to get. Worst of all, a design glitch may turn up after all these wheels are in motion. Then, everything grinds to a halt until yet another so-called engineering change order is made.

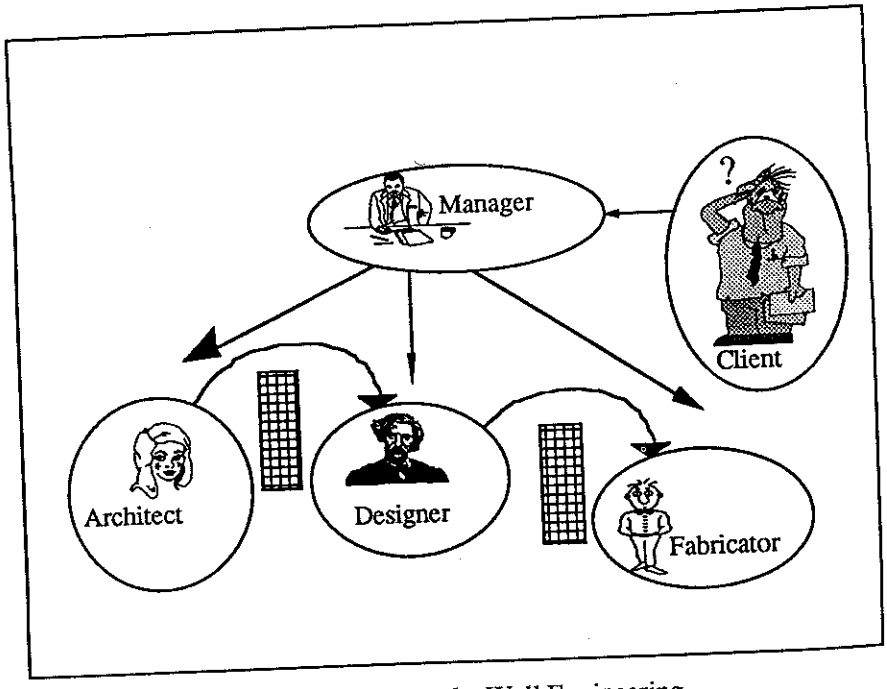

Figure 1-4: Over the Wall Engineering

Several companies have addressed the above problem by resorting to a more flexible methodology, which involves a collaborative effort during the entire life cycle of the product. It is claimed (Business Week, April 1990) that this approach[2] results in reduced development times, fewer engineering changes, and better overall quality. The importance of this approach has been recognized by the Department of Defense, which initiated a major effort -- the DARPA Initiative in Concurrent Engineering (DARPA DICE) -- with funding in the millions of dollars.

It is conceivable that the current cost trends in computer hardware will make it possible for every engineer to have access to a high performance engineering workstation in the near future. The "over the wall" approach will probably be replaced by a network of computers and users, as shown in Figure 1-5; in the figure we use the term *agent* to denote the combination of a human user and a computer.

The following is a list of issues that we consider important for computer-aided integrated and cooperative product development.

1. **Frameworks**, which deal with problem solving architectures.
2. **Organizational issues**, which investigate strategies for organizing engineering activities for effective utilization of computer-aided tools.
3. **Negotiation techniques**, which deal with conflict detection and resolution between various agents.
4. **Transaction management issues**, which deal with the interaction issues between the agents and the central communication medium.
5. **Design methods**, which deal with techniques utilized by individual agents.
6. **Visualization techniques**, which include user interfaces and physical modeling techniques.

Several papers in Volume III address some of the above issues; [33] contains additional papers in this area. Chapters 7 and 8, Volume III, discuss the DFMA and the ECMG frameworks, respectively, that bring manufacturability knowledge into the early design phases. The manufacturing knowledge is tightly integrated into the the design framework. The Engineous system, described in Volume III, Chapter 9, is a generic shell that combines knowledge-

[2] "Concurrent engineering", "collaborative product development", "cooperative product development", "integrated product development" and "simultaneous engineering" are different phrases used to connote this approach.

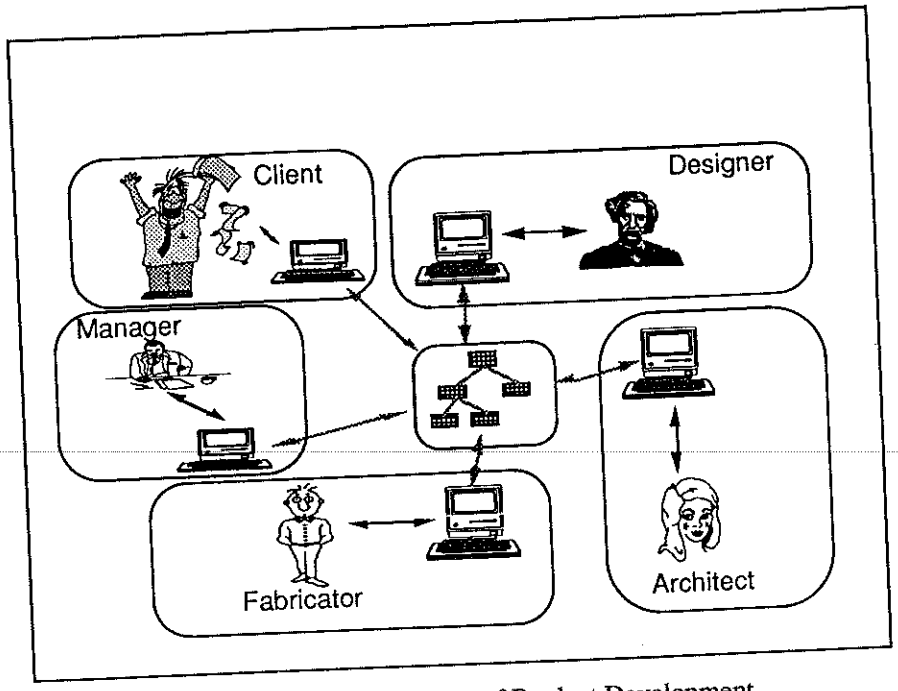

Figure 1-5: Modern View of Product Development

based expert systems, numerical optimization, and genetic algorithms for product design.

While the above systems are closely coupled architectures, the systems described in Chapters 10, 11, and 12 (Volume III) are loosely coupled and reflect the architecture shown in Figure 1-5. A multi-level and a multi-modal architecture, DMA, that supports easy integration of various design/manufacture CAD systems is proposed in Chapter 10 (Volume III). The design module supports an axiomatic approach to design [41]. The manufacture module contains manufactability knowledge, such as assembly sequencing, etc.

A dual design partner scheme is proposed in Chapter 11 (Volume III). This scheme supports two competing system behaviors. One expert machine -- the stabilizer -- resists change and always presents a conservative hypothetical model of the product. The other expert machine -- the innovator -- strives for well calculated and justified alternative hypothetical models of the product. The dual partner scheme is being implemented using the blackboard architecture [25].

The DICE project (Volume III, Chapter 12) implements a blackboard architecture over an object-oriented database management system; thus the blackboard and the object-store are tightly integrated. In addition, the objects in the blackboard have behavior associated with them. Hence, the need for a sophisticated scheduler -- as provided in the traditional blackboard systems -- is obviated. The DICE project also incorporates comprehensive transaction and version management mechanisms. The DICE version described in this volume was implemented in Common LISP. Other implementations also exist in the OPAL/GEMSTONE and C++/ONTOS environments.

Table 1-7 summarizes the various efforts in integrated design systems.

Table 1-7: Summary of Integrated Design Frameworks

SYSTEM	CHAPTER (VOL. III)	FEATURES	NO. LEVELS	STATUS
DFMA	7	Tightly coupled	1	In-house use
ECMG	8	Tightly coupled; Domain-independent	1	Commercially available
Engineous	9	Tightly coupled expert systems; genetic algorithms; optimization	1	In-house use
Dual Partner	11	Loosely coupled; Blackboard; database	n	Prototype
DMA	10	Loosely coupled	n	Prototype
DICE	12	Loosely coupled; Blackboard; object-oriented databases; negotiation; transaction management	n	Prototype

1.7. SUMMARY

In this overview chapter, we have presented a framework for helping to understand the field of "AI in Engineering Design" in general, and the papers in this collection, in particular.

Applying AI software engineering methodology to Engineering Design problems. We first considered "Engineering Design" and "Artificial Intelligence" as separate disciplines, the former providing special kinds of ill-structured problems, and the latter providing a methodology for developing knowledge-based systems that effectively solve certain types of ill-structured problems.

Design problems are *ill-structured* in that the mapping of desired functionality onto a (physical) structure that correctly implements it is generally not straightforward. Furthermore, most design problems call for not only a *correct* design but a *good* design -- good with respect to one or more (possibly ill-defined) metrics (e.g., cost, area, volume, power consumption, etc.); this further complicates the mapping, thereby decreasing the likelihood that a simple (polynomial time) algorithm will suffice for carrying out the mapping, and increasing the likelihood that some degree of search (e.g., generate-and-test) will be necessary. Finally, the design problem representation itself may begin its life as an ill-structured set of "requirements" and only gradually (enabled by feedback from actual design experience) evolve into a set of formal "specifications".

For the purposes of this book, we have described Artificial Intelligence as a discipline that provides a multi-level methodology for engineering knowledge-based problem-solving systems. In particular, a *knowledge level* specification of the system (and the class of problems it must solve) is mapped into an *algorithm level* description of an efficient search algorithm for efficiently and acceptably solving that class of problems. That (simulatable) algorithm description is then mapped into an actual piece of code at the *program level*, using one or more programming paradigms (e.g., procedural programming, rule-based programming, object-oriented programming), shells (e.g., VP-EXPERT^{TM}), or commercially available subsystems (e.g., an ATMS in KEE^{TM}). The application of AI to Engineering Design thus looks like a specialization of this software engineering methodology to: design tasks (specified at the "knowledge level"); design process models (described at the "algorithm level"); and design programs built from shells, commercially available design subsystems, and manually constructed code (implemented at the "program level").

Mapping a knowledge level specification for a design system into a algorithm-level search algorithm. In considering mapping a knowledge level specification for a design system into an algorithm-level search algorithm, it is

useful to decompose the algorithm into passive and active components. One passive component is the *design space* to be searched. The active design components are the various *functional components of the design process model* (e.g., refinement, hillclimbing, constraint propagation, backtracking, etc.), which, in effect, generate the design space and navigate through it. These active components draw upon another passive component, declaratively represented *design knowledge*, interpreting this knowledge at run time (e.g., to estimate the cost of a particular design, to choose between several design alternatives, etc.).

The same piece of knowledge can be embedded into an algorithm in a variety of ways, with varying degrees of effectiveness. The most effective way to map available design knowledge into the algorithm-level search algorithm is to carefully engineer the design space itself, so that it -- *a priori* -- will contain (when generated at run-time) as few incorrect or poor designs as possible. The next most effective way to use design knowledge is to *compile* it into the active components of the search algorithm (e.g., creating customized routines for efficiently performing special tasks such as routing, placement, estimation, simulation, etc.) The least effective (though sometimes easiest, and sometimes necessary) way to use design knowledge is to represent it declaratively (e.g., as is often the case in shells), and then *interpret* it at run time.

Other factors also come into consideration when mapping a knowledge level specification of a design system into an algorithm-level search algorithm. Design tasks can be categorized along various dimensions; different search algorithms will be appropriate for different types of design tasks. Useful dimensions for taxonomizing design tasks include: available methods and knowledge (addressing that task); gap in abstraction levels between specification and implementation; amount of unspecified (physical) structure; and amount and type of knowledge a system user can provide.

Of primary importance in distinguishing types of design tasks is the amount and types of available knowledge (and the form in which the knowledge is available). The more design knowledge available in the right form, the more *routine* (or "direct") a design process can be used (involving a top-down refinement and/or hillclimbing process that converges on an acceptable design with little or no search). Any missing knowledge or knowledge in the wrong form or incorrect knowledge must be compensated for. Such *innovative* design problems can be addressed by various "indirect" techniques such as case-based reasoning, structural mutation, combining multiple knowledge sources, and explicit planning of the design process.

Design processes can be non-routine and *indirect* in the sense that generating new points in the design space may require an explicit problem-solving process, rather than the direct application of a single procedure or the direct interpretation of a single piece of knowledge. Using case-based reasoning to generate new points in the design space is usually indirect in that it requires nontrivial processes of design case selection, adaptation, and reuse. Using structural muta-

tion to generate new points can be indirect in the sense that the quality and even the functionality of the mutations may not be knowable *a priori*, may require a problem-solving process (e.g., qualitative or numerical simulation) to determine, and may lead to a search through the space of possible mutations for a correct and good one. Using multiple knowledge sources to generate new points in the design space is usually indirect in that integrating partial solutions is a nontrivial problem-solving process.

Design processes can also be non-routine and indirect in the sense that *control* of the search is indirect -- it requires an explicit problem-solving process, rather than merely the direct application of a simple control procedure or the direct interpretation of a single piece of control knowledge to decide what to do next. The design search control problem can be usefully viewed as a *planning problem*, and various planning techniques can be applied: forward or backward planning, "hierarchical planning" (i.e., top-down refinement of design plans), or "exploratory design" (i.e., hillclimbing in the space of problem formulations).

The mapping of a knowledge level specification of a design system into an algorithm-level search algorithm can draw on formally represented bodies of generally useful "common sense" knowledge and procedures relevant to reasoning about the physical artifacts being designed. Much has been learned regarding qualitatively reasoning about *physical systems* in general. We have initial answers to such questions as: how to qualitatively simulate certain classes of physical systems; how to derive aggregate system behavior from the behavior of the parts; how to determine the function of the system given its aggregate behavior and a description of the system's context; etc. Much also has learned about (qualitatively) reasoning about the *geometry* of physical objects in general: how to satisfy placement and sizing constraints; how to satisfy constraints involving forces being applied at various points in space; how to satisfy kinematic constraints on how physical structures can move; how to analyze stresses based on shape; and how to simulate a mechanism's movement through space.

Mapping an algorithm-level search algorithm into a program. Implementing a design search algorithm can involve several types of tasks: coding in an appropriate programming language, such as C++, LISP, OPS5, KEE™; using commercially available tools for representing design artifact representations (e.g., parametric modellers) and for processing common tasks (e.g., constraint managers, geometric modellers and constraint managers, engineering databases); instantiating a domain-independent, design process shell (e.g., for hierarchical refinement and constraint propagation); and creating customized procedures or algorithms for special purpose tasks, either by hand, or by running a knowledge compiler.

Design as part of a larger engineering process. Design is only one phase of

aspect of a larger engineering process that also includes market studies, conceptualization, research and development, manufacturing, testing, maintenance and marketing. The more the design process can be integrated with the other engineering phases, the more cost-effective the entire process will be. Approaches to computer-aided support of an integrated engineering process can range from loose couplings of the phases (facilitated by electronic mail, or shared files, or blackboard architectures), to tight couplings that constrain earlier phases (e.g., design) with requirements anticipated in later phases (e.g., manufacturing constraints) and reformulated so that they are expressed in the language of the earlier phases.

Other summary references. We have intended this chapter as a brief but complete summary of the state of the field of AI in Engineering Design. Other useful summary references worth reading include [3] (which introduced the "routine", "innovative", and "creative" design distinction), [21] (which distinguishes different design process models on the basis of types of design goal interactions), and [43] (which introduced the distinction between the "program level" and the "algorithm level", which was called the "function level" in that paper).

1.8. APPENDIX A: VENDORS OF SOME AI-BASED TOOLS FOR COMPUTER-AIDED ENGINEERING

Ashlar, Inc.
1290 Oakmead Pkwy.
Sunnyvale, CA 94806
Tool: Vellum™

Cognition Inc.
900 Tech Park Drive
Bellerica, MA 01821
Tool: ECMG™ and MCAE™

ICAD Inc.
1000 Massachusetts Avenue
Cambridge, MA 02138
Tools: ICAD™

Integraph Corp.
Mail Stop WYLE3
Huntsville, AL 35894-0001
Tool: MicroStation™

Mentor Graphics
8500 South West Creek Side Place
Beaverton, OR 97005
Tool: ADE™, Logic Synthesizer™

Parametric Technology Corp.
128 Technology Sr.
Waltham, MA 02154
Tool: Pro/ENGINEER™

ComputerVision
55 Wheeler Street
Cambridge, MA 02138
Tool: DesignView™

Spatial Technology
2425, 55th Street, Bldg. A
Boulder, CO 80301
Tool: ACIS™

Wisdom Systems
Corporate Circle
30100 Cagrin Blvd.
Suite 100
Pepper Pike, OHIO 44124
Tool: Concept Modeller™

1.9. BIBLIOGRAPHY

[1] Benjamin, P., Ed., *Change of Representation and Inductive Bias*, Kluwer Publishing, 1990.

[2] Birmingham, W. and Tommelin, I., "Towards a Domain-Independent Synthesis System," in *Knowledge Aided Design*, Green, M., Ed., Academic Press, 1991.

[3] Brown, D. and Chandrasekaran, B., "Expert systems for a class of mechanical design activity," *Proc. IFIP WG5.2 Working Conf. on Knowledge Engineering in Computer Aided Design*, IFIP, September 1984.

[4] Brown, D. and Chandrasekaran, B., *Design Problem Solving: Knowledge Structures and Control Strategies*, Morgan Kaufmann, San Mateo, CA, 1989.

[5] Chandrasekaran, B., "Design Problem Solving: A Task Analysis," *AI Magazine*, 1990.

[6] Clancey, W., "Classification problem-solving," *AAAI*, August 1984.

[7] De Kleer, J., "An assumption-based TMS," *Artificial Intelligence*, Vol. 28, No. 2, pp. 127-162, March 1986.

[8] Dechter, R. and Pearl, J., "Network-based heuristics for constraint satisfaction problems," *Artificial Intelligence,* Vol. 34, pp. 1-38, 1988.

[9] Dechter, R., "Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition," *Artificial Intelligence,* Vol. 41, pp. 273-312, January 1990.

[10] Feigenbaum, E. and Feldman, J., Ed., *Computers and Thought,* McGraw-Hill, New York, 1963.

[11] Gero, J., Ed., *Preprints of the international round-table conference on modelling creativity and knowledge-based creative design,* University of Sydney, 1989.

[12] Coyne, R., Rosenman, M., Radford, A., Balachandran, M., and Gero, J., *Knowledge-Based Design Systems,* Addison-Wesley, Reading, Mass., 1990.

[13] Kelly, Kevin M., Steinberg, Louis I., and Weinrich, Timothy M., *Constraint Propagation in Design: Reducing the Cost,* unpublished working paper, March 1988, [Rutgers University Department of Computer Science AI/VLSI Project Working Paper No. 82].

[14] Guha, R. and Lenat, D., "Cyc: A Mid-Term Report," *The AI Magazine,* Vol. 11, No. 3, pp. 32-59, Fall 1990.

[15] Lowry, M. and McCartney, R., Ed., *Automated Software Design,* MIT Press, Cambridge, MA 02139, 1991.

[16] Maher, M. L., "Engineering Design Synthesis: A Domain-Independent Approach," *Artificial Intelligence in Engineering, Manufacturing and Design,* Vol. 1, No. 3, pp. 207-213, 1988.

[17] McCarthy, J. and Hayes, P., "Some philosophical problems from the standpoint of artificial intelligence," in *Readings in artificial intelligence,* Webber, B. and Nilsson, N., Ed., Morgan Kaufmann, Los Altos, CA., 1981.

[18] Meyer, E., "Logic Synthesis Fine Tunes Abstract Design Descriptions," *Computer Design,* pp. 84-97, June 1 1990.

[19] Mitchell, T. M. and Keller, R. M. and Kedar-Cabelli, S. T., "Explanation-Based Generalization: A Unifying View," *Machine Learning,* Vol. 1, No. 1, pp. 47-80, 1986.

[20] Mittal, S. and Araya, A., "A Knowledge-based Framework for Design," *Proceedings AAAI86,* Vol. 2, Philadelphia, PA, pp. 856-865, June 1986.

[21] Mostow, J., "Toward better models of the design process," *AI Magazine,* Vol. 6, No. 1, pp. 44-57, Spring 1985.

[22] Mostow, J., "A Preliminary Report on DIOGENES: Progress towards Semi-automatic Design of Specialized Heuristic Search Algorithms," *Proceedings of the AAAI88 Workshop on Automated Software Design*, St. Paul, MN, August 1988.

[23] Mostow, J., "Design by Derivational Analogy: Issues in the Automated Replay of Design Plans," *Artificial Intelligence*, Elsevier Science Publishers (North-Holland), Vol. 40, No. 1-3, pp. 119-184, September 1989.

[24] Mostow, J., "Towards Automated Development of Specialized Algorithms for Design Synthesis: Knowledge Compilation as an Approach to Computer-Aided Design," *Research in Engineering Design*, Amherst, MA, Vol. 1, No. 3, 1989.

[25] Nii, P., "The Blackboard Model of Problem Solving: Part I," *AI Magazine*, Vol. 7, No. 2, pp. 38-53, 1986.

[26] Nilsson, N., *Principles of Artificial Intelligence (second edition)*, Morgan Kaufmann, 1984.

[27] Ohr, S., *CAE: A Survey of Standards, Trends, and Tools*, John Wiley and Sons, 1990.

[28] Pentland, A., "ThingWorld: A Multibody Simulation System with Low Computational Complexity," in *Computer-Aided Cooperative Product Development*, Sriram, D., Logcher, R., and Fukuda, S., Ed., Springer-Verlag , pp. 560-583, 1991.

[29] Requicha, A. and Voelcker, H., "Solid Modeling: Current Status and Research Directions," *Solid Modeling: A Historial Summary and Contemporary Assesment*, pp. 9-24, March 1982.

[30] Rich, C. and Waters, R. C., Eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, Los Altos, CA, 1986.

[31] Rychener, M., Ed., *Expert Systems for Engineering Design*, Academic Press, Inc., Boston, 1988.

[32] Setliff, D. and Rutenbar, R., "ELF: A Tool for Automatic Synthesis of Custom Physical CAD Software," *Proceedings of the Design Automation Conference*, IEEE, June 1989.

[33] Sriram, D., Logcher, R., and Fukuda, S., Eds., *Computer-Aided Cooperative Product Development*, Springer Verlag, Inc., 1991.

[34] Sriram, D., Cheong, K., and Kumar, M. L., "Engineering Design Cycle: A Case Study and Implications for CAE," in *Knowledge Aided Design*, Green, M., Ed., Academic Press, 1992.

[35] Steele, G., *The Definition and Implementation of a Computer Programming Language Based on Constraints,* unpublished Ph.D. Dissertation, Massachusetts Institute of Technology, August 1980.

[36] Stefik, M., "Planning and Meta-Planning (MOLGEN: Part 2)," *Artificial Intelligence 16:2*, pp. 141-169, May 1981.

[37] Stefik, M., "Planning with Constraints (MOLGEN: Part 1)," *Artificial Intelligence 16:2*, pp. 111-140, May 1981.

[38] Steinberg, L., Langrana, N., Mitchell, T., Mostow, J., Tong, C., *A Domain Independent Model of Knowledge-Based Design,* unpublished grant proposal, 1986, [AI/VLSI Project Working Paper No. 33, Rutgers University].

[39] Steinberg, L., "Dimensions for Categorizing Design Tasks," *AAAI Spring 1989 Symposium on AI and Manufacturing,* March 1989, [Available as Rutgers AI/Design Project Working Paper Number 127.].

[40] Steinberg, L. and Ling, R., *A Priori Knowledge of Structure vs. Constraint Propagation: One Fragment of a Science of Design,* unpublished Working paper, March 1990, [Rutgers AI/Design Group Working Paper 164].

[41] Suh, N., *The Principles of Engineering Design,* Oxford University Press, 200 Madison Ave., NY 10016, 1990.

[42] Sussman, G., *A Computer Model of Skill Acquisition,* American-Elsevier, New York, 1975.

[43] Tong, C., "Toward an Engineering Science of Knowledge-Based Design," *Artificial Intelligence in Engineering, special issue on AI in Engineering Design,* Vol. 2, No. 3, pp. 133-166, July 1987.

[44] Tong, C., "A Divide-and-Conquer Approach to Knowledge Compilation," in *Automating Software Design,* Lowry, M. and McCartney, R., Eds., AAAI Press, 1991.

[45] Wilensky, R., *Planning and Understanding,* Addison-Wesley, Mass., 1983.

[46] Williams, B, "Interaction-based Invention: Designing novel devices from first principles," *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI90),* Boston, MA, pp. 349-356, 1990.

[47] Winograd, T., *Understanding Natural Language,* Academic Press, New York, 1972.