CHAPTER SEVEN

Artificial Intelligence for Design

Thomas F. Stahovich

INTRODUCTION

Artificial Intelligence (AI) is the study of knowledge representations and inference mechanisms necessary for reasoning and problem solving.[1] AI encompasses a wide variety of topics such as logic, planning, machine vision, and natural language processing. This chapter focuses on those topics that are the most useful for design synthesis: search, knowledge-based systems, machine learning, and qualitative physical reasoning.

This chapter is divided into four main sections, one for each of these topics. Each section discusses the theory behind the techniques, provides examples of their application to synthesis, and summarizes the current understanding about their usefulness. To the extent possible, the sections are independent of one another, and thus can be read in any order without loss of continuity. This chapter draws examples primarily from mechanical engineering; however, the techniques are suitable for many kinds of design problems. The earlier sections describe more mature technologies, whereas the later sections describe technologies with much of the potential left to be explored. The chapter concludes with thoughts about future directions.

SEARCH

Search is one of the oldest AI techniques and was the basis of much of the early work in AI. For example, the Logic Theorist of Newell and Simon (Newell, Simon, and Shaw, 1963) which was one of the first implemented AI systems, used search to prove theorems in propositional calculus (logic). Search has continued to be an important AI tool and has contributed to a number of significant accomplishments. For example, Deep Blue, a chess-playing computer program based on search, recently defeated a grand master (Hamilton and Garber, 1997).

In some applications, search is the primary problem-solving tool. However, even when it is not, search often still has an important role to play. For example, in

[1] Philosophers and scientists have long struggled to define what constitutes both (human) intelligence and artificial intelligence. The definition used here takes the pragmatic view of AI as a set of problem-solving tools. Russell and Norvig (1994) lists several alternative definitions of AI.

rule-based systems (second section), search can be used to implement rule chaining; in machine learning (third section), it is used to find the best hypothesis to explain a set of observations; and in qualitative physical reasoning (fourth section), it is used to find a consistent set of values for the state variables.

This section provides an overview of common search techniques and describes their application to design synthesis. It also describes the key issues in search-based design, including the problem of exponential explosion and the benefits of abstraction. Note that this section covers only a representative sampling of search techniques to illustrate the key issues involved. For a more detailed discussion of individual algorithms, refer to Korf (1988) or any introductory AI text such as that by Winston (1992) or by Russell and Norvig (1994).

SEARCH TECHNIQUES

Path-Finding Problems. A search problem is characterized by a "search space" or "problem space" consisting of states and operators. States are possible solutions or possible partial solutions to the problem. Operators map from one state to another. A particular instance of a search problem is characterized by the initial and goal states, and the search task is to identify a sequence of operators that map from one to the other.

For some problems the goal state is known, and the task is to find a path (sequence of operators) from the initial state to the known goal state. An example is finding an interference-free path by which tubing can be routed from one hydraulic component to another. In this case the location of one of the hydraulic components is the initial search state and the location of the other is the goal state. For many search problems, however, the goal state is not known explicitly, but rather is described implicitly by a test. An example is searching for a layout for a set of objects that will allow them to fit inside a small container (the packing problem). In this case the final state is defined implicitly by a test that determines if the objects fit without interference.

A search space is typically represented as a tree. The nodes in the tree are the search states and the arcs connecting them are the operators. The root of a search tree is the initial state of the search problem. The next level deeper in the tree consists of states that can be reached by applying a single legal operator to the root. Similarly, the nodes n levels deep in the tree are the states that can be reached by applying legal operators to the states at level n–1. Figure 7.1 shows a generic example of a search tree.

The size of a search space is characterized by the branching factor (b) and the depth of the tree (d). The branching factor is the average number of child nodes that can be reached by applying the legal operators to a node. The depth is the distance – number of nodes – from the root to a solution. A good estimate of the total size of the search space is b^d, and thus the time required to exhaustively search the tree is exponential in the tree depth.

Search techniques differ in terms of the order in which they visit the nodes in the search tree. The remainder of this section considers three classes of techniques: brute-force, heuristically informed, and stochastic.

Brute-force techniques search the tree systematically, without using any knowledge of the search space. The two classic techniques are breadth-first and depth-first

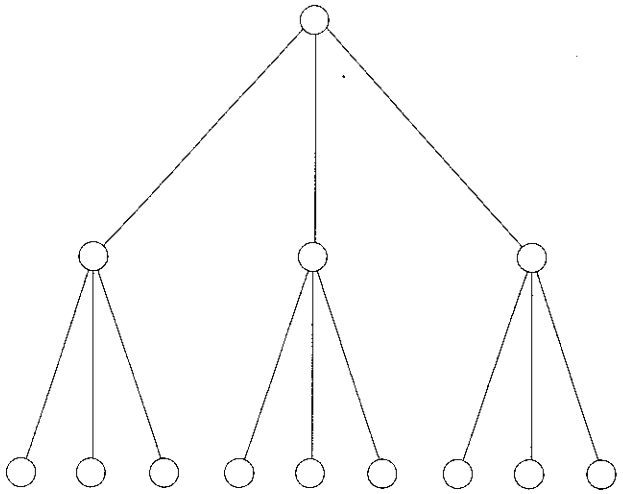


Figure 7.1. A search tree with the branching factor and depth both equal to three.

search. Breadth-first search visits all of nodes at one level of the tree before proceeding to the next level. This requires exponential storage and exponential time; both are proportional to b^d. Depth-first search attempts to extend a single line of reasoning from the root all the way to a solution. Whereas breadth-first search examines the tree row by row, depth-first search examines it branch by branch. Depth-first search selects one child of the root, then one child of that child, and so on, until a solution is found or a leaf of the tree is reached. In the latter case, the search backtracks to the last decision point and selects a different child node. Depth-first search still requires exponential time because on average it visits the same nodes as breadth-first search, but it simply does so in a different order. The benefit of depth-first search is that the storage requirements are only linear in the depth of the tree rather than exponential as in the case of breadth-first search.

If the search tree is infinitely deep, depth-first search may not terminate. For example, if the first branch does not contain the solution and the tree is infinite, the approach will continue along that branch forever. Breadth-first search avoids this problem, but at the expense of exponential storage. Depth-first iteratively deepening (DFID) is a brute-force search technique that avoids both of these problems (Korf, 1985). In DFID, the tree is searched as in depth-first search, except that each branch is terminated when a preselected cutoff depth is reached. Initially, a small cutoff depth is selected. If no solution is found, the cutoff depth is increased and the search is repeated. This process continues in this fashion until a solution is found. Although DFID ends up visiting some nodes over and over again, it can be shown that this does not have a significant adverse affect on performance.

Because search is exponentially expensive, there will always be problems that are too large to solve with brute-force methods, even when the fastest computers are used. Imagine, for example, that the task is to synthesize a device by brute-force combination of components selected from a library. If there were only 10 different kinds of components in the library and we consider devices composed of only 20 components, the size of the search space would be 10^{20}. It is clearly not feasible to search a space this large by brute force. The remedy is to use problem-specific knowledge to guide the search so as to avoid unnecessarily searching large portions of the space. Search methods that do this are called *heuristically informed*. They use a

heuristic measure of the remaining distance to the goal, or some other measure of quality, to decided which nodes in the search tree to visit first. For example, hill climbing expands the best child of the root node, the best child of that node, and so on until a solution is reached. In this sense, hill climbing is analogous to gradient descent in numerical optimization.

Heuristics are often quite effective at reducing the amount of search required to find a solution. Additionally, if the heuristics are known to be an underestimate of the remaining distance to the goal, they can provide a guarantee that the solution is optimal.[2] For example in A^* search, each unexpanded node is assigned a cost equal to the sum of the cost of the solution so far plus an underestimate of the remaining cost of reaching the goal (Hart, Nillson, and Raphael, 1968). A^* expands all nodes whose cost is less than the known cost of the current best solution. When all of the nodes yet to be expanded have a cost greater than the current best solution, that solution is guaranteed to be optimal.

Another way to avoid the exponential cost of search is through abstraction: Instead of directly solving a complex problem, solve a simpler abstraction of it and then fill in the missing details with a second problem-solving effort. For example, in mechanical design, it is often easier to first solve a problem at the functional level and then map each of the functions to an embodiment. (Several examples of this are discussed following, under Search Applications.) Because there are often multiple ways to implement a given function, the search space for functional design is much smaller than for embodiment design. Although abstraction reduces the cost of finding a solution, it may sacrifice optimality: even if each step is solved optimally, there is no guarantee that the final solution is globally optimal.

If heuristics and abstractions are unable to reduce the search space to a tractable size, stochastic search methods are useful. Rather than searching systematically, these methods stochastically sample a large number of points distributed throughout the search space. By pursuing many diverse solutions, these approaches avoid being trapped at local maxima (or minima). Common stochastic methods include simulated annealing and genetic algorithms (Chapter 8).

Constraint Satisfaction. The discussion thus far has focused on search problems that can be characterized as path-finding problems because the task is to find a path (sequence of operators) from the initial search state to the goal state. Another major class of search problems is constraint-satisfaction problems (CSPs). These problems are characterized in terms of a set of variables, a set of possible values for each variable, and a set of constraints on the variables. The task is to select a value for each variable such that the constraints are satisfied. A common example is map coloring in which the goal is to assign a color – selected from a small set of colors – to each country on the map such that no adjacent countries have the same color.

Constraint satisfaction problems can be solved with the brute-force approaches used for path-finding problems, but there are better techniques. The simplest of these is backtracking. This approach begins by assigning an order to the variables. Then, similar to depth-first search, values are assigned to the variables, in order, one at a

[2] There are other optimal search techniques such as exhaustive search and branch-and-bound search that do not rely on heuristics, but these approaches are less efficient.

time. After each assignment, the constraints are evaluated, and if any are violated, the approach backtracks to the first variable for which there are other choices. Another choice is selected and the method continues on in this fashion until either a consistent solution is found or all combinations of assignments have been explored.

Backtracking attempts to resolve constraint violations by returning to the most recent decision point. Often, however, the violation is a result of a much earlier decision and a substantial amount of backtracking is required to resolve the conflict. Dependency-directed backtracking (Stallman and Sussman, 1976) attempts to identify which variable is actually responsible for the conflict so that the solution process can directly backtrack to that variable. Dependency-directed backtracking is thus typically far more efficient than plain backtracking, although in the worst case, both are exponentially expensive.

An even more efficient way to solve constraint satisfaction problems is to preprocess the search space and eliminate any local inconsistencies before searching for a globally consistent solution. The techniques for doing this are called *arc consistency* (Waltz, 1975; Mackworth, 1977). Consider a pair of variables x and y that are related by a constraint. All those values of x that do not have a corresponding legal value in y can be pruned, and vice versa. For example, if x and y are constrained to be equal, then the possible values for x and y can be reduced to the intersection of the initial sets of possible values. To solve a problem, arc consistency is repeatedly applied to each of the constraints until no more variable values can be eliminated. Then, either a backtracking or brute-force approach is used to find a globally consistent solution from the choices remaining. Often arc consistency results in an enormous reduction in the search space so that comparatively little search has to be performed.

SEARCH APPLICATIONS

This section describes synthesis systems that rely on search as the primary problem-solving method.[3] Search has been used for three main types of synthesis problems: (1) synthesis as the combination of standard components, (2) synthesis as repair, that is, synthesis as the application of modification operators to transform an initial design into a working design, and (3) synthesis as the selection of parameter values for a parametric design. This section provides examples of each of these three types of problems.

Synthesis as Combination. Ulrich (1988) uses search combined with the bond graph representation to synthesize single-input, single-output devices. Bond graphs are used to generate schematic designs that are then mapped to implementations. Bond graphs provide a form of abstraction allowing problems to be solved at the functional level before considering embodiments. Bond graphs are a modeling formalism for describing devices composed of networks of lumped-parameter

[3] Many of the systems described later in this chapter also employ search in one form or another. For example, the CADET system described at the beginning of the third section uses search to synthesize a design by selecting a sequence of cases from a case base. Similarly, the LearnIT II system described later in that section uses a form of hill climbing to construct decision trees.

elements including generalized capacitors, resistors, and inductors (Karnopp and Rosenberg, 1975). These elements can be mechanical, electrical, and fluidic. Bond graphs can also model elements such as transformers and gyrators that convert power in one medium to power in another.

The synthesis problem is specified in terms of two bond graph chunks, one describing the input to the device, the other describing the output. Each chunk is associated with a variable. The designer specifies the design requirements in terms of a desired relationship between the input and output variables. Schematic designs are generated by using search: Bond graph elements are chained together until the input chunk is connected to the output chunk. To limit the search, restrictions are placed on the number of bond graph elements a solution can contain. The initial schematic solutions are then evaluated to determine if they provide the desired relationship between the input and output variables. If not, debugging rules are used to modify the bond graph.

Each of the successfully debugged bond graphs is mapped to an implementation by using a library of embodiments for the different types of bond graph elements. The resulting designs are inefficient because each element is mapped to its own embodiment. To produce more efficient designs, a function-sharing procedure is used. This procedure eliminates a component from the embodiment and then uses a set of rules to identify other components that could be modified to provide the missing functionality.

Figure 7.2 shows an example concerning the design of a device for measuring the rate of change of pressure. The input is pressure and the output is a linear displacement that is required to be proportional to the time rate of change of the input. The bottom of the figure shows the final schematic design and an initial implementation. When the function-sharing procedure is applied, the fluid resistance in the bottom branch of the circuit is eliminated and replaced by using an undersized piston that allows leakage.

Williams' IBIS system (Williams, 1990) synthesizes devices by searching through a network of qualitative interactions. The desired behavior is specified as a desired relationship between two or more quantities. For example, the sign of the derivative of one quantity may be specified to be equal to the sign of the difference between two other quantities (a fluid level regulator). IBIS searches through its network of interactions to identify a set of interactions that could connect the specified quantities and achieve the desired relationships between them. Once it has found such a set, IBIS uses a library to map each interaction to a structure that implements it.

Subramanian and Wang (1993, 1995) use search to synthesize mechanisms that transform specified input motions, such as continuous rotation, into specified output motions, such as translational oscillation. They first find a sequence of primitive mechanisms that can achieve the desired transformation. They then produce a detailed design by selecting implementations for the primitive mechanisms from a library as shown in Figure 7.3. Their search algorithm is recursive and works backward from the specified output motion toward the specified input motion. The algorithm begins by identifying all primitive mechanisms that could produce the desired output and selects one at random. If the input of this primitive mechanism is compatible with the specified device input motion, the process terminates. If not, the input of

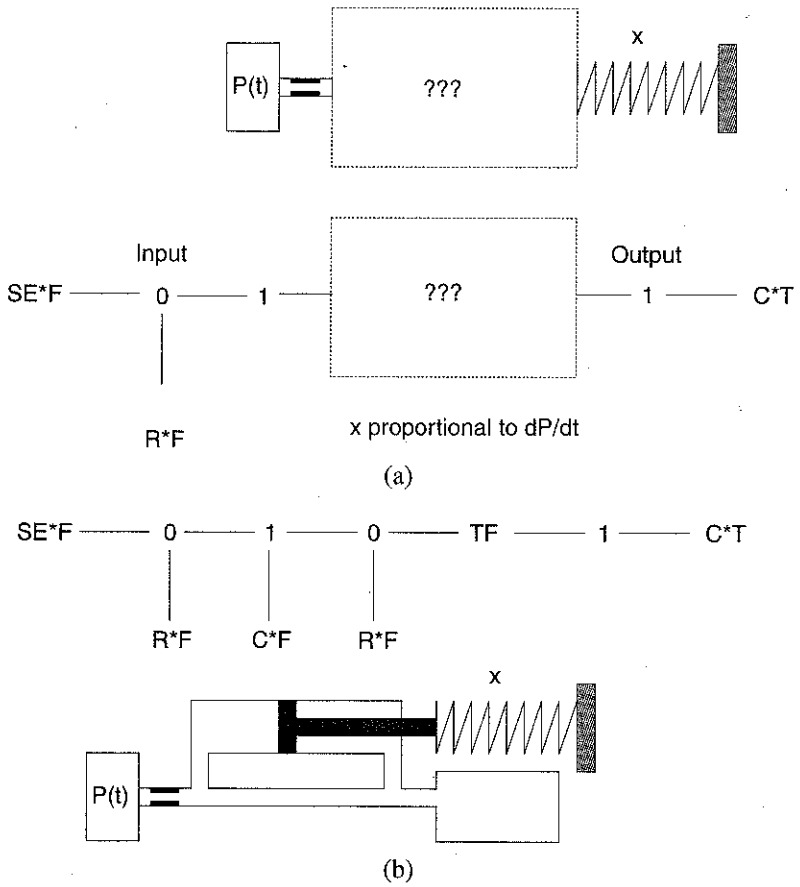


Figure 7.2. A schematic synthesis problem from Ulrich (1988): (a) specification and (b) solution.

this primitive mechanism is treated as if it were the specified device output and the algorithm recurses. In this fashion, the approach is able to chain together components to construct single-input, single-output (SISO) mechanisms.

The approach can also be used to design single-input, multiple-output (SIMO) devices. SIMO devices are initially designed as multiple SISO devices operating in parallel. Then, if any of the SISO solutions have primitive mechanisms in common, those solutions are merged so that the common mechanisms are shared.

Synthesis as Repair. Joskowicz and Addanki (1988) use search to design kinematic pairs (Figure 7.4). The desired behavior of a kinematic pair is described as a desired configuration space (C space). The C space represents the configurations in which the pair of parts interpenetrates (blocked space), the configurations in which they do not touch (free space), and the configurations in which they just touch (boundaries between free and blocked space). Only the latter two types of configurations are legal kinematic states.

Each of the two interacting parts is described as a two-dimensional contour composed of line and arc segments. They begin with an initial contour for each part. They compute the corresponding C space and compare it to the desired C space. If the two do not match, the part contours must be modified. If the actual C space contains a boundary not found in the desired C space, one or more of the contour segments responsible for that boundary must be removed. If, in contrast, the actual C space is lacking boundaries contained in the desired C space, then one or more segments

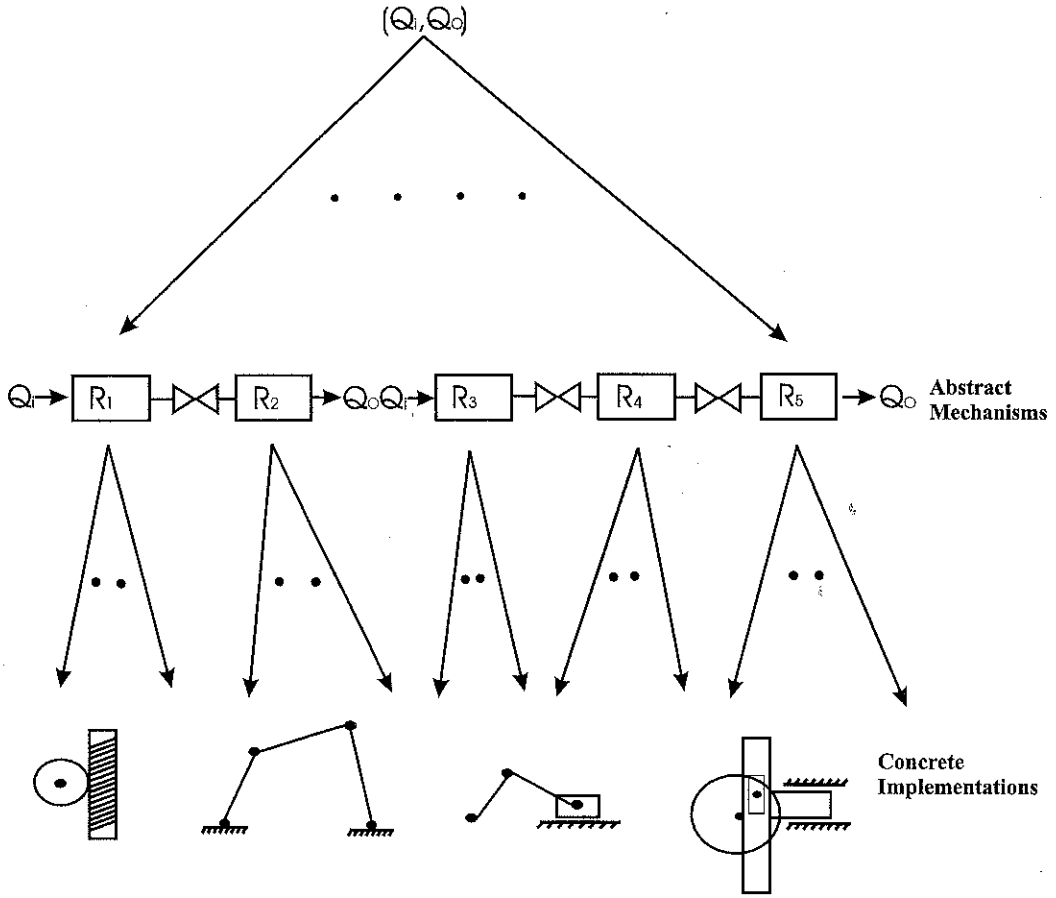


Figure 7.3. Mechanism synthesis using search from Subramanian and Wang (1993). Here Q_i and Q_O are the specified input and output motions; R_j are primitive mechanisms each of which has multiple possible embodiments. The last row shows examples of embodiments.

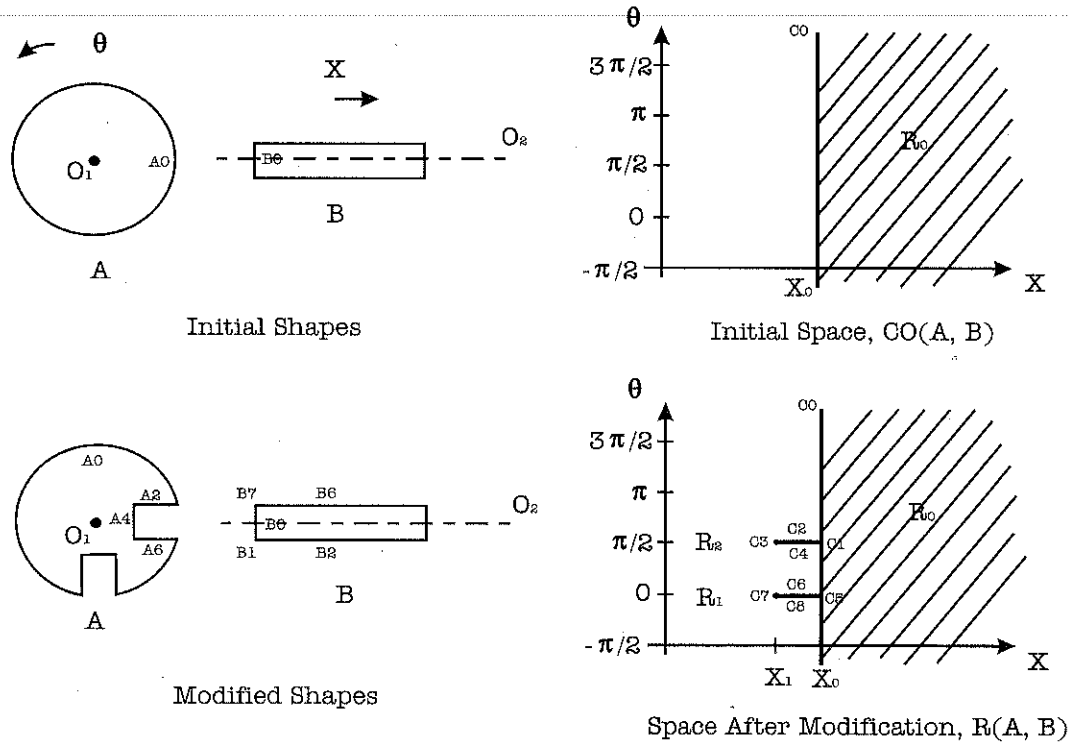


Figure 7.4. The design of kinematic pairs from Joskowicz and Addanki (1988). Left: initial and final part contours; right: initial and final C spaces.

must be added to the contours of the parts. A table of elementary interactions is used to determine which kinds of segments should be added to produce a specific kind of boundary in C space.

The design procedure is a form of backtracking search. At any point in the design process there may be multiple potentially useful modifications. One of these is selected and applied. If this causes any undesired side effects, the search backtracks and another choice is selected. The search process continues in this fashion until the desired C space is achieved.

Shea, Cagan, and Fenves (1997) use simulated annealing (a form of stochastic search; see Chapter 8) and shape grammars (Chapters 2 and 3) to design trusses. Their task is to design a truss of minimum weight subject to constraints imposed by geometric obstacles and stress considerations. Additionally, the number of different sizes (cross sections) of bars that can be used in a single design is constrained to reduce manufacturing cost. The shape grammar specifies legal modifications to the truss, including adding and removing bars, increasing and decreasing the size of individual bars, and moving junction points. Simulated annealing selects and applies modification rules to decrease the weight of the structure while attempting to satisfy the constraints.

Synthesis as Parameter Selection. Orelup and Dixon's Dominic II system uses hill climbing to solve parametric design problems (Orelup and Dixon, 1987). A design problem is described by a set of design variables that the designer can directly adjust and a set of performance parameters that evaluate the quality of the design. The designer specifies the constraints on the variables and the range of acceptable values (good, fair, and poor) for the performance parameters. Starting from some initial design state, Dominic II uses hill climbing to adjust the variables and improve the performance parameters. The program monitors its own performance and detects when the search becomes unproductive. For example, the search may cycle between design states or it may be unable to make changes due to an active constraint. The program selects new search strategies when these kinds of situations occur. For example, the program may allow the search to (temporarily) move to states with lower performance or it may change two variables at once in order to satisfy an active constraint.

CURRENT UNDERSTANDING

Synthesis as the combination of standard components has been extensively explored. This approach is typically used to assemble components that have well-defined input and output ports such as motors (current in, rotation out), racks and pinions (rotation in, translation out), gear reducers (rotation in and out), hydraulic cylinders (pressure in, translation out), and so on. There are several advantages to using these types of components. First, they provide an easy means of ensuring compatibility between components: two components are compatible if their ports are of the same type. This compatibility test often provides a significant reduction in the size of the search space because partial solutions violating the test can be pruned without need of further exploration. Second, these types of components have composable behaviors: The behavioral model of a component is independent of the components to which it is attached. Thus the model of a complete device can be assembled by linking together predefined component models. Third, for these types of devices, the desired

function can be conveniently described as a desired qualitative relationship between scalar parameters. For example, the desired function of a pressure gauge is for a displacement to be proportional to a pressure.

The main limitation of synthesis as combination, as the name suggests, is combinatorial explosion: the size of the search space is n^m where n is the number of available components and m is the maximum number of components allowed in the design. The exponential problem size often prohibits brute-force search. Abstraction is commonly used to help manage problem size. For example, the systems described above use bond graphs, abstract mechanisms, and qualitative interactions to synthesize an abstract functional design. This functional solution is then used as the starting point for embodiment design. This two-step process replaces one large exponential with the sum of two much smaller exponentials. Although abstraction is often quite effective at reducing problem size, it is often still necessary to use heuristics to guide the search process.

Synthesis by repair is another common application of search to synthesis. This approach also suffers from combinatorial explosion, however, heuristics are often available in the form of explicit debugging knowledge. For example, Joskowicz and Addanki (1988) repair shape by using explicit knowledge of which modifications are likely to produce particular boundaries in C space.

Heuristically informed search techniques such as hill climbing have been used for selecting parameter values in parametric design. However, numerical optimization techniques often perform better for this application. One of the deficiencies of hill climbing is that the solution can get trapped at a local maxima before reaching the solution. Stochastic optimization techniques (Chapter 8) are particularly good at avoiding local maxima.

KNOWLEDGE-BASED SYSTEMS

Knowledge-based systems (KBSs) have been widely used in design. These types of computer systems are often called *expert systems* because they solve problems by using knowledge obtained from experts and because they can often achieve expert-level performance. A knowledge-based system consists of a knowledge base and a compatible inference engine. There are a variety of different knowledge representations for constructing knowledge bases, and thus a variety of different inference engines. The representations differ in the types of inferences they support and how they describe facts about the world. Davis, Shrobe, and Szolovits (1993) provide a comprehensive analysis of existing knowledge representation technologies. In this section, we discuss the technologies most commonly used in design: rules and frames.

Rule-based systems describe knowledge in the form of production rules (Davis, Buchanan, and Shortliffe, 1977; Davis and Lenat, 1982; Hayes-Roth, Waterman, and Lenat, 1983; Buchanan and Shortliffe, 1984; Dym and Levitt, 1991). A rule is composed of an "if" part, called an antecedent, and a "then" part, called a consequent. The antecedent is a set of patterns or clauses that indicate when the rule is applicable. The consequent describes the deductions to be made or the actions to be taken when the rule is executed. Figure 7.5 shows an example of a rule from R1,

IF:

- The most current active context is putting unibus modules in the backplanes in some box
- It has been determined which module to try to put in a backplane
- That module is a multiplexer terminal interface
- It has not been associated with panel space
- The type and number of backplane slots it requires are known
- There are least that many slots available in a backplane of the appropriate type
- The current unibus load on that backplane is known
- The position of the backplane in the box is known

THEN:

- Enter the context of verifying panel space for a multiplexer

Figure 7.5. A rule for configuring computer systems from R1 (McDermott, 1981).

a rule-based system that translates a customer's requirements for a computer system into a detailed configuration of components.

Individual rules are small chunks of knowledge. Solving a complete problem typically requires chaining together multiple rules with a rule-chaining engine. In forward-chaining systems, the rule antecedents are matched against the known facts to determine which rules are applicable. If more than one rule applies, a conflict resolution strategy is used to determine which should be executed first. Common strategies include picking the most specific rule (the one with the most clauses in the antecedent) or picking the rule whose antecedent is satisfied by the most recently deduced facts. The rule chainer continues to execute all applicable rules until there are none remaining. Note that in contrast to traditional procedural programs, rule-based systems do not provide a means for explicitly controlling the order of the program's execution. The rule chainer, rather than the system designer, determines the order in which the rules are used.

Frame-based systems describe knowledge in terms of taxonomic hierarchies (Bobrow and Winograd, 1977; Brachman and Schmoze, 1985).[4] A frame can be a "class frame" describing an entire class of objects or an "instance frame" describing a particular instance of a class. For example, one frame could represent the entire class of trucks, whereas another could represent a particular red truck (i.e., an instance). Frames contain slots for describing the attributes of a class or instance. For example, a frame describing a truck may have a slot for the cargo capacity. Class frames are

[4] Frames grew out of work in semantic networks. See Brachman (1979).

organized in a superclass–subclass hierarchy in which subclasses inherit attributes from their superclasses. Inferences are made about objects (instance frames) by knowing the classes to which they belong.

Rules and frames are often combined to form a hybrid representation (Stefik et al., 1983; Kehler and Clemenson, 1984; Fikes and Kehler, 1985). This is accomplished by associating sets of rules with individual frames. The frame taxonomies serve to partition the rules and define their scopes of application. This helps the system designer control when and for what purposes different rules are used. Frames also provide a language for describing the objects referred to in the rules. Additionally, frames provide a means for making certain inferences about objects, based on class membership, without need for explicit rules. This hybrid representation is perhaps the most common knowledge representation for design tools.

KNOWLEDGE-BASED SYSTEMS APPLICATIONS

The origins of knowledge-based systems are generally traced to the DENDRAL system (Feigenbaum, Buchanan, and Lederberg, 1971), which used heuristic knowledge to interpret mass-spectroscopy data and infer the structure of an unknown compound. The MYCIN system (Shortliffe, 1974; Davis et al., 1977), which used production rules to select antibiotic therapies for bacteremia, was the first rule-based system with a separable knowledge base and inference engine. MYCIN's inference engine, called EMYCIN (van Melle, Shortliffe, and Buchanan, 1984), was used to implement a variety of other rule-based systems including SACON, a system that assisted an analyst in the use of a complicated finite-element analysis tool (Bennett and Englemore, 1984).

Knowledge-based systems have been used widely in design. For example, they have been used to design computer systems (McDermott, 1981), V-belts (Dixon, Simmons, and Cohen, 1984), VLSI devices (Subrahmanyam, 1986), pneumatic cylinders (Brown and Chandrasekaran, 1986), paper transport systems (Mittal and Dym, 1986), dwell mechanisms (Kota et al., 1987; Rosen, Riley, and Erdman, 1991), and electrical transformers (Garrett and Jain, 1988). Here we review two of these systems to illustrate the issues involved in building knowledge-based systems for design.

R1. The R1 systems (also called XCON) is one of the best known rule-based systems (McDermott, 1981; Bachant and McDermott, 1984; van de Brug, Brachant, and McDermott, 1986; Barker and O'Connor, 1989). It was developed by Digital Equipment Corporation to configure built-to-order computer systems. R1 took as input a list of computer components that a customer had ordered and produced as output a set of diagrams showing how those components should be assembled. The program also determined what other components were needed to complete the order and produce a functional computer.

R1 decomposed the configuration task into a set of loosely coupled, temporally ordered subtasks. The program imposed a partial order on the set of components to be configured, such that if the components were configured in that order they could be configured without need of backtracking. The program determined the partial ordering dynamically, based on the characteristics of the problem at hand.

R1 had a knowledge-base of approximately 10,000 rules and a database of approximately 30,000 computer components. Figure 7.5 shows a typical rule. The rules were placed into groups on the basis of the subtasks to which they were relevant. This allowed each rule to presuppose certain things about the current state of the configuration without need of explicit clauses in the antecedents. In performing a subtask, the program applied all applicable rules relevant to the subtask. When multiple rules were applicable, the program used generic conflict resolution strategies to determine which to apply first. When all of the applicable rules had been used, the subtask was completed and no additional rules were needed to verify success.

In contrast to traditional procedural software, the R1 system required a substantial amount of ongoing maintenance. As a result of new product releases and new configuration techniques, 40% of R1's rules changed each year. Because of the large number of rules in R1's rule base, adding new rules presented a significant technical challenge. The main difficultly was in bounding the potential relevance of a piece of knowledge and controlling which piece to apply when more than one was relevant. To overcome these difficulties, R1's developers created the Rime problem-solving approach (van de Brug et al., 1986). Rime defines six different roles that a rule could serve. These included proposing which configuration operator to apply, rejecting clearly inferior operators, selecting the best of the remaining operators, applying a selected operator, recognizing success, and recognizing failure. By providing the programmer with a precise way to specify the role of any new piece of knowledge, Rime greatly simplified the task of maintaining R1's large scale, evolving rule base.

The R1 system was used at Digital Equipment on a daily basis for over a decade. The program was used to configure hundreds of thousands of computer systems and was estimated to have saved the company $40 million annually (Barker and O'Connor, 1989).

PRIDE. PRIDE is a knowledge-based system for designing paper transport systems in copy machines (Mittal and Dym, 1986). A design problem is described to the system in terms of the required locations of the paper entrance and exit, the properties of the paper to be used, the timing requirements, the desired entrance and exit speeds, and so on. The program specifies a design solution in terms of the number and locations of the pinch rolls, the materials for the rolls, the values of various geometric parameters, and the like.

In PRIDE, the plan for designing a transport system is expressed as a hierarchy of design goals that decompose the design process into simpler steps. To begin a new design, the top-level goal of designing a paper transport is instantiated. This, in turn, results in the instantiation of subgoals representing subproblems, such as deciding how many roll stations are needed and where they should be located. Figure 7.6 shows a snapshot of the goals that exist when a new design has begun.

Each goal is an autonomous specialist responsible for designing some subset of the design parameters. A goal contains all of the alternative ways or "methods" for making a decision about the values of the goal's design parameters. There are a variety of different kinds of methods: A method can be a "design generator" that explicitly chooses parameter values; a method can be a set of production rules whose consequents are themselves methods; and a method can be a new set of goals.

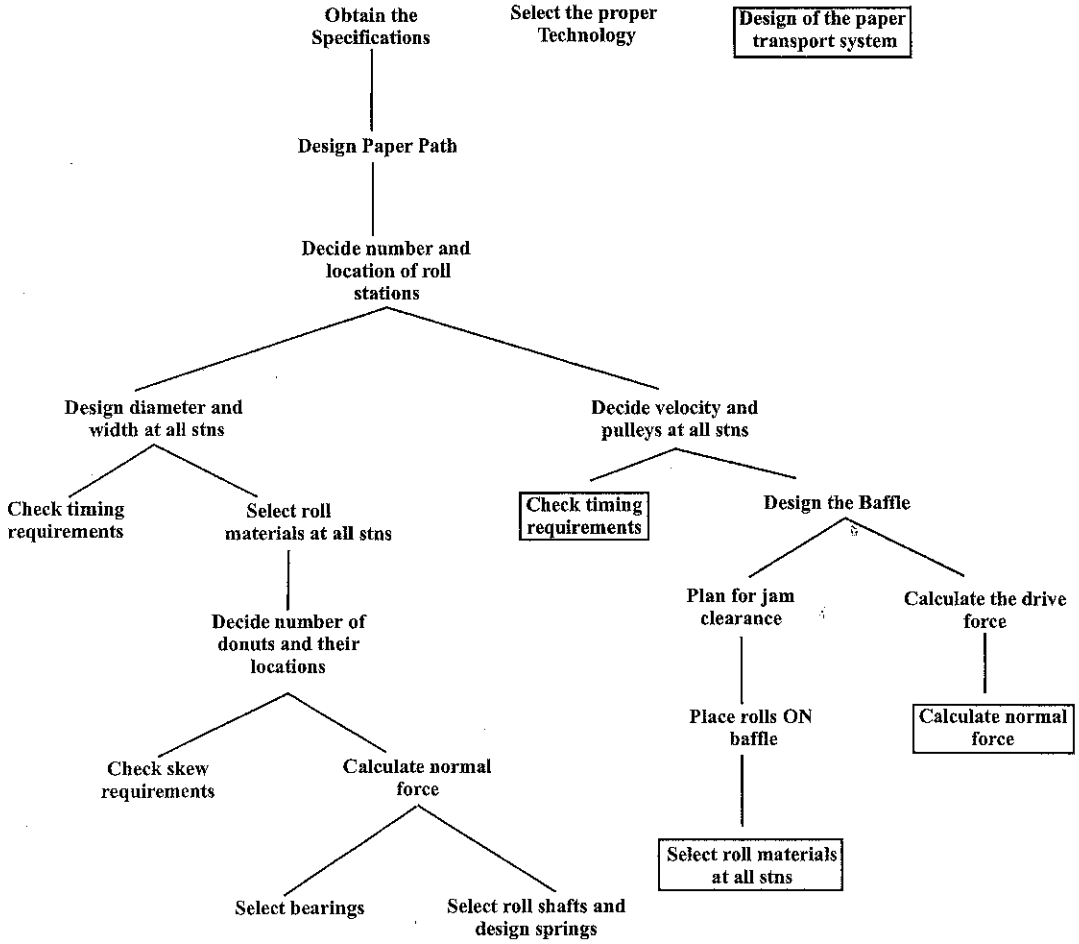


Figure 7.6. A snapshot of the goals that exist when PRIDE (Mittal and Dym, 1986) begins a new paper transport design problem.

The latter type of method is the mechanism by which the program traverses the hierarchical design plan.

Each goal contains a list of constraints that verify that the design is satisfactory. If a constraint is violated, PRIDE uses a form of dependency-directed backtracking to try to resolve the conflict. The violated constraint itself can direct the backtracking by sending advice to the solver. This advice is provided to the system by an expert and is explicitly included in the representation of the constraints.

The PRIDE system is used daily by a copy machine manufacturer for performing feasibility studies of new copier designs. The program performs competently and can do in 30 minutes what previously took weeks (Mittal and Dym, 1986; Dym, 1994).

CURRENT UNDERSTANDING

As knowledge-based systems have been extensively used, a number of specialized development techniques have emerged. These have been documented in a variety of sources, including Davis and Lenat (1982), Hayes-Roth et al. (1983), and Buchanan and Shortliffe (1984). These techniques include specialized software engineering techniques and methods for obtaining knowledge from experts (knowledge engineering).

A significant part of design knowledge is about procedure, that is, what should be done and when. However, rule-based systems were intended to represent declarative rather than procedural knowledge (Davis, 1982). Thus, special efforts must be taken to express the latter. Recall that ordinarily the rule chainer determines which rules are used first. R1 was able to represent procedural knowledge by associating rules with subtasks. Special rules were used to select which subtask to try next. PRIDE used frames to explicitly represent the hierarchy of subgoals to be achieved. Different sets of rules were associated with different frames, thus ensuring that the rules were used only at the proper time.

R1 was able to solve problems without iteration or backtracking. This was primarily due to the nature of the configuration task. Most design problems cannot be solved in this linear fashion, but rather some amount of iteration is usually necessary. The generate-test-debug approach used by AIRCYL (Brown and Chandrasekaran, 1986) and PRIDE (Mittal and Dym, 1986) has proven to be an efficient approach to iteration. One body of knowledge is used to generate candidate designs while a second is used to debug them if they do not meet the design requirements. The debugging knowledge can be acquired from a domain expert and explicitly represented by production rules. Alternatively, dependency-directed backtracking can be used to attempt to debug the deficiency (Stallman and Sussman, 1976; Simmons and Davis, 1987).

Knowledge-based systems work best for problems in which the relevant knowledge is bounded and known. These systems are well suited to symbolic rather than quantitative reasoning. Similarly, they are ill suited to geometric reasoning because it is difficult to represent geometry with a small set of axioms (Forbus, Nielsen, and Faltings, 1991a). Given these considerations, knowledge-based systems are best suited to routine design problems such as selecting components from a library or selecting parameter values in a parametric design problem. As R1 and PRIDE demonstrated, knowledge-based systems can achieve expert-level performance. Furthermore, knowledge-based system techniques can be used to construct robust design tools suitable for production use in real-world industrial applications.

MACHINE LEARNING

Machine Learning (ML) is the subdiscipline of AI concerned with collecting knowledge computationally. A program is said to have learned if its performance at a task improves as a result of previous experiences (Mitchell, 1997). There are several different classes of machine-learning methods. Our discussion is limited to the three most common: inductive methods, instance-based methods, and analytical methods.

Inductive methods draw generalizations from a set of examples by identifying regularities. Most inductive methods operate by computing an approximation to an unknown target function. Common inductive methods include decision-tree learning (Quinlann, 1986; Quinlann, 1993), version-space learning (Mitchell, 1977; Mitchell, 1982), and neural networks (Parker, 1985; Rumelhart and McClelland, 1986). Instance-based methods do not compute an explicit generalization and instead directly match new problems to the most similar training examples. Common

instance-based approaches include the *k*-Nearest-Neighbor algorithm (Cover and Hart, 1967) and case-based reasoning (Kolodner, 1993).

Similar to inductive methods, analytical methods explicitly generalize from training examples (Winston et al., 1983; Mitchell, Keller, and Kedar-Cabelli, 1986; Kedar-Cabelli and McCarty, 1987). However, unlike inductive methods that identify empirical (statistical) regularities, analytical methods use a domain model to construct the generalizations. The domain model is used to determine which properties of the example are significant, and hence which should be the basis of the generalization. Analytical methods are also referred to as "explanation-based learning" because they generalize by constructing an explanation (proof) for the example.

Learning is often used to approximate design spaces in order to accelerate the search for a satisfactory or optimal design. For example, Ivezic and Garrett (1998) have used neural networks to predict the performance of parametric designs. Their program is trained on a small number of sample designs and can then predict the performance of new designs without incurring the cost of expensive simulations and analyses. Similarly, Jamalabad and Langrana (1998) use an instance-based approach to accelerate numerical optimization by learning sensitivity (derivative) information. Finally, Schwabacher, Ellman, and Hirsh (1998) use decision-tree learning algorithms to learn how to select starting prototypes for numerical optimization problems.

As these examples illustrate, machine learning has a number of uses in design. In the sections that follow, we focus on the two applications that have the most impact on automated synthesis: case-based reasoning and learning and reusing design strategies.

CASE-BASED REASONING

When solving new problems, designers frequently rely on previous experience. Case-based reasoning (CBR) is a learning technique intended to assist in the reuse of previous problem-solving experience. There are two main tasks a case-based system must perform: (1) identifying relevant previous design cases and (2) adapting them as necessary to satisfy the new design requirements. Some case-based systems focus on just the first of these tasks and rely on the designer to perform case adaptation. Other systems perform both tasks.

An important consideration in case-based design is how the cases should be represented. The requirements for a suitable representation depend on the nature of design domain and whether or not the system performs case adaptation in addition to retrieval. For retrieval-only systems, it is often adequate to characterize a case in terms of a fixed set of attributes. This allows for efficient indexing and retrieval of cases. (The cases can be organized in a taxonomic hierarchy to further accelerate the retrieval process.) Other types of data, such as text and CAD models, must be included for the designer to be able to adapt the case to new problems; however, this additional information need not be described in a machine-understandable form. For systems that automate case adaptation, the representation must support a broader range of inferences. Adapting a case typically involves reasoning about structure, behavior, function, and the like, and thus the representation must facilitate this reasoning.

Case-based reasoning has been used in a variety of design domains, including architectural design (Domeshek and Kolodner, 1992), structural design (Maher,

Balachandran, and Zhang, 1995), and mechanical design (Goel, Bhatta, and Stroulia, 1997; Chandra et. al., 1992). Here, we focus on just three systems that illustrate the basic issues in case-based reasoning for design.

CASECAD is a case-based system for the conceptual design of structural systems for buildings (Domeshek and Kolodner, 1992). CASECAD uses a multimedia representation for cases. Each case contains references to CAD drawings, images, and text that are used by the designer to understand the case information. Each case also contains a list of attribute-value pairs that are used for indexing and retrieval. The attributes describe function, such as the desired dead load and wind load capacities; behavior, such as the displacement and the cost; and structure, such as the maximum span and material types. CASECAD has a browsing mode that allows the designer to interactively navigate through the case base. It also has a retrieval mode that allows the designer to retrieve designs by specifying desired attribute values.

Kritik2 is a case-based system for the conceptual design of physical devices (Goel et al., 1997). Conceptual design can be viewed as the task of mapping function to structure. Kritik2 performs this task by using the structure-to-function maps of previous designs to adapt them to new functional specifications. A case in Kritik2 is represented with a structure-behavior-function (SBF) model that explains how the structure of the device accomplishes its functions. The structure of a device in the SBF language is expressed in terms of its constituent components and substances, and the interactions between them. Components include things like batteries and light bulbs, whereas substances include things like electricity and light. The behavior of a device is described as a sequence of causal transitions between states. A state is defined in terms of the existence of a substance, such as the existence of electricity at a light bulb. The function of a device is characterized by the input and output states. Figure 7.7 shows the SBF model describing a red light bulb circuit.

To begin a new design problem with Kritik2, the designer specifies the desired function with an SBF model. The program uses this as a probe into the case memory and identifies all cases that at least partially match the desired function. These are then heuristically ordered by their ease of adaptation. Once the program has selected a design case, it checks if the new design requirements are satisfied, and repairs the design if necessary. The program is able to repair failed designs by modifying components and by changing substances. For example, when adapting the design in Figure 7.7 to a problem with a larger required light output, the program identifies that the voltage of the battery is responsible for the failure of the retrieved design. The program then replaces the battery with a higher voltage part.

Whereas Kritik2 adapts a single case, CADET (Chandra et al., 1992) constructs conceptual designs by combining snippets accessed from multiple previous design cases. Each of CADET's cases is a complete design of some physically realizable device. The cases are described in terms of function, behavior, and structure. The behavior, which is perhaps the most important part of the representation, is defined by a set of qualitative differential relations ("influences") relating the input and output variables. For example, Figure 7.8 shows a case in which the flow (Q) out of a water tap monotonically increases with the displacement (X) of the gate valve.

The desired artifact is described to CADET as a set of input and output variables related by qualitative influences. CADET's task is to synthesize a structure that can implement the desired influences. The program first searches the case base

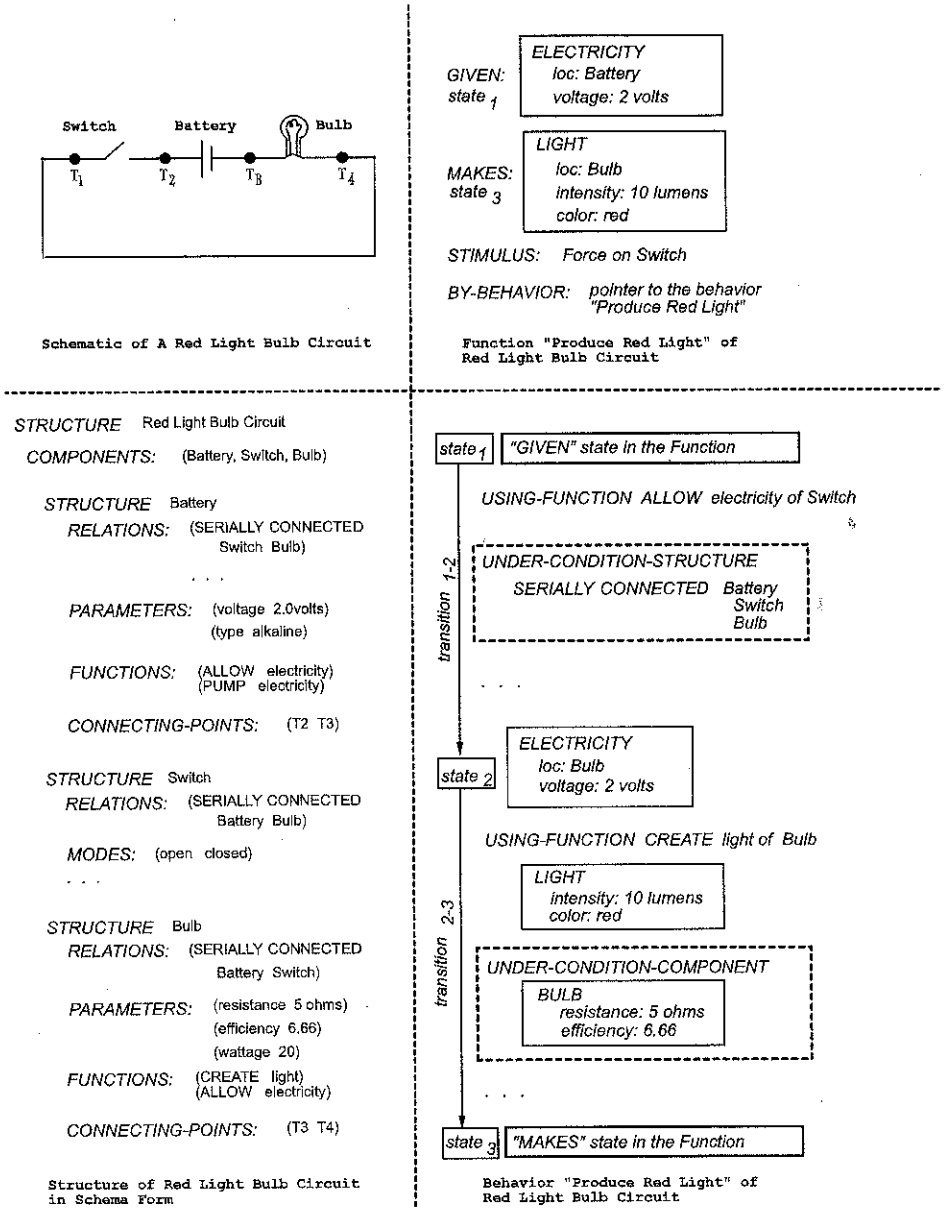


Figure 7.7. The SBF model of a red light bulb circuit from Kritik2 (Goel et al., 1997).

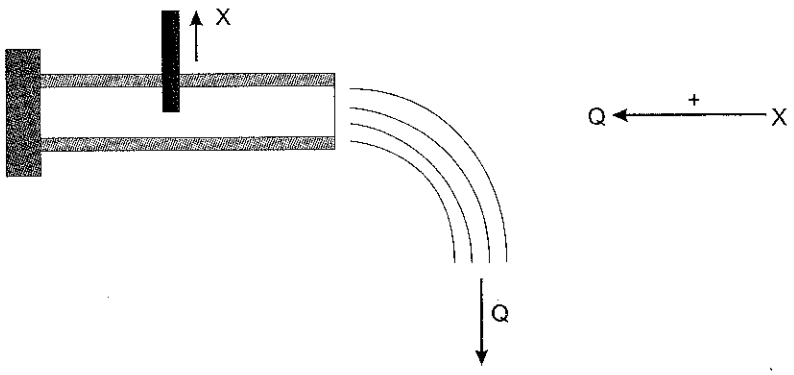


Figure 7.8. The water tap case from CADET (Chandra et al., 1992). Left: physical structure; right: influence graph.

to identify any known devices that could directly solve the problem. For example, if the requirement is to create a design for which a flow of water increases with some linear displacement (Flow ← Linear Displacement), the case in Figure 7.8 would be a solution. For most problems, however, it is necessary to combine several cases. CADET does this by using a set of transformations. For example, if the goal is to create a device for which rotation is caused by pressure (Rotation ← Pressure), CADET might transform this into (Rotation ← Z ← Pressure), where Z is a new variable. CADET then looks for all of the cases that could implement the first influence (Z ← Pressure). The choice of a particular case determines Z and thus initiates the search for a case to implement the second influence. By applying this process recursively, CADET chains together a sequence of cases to produce an influence graph that satisfies the design specification.

LEARNING THE DESIGN PROCESS

With case-based reasoning, previous designs are reused to solve new problems. However, it is sometimes more efficient to reuse the solution process rather than the solutions themselves (Mostow et al., 1989). This section reviews available approaches to learning and reusing design processes.

Much of the early work was in the area of VLSI design. The BOGART system, for example, is able replay design plans created with an interactive VLSI design tool called VEXED (Mostow et al., 1989). VEXED assists the designer in refining a high-level functional specification of a circuit into modules, submodules, and finally components such as transistors and gates. At each stage in the refinement process, VEXED presents the designer with a list of legal refinement rules and the designer selects the best one. VEXED stores this refinement process as a tree of refinement rules called a *design plan*. Once the design plan has been recorded, it can be used to create new designs. Using BOGART, the designer interactively selects a portion of the design plan, which BOGART then replays to solve all or part of a problem.

BOGART assumes that the design plan was recorded while the original design was being created. By contrast, the circuit designer's apprentice (CDA) uses heuristics to generate BOGART-like design plans from existing VLSI designs (Britt and Glagowski, 1996). CDA's approach is called *reconstructive derivational analogy*. CDA uses heuristics to determine which refinement rules might have been used to construct the circuit. Once CDA has inferred a likely refinement plan, that plan can be replayed to solve new problems.

BOGART and CDA are called *replay systems* because they create new designs by replaying the sequence of refinement rules that was used, or may have been used, to construct a previous design. The LEAP system, by contrast, attempts to reuse design knowledge by inferring new refinement rules from previous designs (Mahadevan et al., 1993). It does this by using verification-based learning (VBL), a form of explanation-based learning. LEAP uses a circuit verification theory to prove that a given circuit refinement step is logically equivalent to the original specification. It then generalizes the proof, using a process similar to Prolog-EBG (Kedar-Cabelli and McCarty, 1987), to form a new refinement rule. Figure 7.9 shows an example that concerns the design of a module whose output is specified to be the conjunction

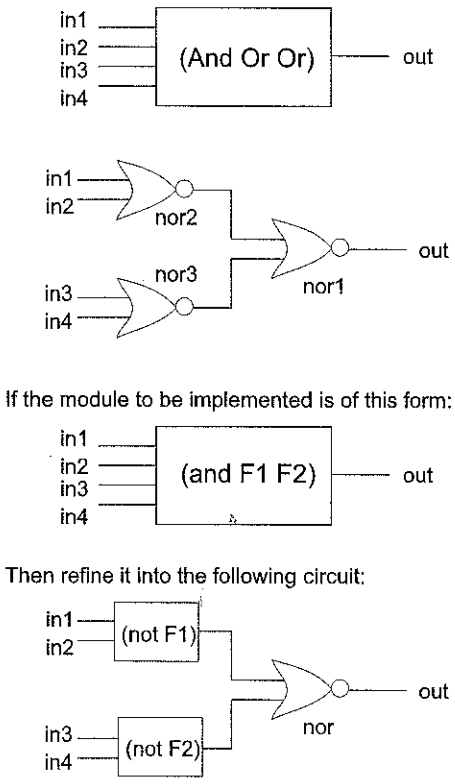


Figure 7.9. Top: specification for a module in LEAP (Mahadevan et al., 1993); middle: the result of manual refinement of the module; bottom: the new refinement rule LEAP learned from this example.

(AND) of two disjunctions (ORs). The designer chose to implement this module by using three NOR (Negated OR) gates as shown in the figure. LEAP first proves that the refined circuit is valid. It then generalizes this proof into the refinement rule shown at the bottom of the figure. This rule states that the conjunction of any two binary Boolean functions (not just ORs) can be implemented by negating each Boolean function and combining their outputs with a NOR gate.

The IDeAL system uses a model-based method for learning generic teleological mechanisms (GTM's) such as cascading, feedback, and feedforward (Bhatta and Goeal, 1997). IDeAL is built on the KRITIK system, an earlier version of the Kritik2 system described above. To learn a GTM, IDeAL requires SBF models of two similar devices, one embodying the GTM, the other lacking it. IDeAL is able to infer the GTM by comparing the two SBF models. For example, when comparing a light bulb circuit with two batteries in series, to one with a single battery, IDeAL learns the notion of cascading: using multiple components in series to achieve a greater output. IDeAL is able to abstract this notion into a domain-independent principle. For example, it can use cascading to increase the temperature drop in a heat exchanger.

LearnIT is a computer program that learns a designer's design strategy by observing how he or she solves a design problem (Stahovich, 2000). The program's domain is iterative parametric design: design problems that are solved by iteratively adjusting a set of parameters until the design requirements, expressed as algebraic constraints, are satisfied. LearnIT is a transparent software layer that sits on top of the usual modeling and analysis software (Figure 7.10). It unobtrusively observes the sequence of design iterations and from this generates a set of design rules describing the designer's strategy. These rules can then be used to automatically create new designs satisfying new design requirements. Because the rules are learned from the

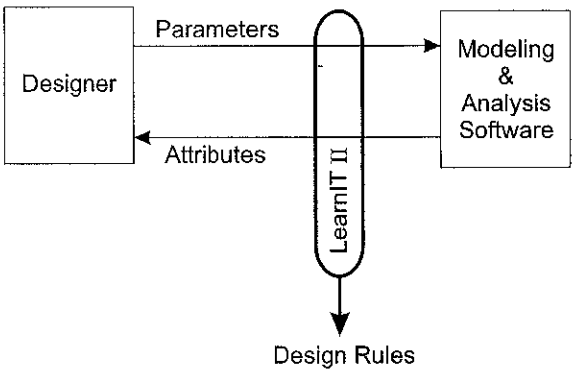


Figure 7.10. LearnIT (Stahovich, 2000) unobtrusively observes the interaction between the designer and the CAD software and learns the design strategy employed.

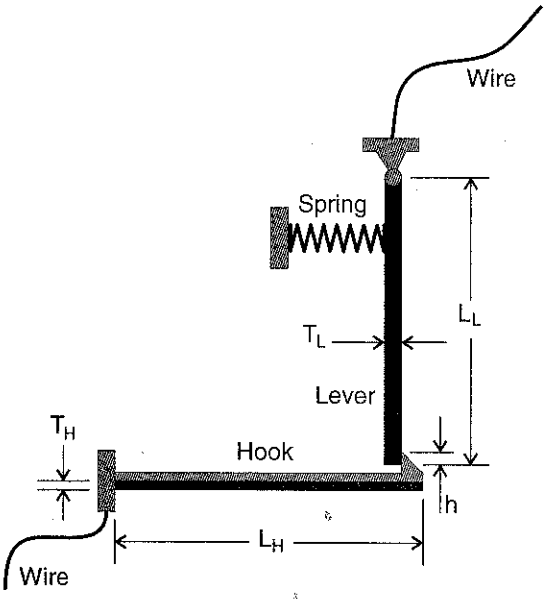
designer, the new designs reflect the designer's engineering judgment, knowledge of implicit constraints, and overall familiarity with the problem.

The program's learning technique is based on two insights. The first is that iterative solutions to design problems are typically a form of debugging. At each iteration the designer identifies the unresolved flaws in the design and chooses a design action to eliminate those flaws. Thus, a design strategy can be thought of as a mapping from design flaws to design actions. The second insight is that the states of the design constraints – whether they are satisfied or not – are often a good indicator of the design flaws the designer is considering at any given time.

These insights lead to a specialized instance-based learning technique. Because the technique is instance based, the learning consists simply of recording observed design iterations. These are recorded in the form of design rules. A rule describes a particular state of the design and the corresponding action to take. The design state is defined by the states of the constraints and the action is a modification (increase or decrease) of a specific parameter.

Figure 7.11 shows a sample rule from the LearnIT system. This rule comes from the design of a circuit breaker. The original design task was to find parameter values to make the device trip after 5 seconds of a 15-amp overload. The sample rule indicates that when the device does not trip at the specified current, the preferred action is to decrease the thickness of each layer of the bimetallic strip. This rule embodies, but does not explicitly represent, several of the designer's insights into this particular design problem. For example, it reflects the fact that reducing the thickness of the bimetallic layers is the most efficient way to increase the hook deflection and thus make the device trip. (The fact that this rule is the most preferred way to repair this flaw is represented by the rule's low rule number.) This modification is particularly effective because it both increases the electrical resistance so that the hook heats faster and decreases the bending resistance. Similarly, the rule's limit embodies the insight that making the hook too thin will make the hook fragile and increase the risk of accidental tripping. Designs created with this rule automatically reflect these implicitly captured insights.

Because LearnIT's approach is instance based, the bulk of the work occurs when a new situation must be matched to a previous rule. LearnIT's rule matching procedure considers the states of the design constraints, the designer's preferences for particular rules, the rule limits, and the likely outcomes of the rules. To create new



Rule 1:
If:
C_1: δ too small
C_2: ΔT too large
C_3: σ less then yield (SAT)
Then:
decrease T_H with a limit of 0.1 mm
Expected outcome:
$\delta - h$ will decrease
ΔT will not change
σ will increase

Figure 7.11. Top: a parameterized circuit breaker model; bottom: a design rule from LearnIT (Stahovich, 2000). Each C_i is a design constraint. δ is the final hook deflection; ΔT is the trip time; σ is the hook stress.

designs, LearnIT applies its rule base in an iterative fashion. It evaluates the state of the design, identifies the best rule, and changes the corresponding parameter. This process repeats until all of the constraints are satisfied or until there are no remaining rules. The new designs that are created in this fashion are similar to those the designer would have created because the rules are learned from the designer.

The LearnIT II system performs the same task as the LearnIT system, but it uses an inductive learning technique rather than an instance-based one (Stahovich and Bal, 2001). LearnIT assumes that the designer's actions are determined primarily by the states of the design constraints. LearnIT II, in contrast, uses decision-tree learning to explicitly determine which properties of the design, or of the design history, best indicate which design actions the designer will take. LearnIT II is able to learn a much broader range of design strategies than LearnIT could. For example, it is able to learn strategies that depend strongly on the design history as well as those that depend strongly on the state of the design itself.

CURRENT UNDERSTANDING

The previous section on knowledge-based systems demonstrated the high level of performance that can be achieved by programs using expert problem-solving

knowledge. One difficulty with knowledge-based systems, however, is that it can be expensive to develop and maintain a knowledge base. The machine-learning techniques discussed in this section provide one means of automatically generating a knowledge base.

Machine learning has a variety of applications in design and engineering. These techniques are useful anytime it is necessary to infer an unknown target function from a set of examples. The learned function can be used to both interpolate and extrapolate from the examples. These techniques can also be used to learn an inexpensive approximation to a known function that is expensive to evaluate. Neural networks are one of the most common techniques for learning continuous-valued target functions, whereas decision trees are one of the most common techniques for learning discrete functions.

There are two main applications of machine learning for synthesis: reusing designs and reusing design processes. Case-based reasoning is the most common technique for design reuse. Case-based systems assist in identifying previous design cases that are relevant to a new problem. Some systems can automatically adapt the previous cases to the new problem. Doing this requires a model of behavior and function. Most current systems rely on hand-generated models. The qualitative physical reasoning techniques described in the next section may provide a means of automatically generating these models, thus greatly extending the range of capabilities of case-based design systems.

The learning and reuse of design processes is an emerging application of machine learning. There is some evidence that suggests it is often more efficient to reuse the solution process rather than the solutions themselves. For this application, some success has been achieved by using learning techniques that identify empirical regularities in the training data. However, the next advances will likely come from explanation-based learning techniques. Here again, the qualitative physical reasoning techniques described in the next section may provide the necessary tools for using explanation-based learning for this application.

As more of design is performed electronically, there are more opportunities to apply machine learning. Much useful information generated during the design process is lost. Machine learning has the potential to capture and preserve this information for future use.

QUALITATIVE PHYSICS

Qualitative physical reasoning (QPR) techniques allow a computer program to perform commonsense reasoning about the physical world. This set of techniques is intended to provide computers with some of the same kinds of physical reasoning abilities that human designer's use in problem solving.

As the name suggests, qualitative reasoning techniques work from qualitative rather than quantitative problem representations. Qualitative representations capture the significant characteristics of a problem and abstract away the rest. For example, for a problem involving fluid flow into and out of a tank, a qualitative representation might describe whether the inflow was less than, equal to, or greater than the

outflow. Even this abstract representation would allow useful predictions to be made. For example, this information is adequate for determining if the tank will eventually empty, although it would not be adequate for predicting how long that would take. The benefit of working from a qualitative representation is that predictions of behavior can be made before the quantitative details have been determined. For example, the IBIS and CADET systems described earlier (in the first and third sections) use networks of qualitative influences to predict the behavior of a candidate design prior to selecting actual physical components.

This section first reviews the two main classes of qualitative reasoning techniques: those suitable for lumped parameter systems and those that consider geometry. Next, two important applications are considered: causal explanation and design generalization. Causal explanations are descriptions of how a device achieves its behavior. These explanations are useful for adapting a design to new applications. Design generalization techniques take a single design, construct an explanation of how it works, and then generate new alternatives that work the same way.

LUMPED PARAMETER SYSTEMS

Much of the early work in qualitative physics focused on devices that could be described with lumped parameter models, such as electric circuits, hydraulic systems, and thermodynamic systems. In a quantitative world these devices are described with ordinary differential equations and algebraic constraint equations. In a qualitative world they are described with qualitative versions of these equations. Typically, these are in the form of qualitative constraints. Behavior is predicted by propagating qualitative values through these constraints.

Among the earliest work was de Kleer's QUAL program, which could produce causal explanations of the small signal behavior of electric circuits (de Kleer, 1979). Small signal behavior is the behavior that occurs within a single operating mode of a device. Each component is modeled with a qualitative constraint equation that relates the signs of the derivatives of the inputs and outputs. The program uses a constraint propagation technique to perform "incremental qualitative" analysis and determine how changes in the circuit's inputs propagate through the circuit.

Williams (1984) created a program that could reason about both the small signal and the large signal behavior of an electric circuit, greatly extending the ability of a program to reason about changes in operating mode. The program computed the small signal model that applied in a particular operating mode, then predicted which parameters changed, possibly causing a transition to another operating mode. The program then used constraint analysis to determine which of the possible transitions would actually happen first.

The ENVISION program, which built upon QUAL, is another system that could reason about changes in operating mode (de Kleer and Brown, 1984). With ENVISION, the behavior of each component is characterized by a set of qualitative states or operating modes. The behavior in each state is characterized by a set of "confluences," describing how changes in variables propagate to other variables. Confluences are concerned with the directions of change rather than the magnitudes. For example, flow through a valve (or other orifice) could be modeled with this

confluence: $\partial P - \partial Q = 0$. This represents the fact that an increase in pressure results in an increase in flow, but it says nothing about the magnitudes of the changes.

ENVISION models the passage of time as a sequence of episodes. Within an episode each component remains in the same state. The behavior within an episode is determined by identifying a consistent direction of change (+, 0, or −) for each variable. The task is a constraint satisfaction problem (see the first section) because the confluences form constraints on the directions of change. ENVISION identifies all possible device states by identifying the consistent variable assignments for each combination of component states. It then reasons about the directions of change to identify legal transitions between device states. Suppose, for example, that device state **A** is valid only when some variable X is less than C, and state **B** is valid when $X \geq C$. If X is increasing when the device is in state **A**, then a transition from **A** to **B** is possible.

Kuipers later formalized the qualitative mathematics for predicting behavior from qualitative constraint equations, resulting in QSIM (Kuipers, 1986). With QSIM, a physical system is described by a set of symbols representing the physical parameters. The value of a parameter is specified in terms of its relationship with a set of ordered landmark values. A set of constraint equations describe how the parameters are related to each other. Some of the constraints are qualitative analogs of common mathematical relationships such as DERIV (velocity, acceleration), MULT (mass, acceleration, force), ADD (net flow, outflow, inflow), and MINUS (forward, reverse). Others specify that one parameter is a monotonically increasing or decreasing function of another: M^+(size, weight) and M^-(resistance, current).

With QSIM, the initial state of a system is defined by a set of active constraints and a set of qualitative values for the parameters. The simulation proceeds by first enumerating all possible qualitative values each parameter can have next, ignoring the constraints. For example, if a parameter is at a landmark and steady, in the next step it may be beyond the landmark and increasing, it may be below the landmark and decreasing, or it may still be at the landmark and steady. The constraints are then applied to identify the legal combinations of next parameter values. It is common for there to be multiple legal combinations and thus the simulation often branches.

Figure 7.12 shows an example of a QSIM simulation of the flow of water between two tanks. Figure 7.12(b) shows the constraints relating the parameters. Initially the levels in the two tanks are the same; then additional water is added to tank **A**. Figure 7.12(c) shows the simulation results. As the water is added to **A**, the pressure at the bottom of the tank increases, causing water to flow into **B**. As the water flows from **A** to **B**, the pressure in **A** decreases while that in **B** increases. As the pressure difference decreases, so does the flow between the tanks. Eventually, the pressure difference and flow become zero. Notice that QSIM is able to predict this behavior without knowing anything quantitative about the system: the specific amounts of water, the density of water, the size of the tanks, and so on are unspecified.

Much of the work in QPR has been device centric, meaning that device models are constructed by assembling models of the individual components. This approach is well suited to systems such as electrical and hydraulic circuits, but it cannot easily handle phenomena such as boiling, which do not involve a fixed collection of "stuff." To handle those kinds of problems, Forbus developed qualitative process theory that takes a process-centric perspective (Forbus, 1984). The representation of a process

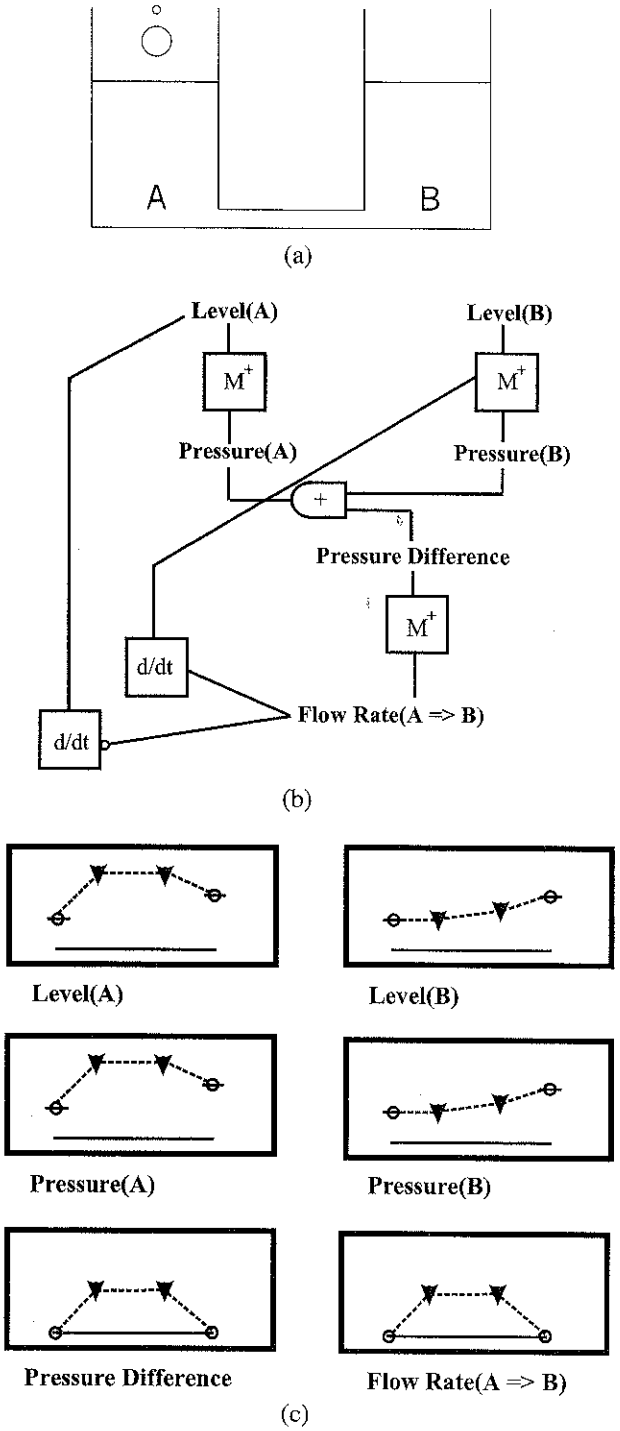


Figure 7.12. An example from QSIM (Kuipers, 1986): (a) Two connected water tanks; (b) the constraint network; (c) simulation results.

includes the set of objects that must exist and the preconditions that must be satisfied for the process to occur. The representation also includes a set of "influences" (similar to confluences) describing how the parameters change when the process is active.

QUALITATIVE PHYSICS WITH GEOMETRY

The first attempt at qualitative reasoning about force and geometry was de Kleer's NEWTON program (de Kleer, 1977), whose domain was the one-dimensional world

of roller coasters. NEWTON used qualitative reasoning to envision the possible behaviors of a particle on a curved track and then used mathematical equations to resolve the ambiguities. The track was described qualitatively as a sequence of segments, with each segment having a constant direction of curvature and a slope of constant sign. The envisionment predicted which segments could possibly be reached and in what order (see e.g., Figure 7.13). For example, when ascending an incline, the particle may reach the top of the segment, and thus reach the start of the next segment, or it may stop part way up and slide back down. The program would use energy equations to determine which of the two behaviors would actually happen in this case.

The next attempt at mechanics problems was Forbus' FROB system, which reasoned about particles in a two-dimensional well (Forbus, 1980). Forbus introduced the notion of a "place vocabulary," which divided the well into regions where the behavior was similar. Simulation proceeded by reasoning about the transitions from one place to the next.

Both NEWTON and FROB reasoned about the motions of individual particles. The next step was to consider the behavior produced by interacting shapes. Initial attempts at this relied on a mix of quantitative and qualitative reasoning. For example, Joskowicz and Sacks created a simulator that combined a qualitative version of Newton's laws of motion with a quantitative geometric reasoning technique. The latter technique worked from a "region diagram," a decomposition of configuration space that is similar to a place vocabulary (Joskowicz and Sacks, 1991). This system could simulate a wide range of practical devices and could produce concise descriptions of the behavior.

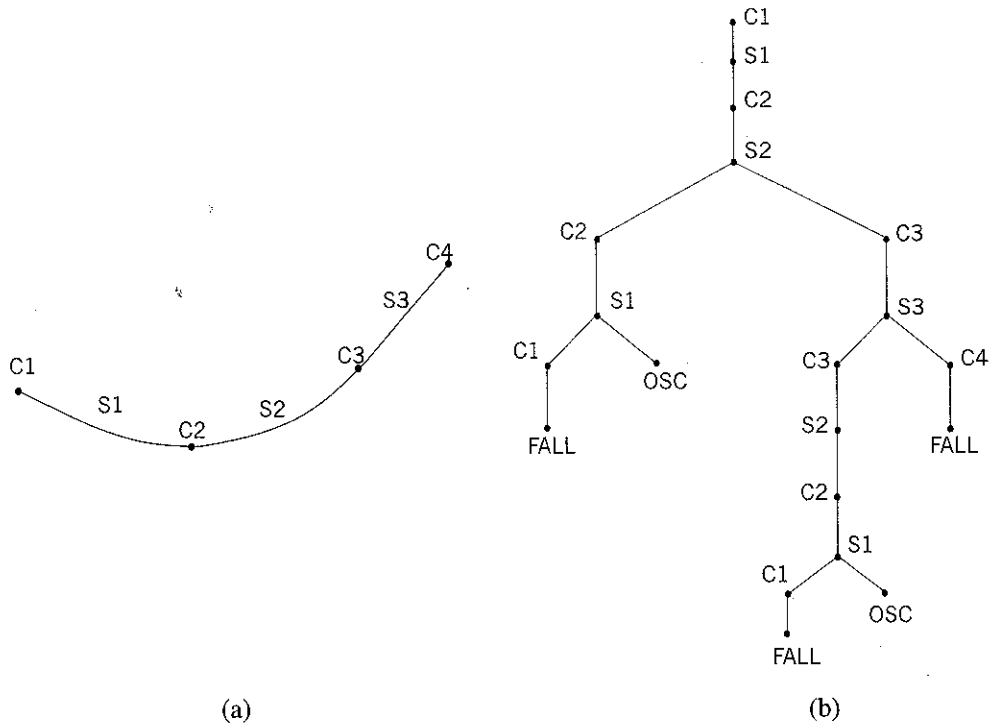


Figure 7.13. An example from NEWTON (de Kleer, 1977): (a) one-dimensional roller coaster with labeled segments; (b) envisionment (tree) indicating possible sequences of motions.

Faltings (1990) developed a system that used place vocabularies to produce qualitative kinematic simulations. The place vocabulary is a qualitative geometric representation that is constructed using quantitative analysis. Thus, this approach, too, is partially quantitative. This work was later combined with qualitative techniques for reasoning about forces and mechanical constraints, producing a system capable of qualitatively predicting rigid-body dynamics (Forbus et al., 1991). This system works as a total envisioner: it computes all possible states of the device and all legal transitions between them. It then computes which of these states are actually visited for a specific set of initial conditions and external inputs.

Stahovich, Davis, and Shrobe (1997, 2000) developed a qualitative rigid-body dynamic simulator that requires no quantitative information. This simulator works directly from a qualitative geometric representation called qualitative configuration space (Qc space). In Qc space a mechanical interaction between a pair of part faces is represented as a qualitative configuration space curve (Qcs curve) describing the configurations of the device for which the pair of faces touch each other (Figure 7.14). Each Qcs curve is a family of monotonic curves all having the same qualitative slope. The end points of the Qcs curves are marked with landmarks. The landmarks are ordered relative to one another, but there are no quantitative distances in Qc space. A special form of Qcs curve, called a *boundary*, is used to represent the neutral positions of springs and the motion limits of actuators.

To compute a step of simulation, the simulator first computes the net force on each body. Summing qualitative quantities often leads to ambiguity. To avoid this, a special qualitative force representation was used. This representation considers the projections of the forces onto the degrees of freedom of the device and describes each force by its magnitude, direction, and the type of constraint it imposes. Once the net forces are computed, quasi-static assumptions are used to determine the direction of motion of each part.

A simulation step ends when an event changes the nature of the forces. Events include two bodies colliding, two objects separating, a spring passing through its neutral position, and a motion source turning on or off. The simulator performs the geometry-intensive task of identifying the next event by working directly from the Qc-space representation. As the parts of the device move, the configuration traces out a qualitative trajectory through Qc space. An event is detected by geometrically determining when the trajectory leaves or reaches a Qcs curve or boundary in Qc space.

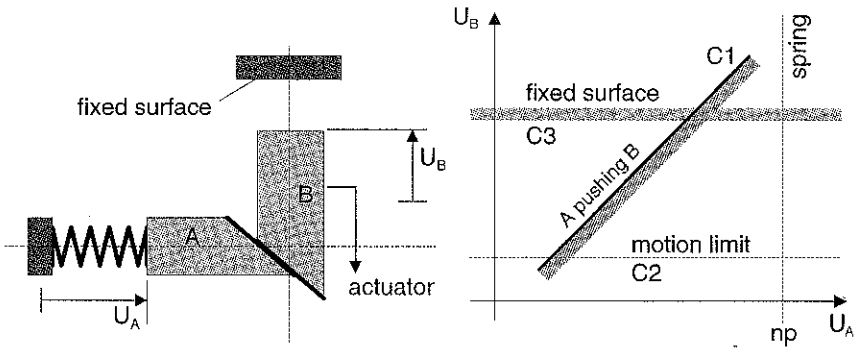


Figure 7.14. A simple mechanical system and its Qc space description (Stahovich et al., 2000).

Sometimes there is more than one possible next event. When this occurs, the simulator branches to consider all of the possibilities. However, unlike an envisioner, which must first compute all possible states of the device, this simulator directly computes just those states of the device that could be visited for the given initial conditions and external inputs.

CAUSAL EXPLANATIONS

This section describes a set of qualitative reasoning techniques that allow a program to produce causal explanations of a device's behavior. These explanations are useful for a variety of tasks including diagnosis and design reuse. Currently, the explanations these techniques generate are intended for human use. However, it may be possible to extend these techniques to create tools for automated design reuse. For example, case-based reasoning systems can adapt previous design solutions to solve new problems, but doing this requires models of behavior and function. Most current systems rely on manually constructed models such as the one in Figure 7.7. Causal reasoning techniques may provide a means of automatically generating behavioral and functional models.

There are a variety of causal reasoning techniques suitable for devices that can be described by ordinary differential equations and algebraic constraints (i.e., lumped parameter models). Many of the qualitative simulation techniques described in the fourth section naturally produce causal explanations. For example both de Kleer's QUAL system (de Kleer and Brown, 1984) and Williams' qualitative circuit simulator (1984) produce causal explanations of behavior. These techniques identify causality by examining the order in which quantities propagate through the qualitative equations. Another technique, called *causal ordering*, produces causal explanations by examining the order in which the equations must be solved; that is, which equations can be solved first, which can be solved once those are solved, and so on (Iwasaki and Simon, 1986; Gautier and Gruber, 1993; Gruber and Gautier, 1993).

Although there are a number of causal reasoning techniques for lumped parameter systems, there are relatively few techniques for reasoning about geometric interactions. One such system is Shrobe's linkage understanding program, which examines kinematic simulations of linkages in order to construct explanations for the purpose of the parts (Shrobe, 1993). The program first numerically simulates the behavior of the linkage by using a kinematic simulator. By examining the order in which the simulator solves the kinematic constraints, the program decomposes the linkage into driving and driven modules such as input cranks, dyads, and four-bar linkages. It then examines traces of the motion of special points on the driven members (such as coupler points) and the angles of the driving members to look for interesting features. Next, geometric reasoning is used to derive causal relationships between the features.

Figure 7.15 shows an example concerning a six-bar linkage. The program decomposes this linkage into a four-bar linkage driving a dyad. When examining the motion traces the program notices: (1) the angle of the output rocker arm (GF) has a period of constant value, (2) the coupler curve (trace of E) has a segment of constant curvature, and (3) the radius of curvature of this segment is nearly equal to the length

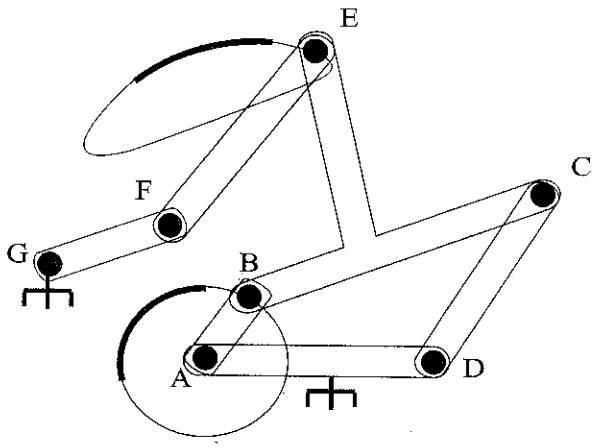


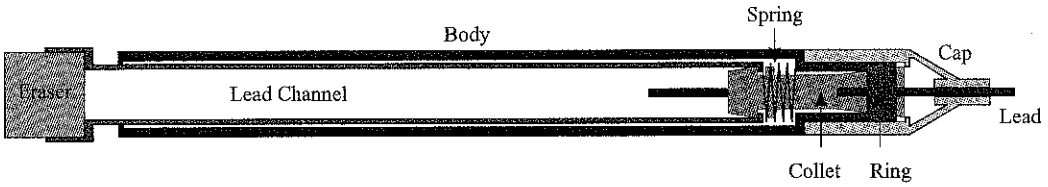
Figure 7.15. Shrobe's linkage understanding systems determines that the purpose of this device is to cause dwell (Shrobe, 1993).

of the driven arm of the dyad (EF). From this and other similar facts, the program hypothesizes that the purpose of the coupler moving in a circular arc is to cause the output to dwell.

Stahovich and Raghavan's ExplainIT program is another system that can produce causal explanations of behavior for devices that depend on geometry. ExplainIT is a computer program that computes the purposes of the geometric features on the parts of a device (Stahovich and Raghavan, 2000). ExplainIT identifies purpose by simulating how removing a geometric feature alters the behavior of a device. Ordinarily simulations describe what happens but not why. Thus, a simulation does not directly indicate which of a device's many behaviors are caused by a given feature on a given part. To identify those behaviors, ExplainIT compares a simulation of the nominal device to a simulation with the feature removed. The differences between them are indicative of the behaviors the feature ultimately causes.

The primary challenge in implementing this "remove and simulate" technique is accurately identifying the differences between the nominal and modified simulations. Direct numerical comparison of the state variables is not useful because there are likely to be differences in force magnitudes, velocities, accelerations, and so on, at every instant of time. Many of these differences are insignificant, such as those resulting from the small change in mass that occurs when the feature is removed. To avoid this problem, ExplainIT abstracts the simulation results into a qualitative form that reveals the essential details. The program then identifies the first point at which the two simulations begin to qualitatively differ. This is the point when the feature must perform its intended purpose to make the device end up in the correct final state. The program uses the laws of mechanics to construct a causal explanation for how the feature causes the behavior observed at this point in the simulation. It then translates this causal explanation into a human-readable description of the feature's purpose. Figure 7.16 shows an example of the kind of explanation the program can provide.

Stahovich and Kara's ExplainIT II system uses a different version of the remove and simulate technique to compute purpose (Stahovich and Kara, 2001). The program starts with two simulations of a device, one with the feature and one without. It then rerepresents the two simulations as a set of processes with associated causes, that is "causal processes." To identify the purpose of the removed feature, the program identifies all causal processes unique to one or the other of the two simulations.



Explanation of the collet taper: The taper on the collet interacts with the ring to produce a wedging force (*i.e.*, the ring is press-fit onto the collet). This force applies a moment to the collet that is balanced by the moment of a force that the lead applies to the collet. By Newton's third law, there must be an equal and opposite force that the collet applies to the lead, and that force gives rise to a friction force, which eventually causes the lead to advance.

Figure 7.16. Top: A mechanical pencil. Bottom: An explanation for the purpose of the taper on the collet paraphrased from ExplainIT's output (Stahovich and Raghavan, 2000).

ExplainIT II's causal-process representation allows the program to reliably determine when a piece of behavior from one simulation is the same as a piece from the other. By a process of elimination, this allows the program to accurately determine when a piece of behavior is unique to one or the other of the simulations. It is common for a particular behavior to repeat multiple times during a simulation, especially if the device operates cyclically. By explicitly considering the causes of behavior, ExplainIT II can determine which instance of behavior from one simulation is the same as that from the other: two similar behaviors are the same if they have the same cause.

DESIGN GENERALIZATION

One of the challenges in automating design synthesis is specifying what is desired. It is often difficult to provide a completely abstract description of the design requirements. The problem is not simply the lack of a suitable specification language. The more challenging problem is thinking about an as yet unrealized design in abstract terms. This section describes an alternative means of describing requirements that avoids this problem. Rather than working from an abstract description, the programs described here work from a concrete example. The designer provides a specific example of the kind of device that is desired, and the programs generalize it to provide alternative designs. These programs could be described as performing "design by example" rather than the traditional "design by specification."

The first example of this approach is 1stPRINCE, which uses a methodology called dimensional variable expansion (DVE) to generalize the design of a structure in the context of numerical optimization (Cagan and Agogino, 1991a, 1991b).[5] The program generalizes a design by expanding the variables representing the critical dimensions (i.e., the critical "dimensional variables") into multiple regions, each of which can have its own set of material properties. This provides additional degrees of freedom to the optimization process and allows the program to innovate new designs that are substantially better with respect to the objective function than the

[5] DVE is technically not a qualitative physics technique in that it reasons primarily about the form of the governing equations. DVE is included in this section because of the kind of task it performs rather than the approach it uses.

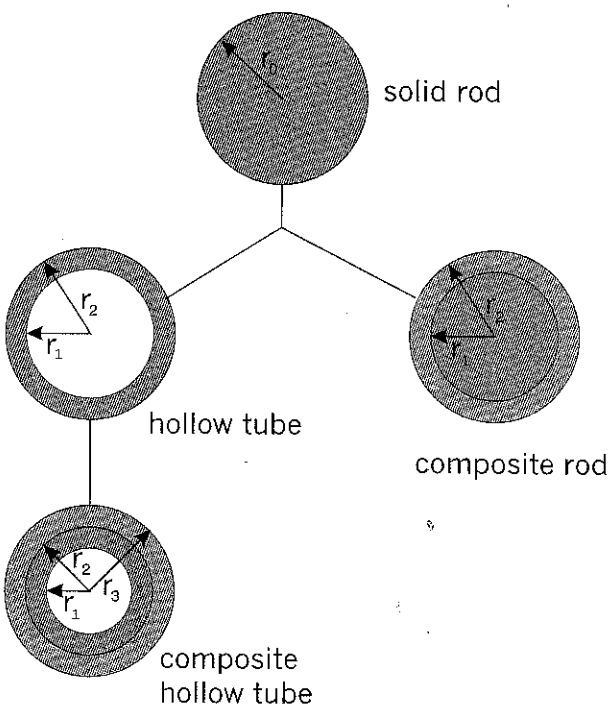


Figure 7.17. Applying DVE to a solid shaft under torsion, 1st PRINCE generates a hollow tube, a composite rod, and a composite hollow tube (Cagan and Agogino, 1991a).

original. Figure 7.17 shows an example in which the objective is to design a torsion shaft of minimum weight. DVE generalizes the initial solid rod into a hollow tube, a composite rod, and a composite hollow tube.

The critical dimensional variables are identified by examining the nature of the design constraints. The solution to a constrained optimization problem is typically determined by a set of "active" constraints. In satisfying the active constraints, the solution naturally satisfies the other inactive constraints, which are still far from their limits. For example, in structural optimization, a constraint imposed by the ultimate tensile strength would be naturally satisfied if a constraint imposed by the yield strength were already satisfied. 1st PRINCE uses monotonicity analysis (Papalambros and Wilde, 1988) to identify the possible sets of constraints that may be active in the optimal solution. Each such set is called a *prototype*. The program applies DVE to the dimensional variables associated with the active constraints in each prototype.

After 1st PRINCE applies DVE and splits a critical dimensional variable into two regions, it repeats the monotonicity analysis. If each of the new regions is limited by the same set of active constraints as the original prototype, the program applies DVE to the two new regions, resulting in a total of four regions. If DVE can be applied for three consecutive iterations, 1st PRINCE uses induction to generalize to an infinite number of infinitesimal regions, resulting in a continuous function. For example, Figure 7.18 shows an example in which the objective is to minimize the weight of a cantilever beam. The initial design has a uniform circular cross section. In the final design, the cross section is a continuous function of the position along the beam.

The second example of "design by example" is Stahovich et al.'s SKETCHIT system, which can transform a stylized sketch of a mechanical device into multiple

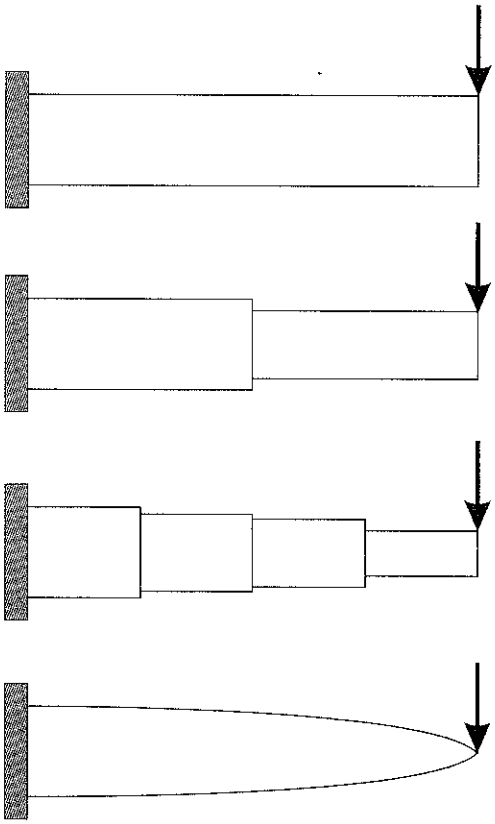


Figure 7.18. Applying DVE to the uniform cantilever beam, 1st PRINCE induces a beam with variable cross section as a means to reduce weight while satisfying stress requirements (1991b).

families of new designs (Stahovich et al., 1996, 1998). SketchIT uses a paradigm of abstraction and resynthesis as shown in Figure 7.19. During the abstraction process, the program reverse engineers and generalizes the original design by using the qualitative configuration space (Qc space) representation described above. Qc space enables the program to identify the behavior of the individual parts of the design while abstracting away the particular geometry used to depict those behaviors.

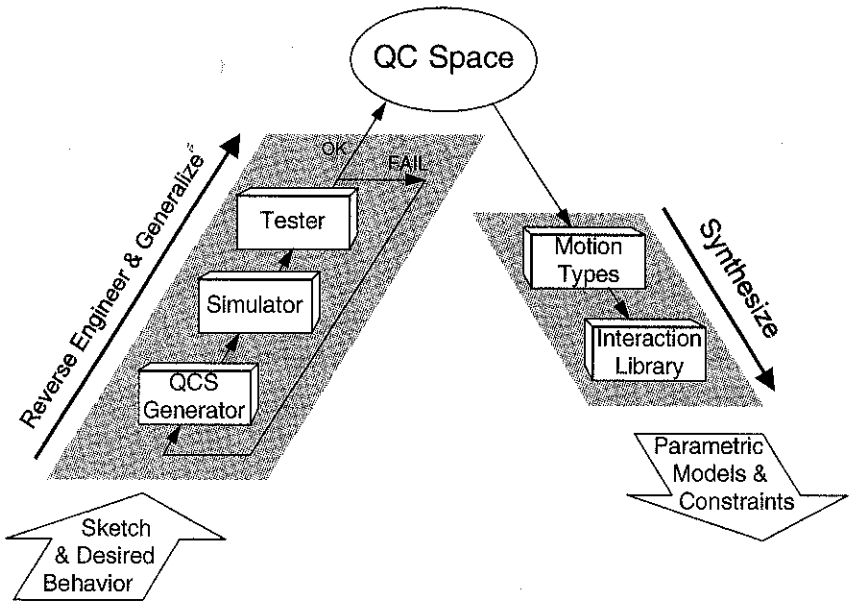


Figure 7.19. Overview of the SketchIT system (Stahovich et al., 1998).

To interpret a sketch, the program begins by directly translating it into an initial Qc space. The program then uses Stahovich's qualitative rigid-body dynamic simulator described above (in a previous subsection) to simulate the behavior of the Qc space and determine if the sketch as drawn is capable of producing the desired behavior. The desired behavior is a specific sequence of kinematic states that this particular geometry should achieve (i.e., the description is concrete rather than abstract). The sequence is described with a state transition diagram.

If the initial Qc space works correctly, the program is done with the abstraction process. If not, the program modifies the Qc space and repeats the process. Once it has found a working Qc space, the program uses it as a specification from which the program synthesizes new implementations. SKETCHIT uses a library of geometric interactions to map the individual parts of the Qc space (Qcs curves) back to geometry. Each library entry is a chunk of parametric geometry with constraints that ensure it implements a particular kind of behavior. By assembling these chunks, the program produces a behavior-ensuring parametric model (BEP model) for the device. A BEP model is a parametric model augmented with constraints that ensure the device produces the desired behavior. Each BEP model is a family of solutions that are all guaranteed to work correctly. Different members of the family are obtained by selecting different parameter values that satisfy the constraints of the BEP model.

SKETCHIT's library contains multiple implementations for each kind of behavior (Qcs curve), and thus the program is able to generate multiple families of implementations (BEP models). Additionally, the Qc space representation allows SKETCHIT to identify when it is possible to replace rotating parts with translating ones and vice versa. This ability provides SKETCHIT with another means of generating design alternatives.

Figures 7.20 and 7.21 show the kind of stylized sketch and state transition diagram that SKETCHIT takes as input. This particular example concerns the design of a circuit

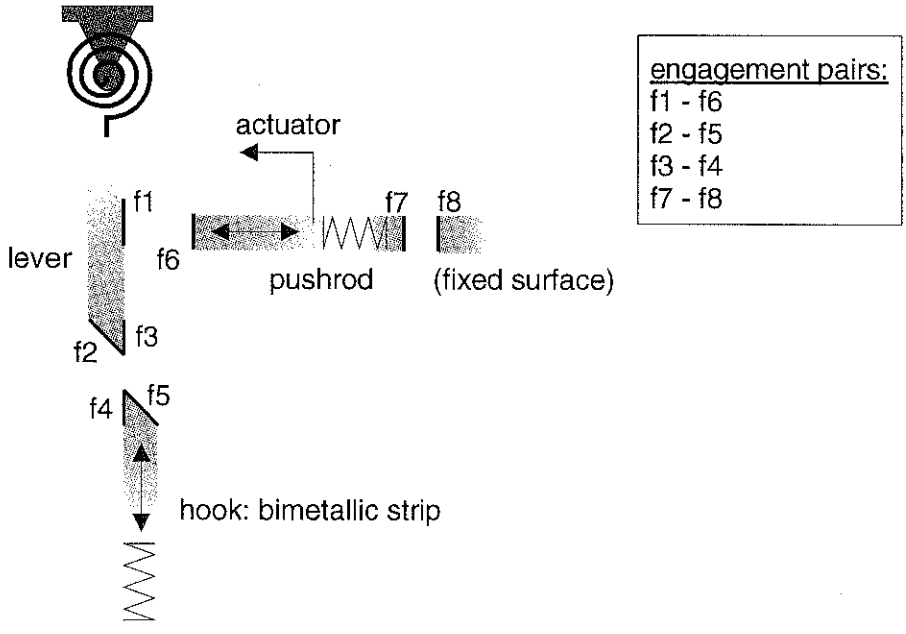


Figure 7.20. A stylized sketch of a circuit breaker given to SKETCHIT as input (Stahovich et al., 1998). Engagement faces are bold lines. The sketch is created in a mouse-driven sketching environment.

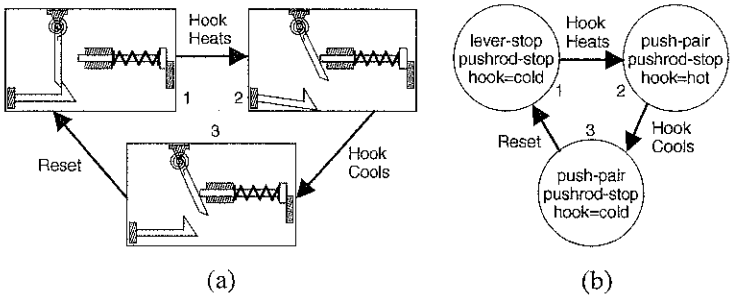


Figure 7.21. The desired behavior of a circuit breaker (Stahovich et al., 1998): (a) physical interpretation; (b) state transition diagram.

breaker similar to those found in a residential electrical system. From this input, the program generates several families of designs. Figure 7.22 shows a portion of the BEP model for one of these families (the one that is most similar to the original sketch). Figure 7.23 shows a different design the program created by selecting new geometry for the interacting faces, and Figure 7.24 shows another design created by replacing a rotating part with a translating one.

QUALITATIVE PHYSICS: CURRENT UNDERSTANDING

Qualitative physical reasoning (QPR) techniques have been used extensively for performing qualitative simulation. They are also frequently used for generating explanations of behavior and for performing diagnosis. The most common application to design is the use of qualitative simulation to reason about the behavior of a device before concrete implementations have been selected for the parts. The IBIS, Kritik2, and CADET systems described in earlier sections all use QPR for this purpose.

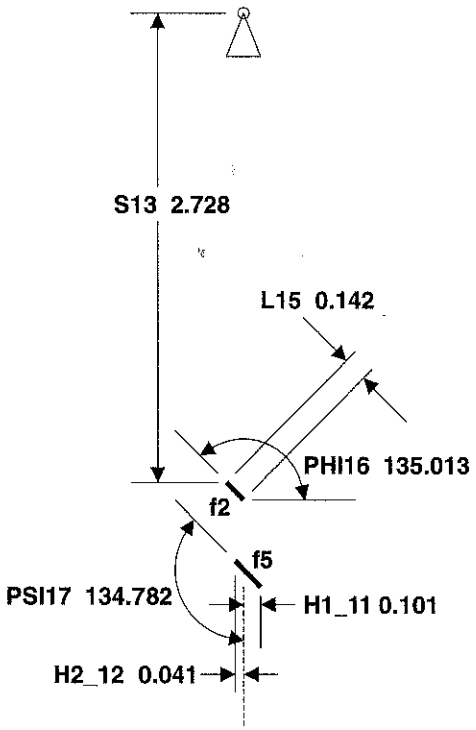


Figure 7.22. Part of a BEP model from SKETCHIT (Stahovich et al., 1998). Top: Parametric geometry; bottom: constraints.

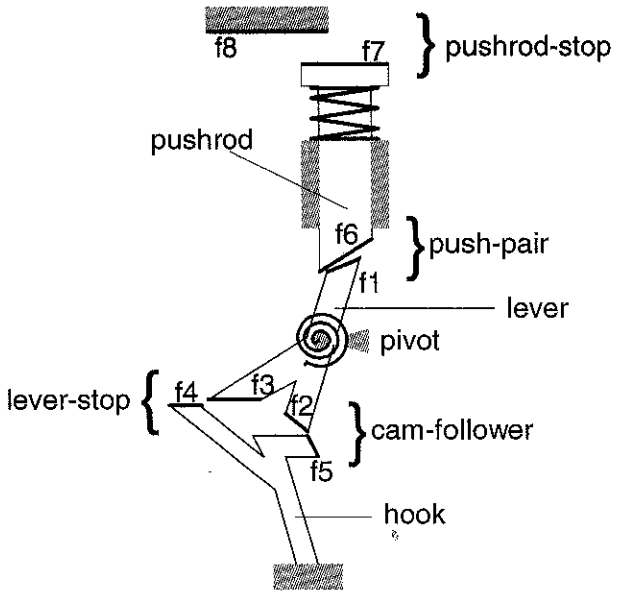


Figure 7.23. A design variant SKETCHIT obtained by using different implementations for the engagement faces (Stahovich et al., 1998).

Current applications do not tap the full potential of QPR as a synthesis tool. There is clearly much work left to be done. Interestingly, some of the areas that are ripe for progress may be the areas where QPR intersects the other techniques described in this chapter. For example, the first section described search-based "design by repair" techniques. These techniques synthesize by searching for a sequence of modification operators to repair an initial candidate design. Qualitative physical reasoning may provide tools for understanding why a design fails so that suitable modification operators can be identified with a minimum of search.

There are a number of ways that QPR can be used in conjunction with machine learning. For example, the third section described systems that learn and reuse design strategies. Qualitative physical reasoning might be able to extend explanation-based learning to this problem area. Recall that explanation-based approaches often perform better than approaches that rely solely on empirical regularities. As mentioned earlier, QPR might also provide a means of extending the capabilities of case-based reasoning systems. The techniques described above in the previous subsection might provide the necessary tools for creating the behavioral and functional models needed for case adaptation.

Another emerging application of QPR is design generalization. A common difficulty with automated synthesis techniques is specifying what is desired. It is often easier to generate an example of what is required than to provide an abstract description of it. Design generalization techniques enable a program to automatically derive the abstract specification from a specific example. The current generalization

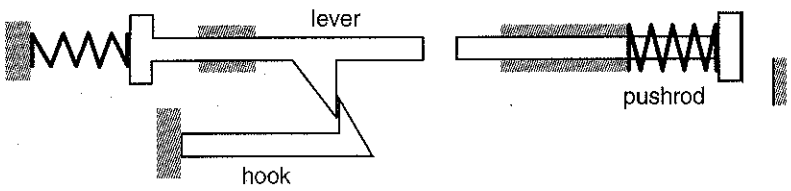


Figure 7.24. A design variant SKETCHIT obtained by replacing a rotating lever with a translating part (Stahovich et al., 1998).

techniques are limited to kinematic/dynamic behavior and structural behavior. There is a need for approaches that can handle other kinds of behavior, including thermal behavior, fluid flow, and the like.

There are a variety of QPR techniques suitable for lumped parameter models; however, there are relatively few techniques that can reason about geometry. Because much of the behavior of a mechanical device is determined by geometry, there is a need for additional qualitative geometric reasoning techniques. Creating these techniques will provide a significant challenge, but the potential benefits are vast.

FINAL CHALLENGES

In this chapter, we have explored the application of AI to design synthesis. We have considered those subdisciplines of AI that are the most relevant to synthesis, namely search, knowledge-based systems, machine learning, and qualitative physical reasoning. The strengths and weaknesses of these techniques were discussed earlier; here we conclude by suggesting a few important areas for future work.

It has been common in AI research to consider devices that are composed of linear sequences of idealized components connected at well-defined ports. Examples include chains of power transmission components and chains of simple mechanisms. These types of devices provide a number of computational advantages. The linear topology helps to restrict the size of the search space, and the discrete nature of the components and interconnections facilitates the construction of behavioral models.

Although these types of devices are used in practice, there are many common, real-world devices that cannot be described as a linear sequence of discrete parts. Many real-world devices have components that are highly interconnected, and the connections between the components often change as the device operates. Furthermore, a given function may be split across multiple components, or a single component may have multiple functions. Consider, for example, a car door, which must provide a means of entering, exiting, and securing the vehicle. The door must also provide structural integrity, aesthetic shape, controls for power mirrors and windows, ducts for defrosters, channels for wires, and so on. In this case, there are no obvious linear sequences of functions or components. In fact, much of the functionality is distributed in a few large pieces of sheet metal. Scaling existing synthesis techniques to handle these kinds of complexities will provide a significant challenge. Part of the solution may be to choose a new ontology, viewing devices as collections of interactions (Williams, 1990; Stahovich et al., 1998) rather than as collections of components.

Many of the automated synthesis techniques described in this chapter are restricted to functions that can be expressed as a desired relationship between two scalar parameters. Examples include transforming a voltage into an angular deflection or transforming a reciprocating linear motion into a continuous angular motion. Although there are useful devices that can be specified in this fashion, many common, real-world devices cannot. Consider a specification of this form: create a device that can bore a hole in a pressurized water pipe and insert a fitting without losing any water.[6] In this case, there are no obvious parameters to be related and hence traditional specification languages would be unsuitable.

[6] Such devices are used to tap into pressurized water mains.

Future work on specification languages should also address the fact that devices typically have multiple states or operating modes, and hence have different functions at different times. For example, a camera has modes for loading film, winding film, cocking, shooting, rewinding, and so on. Work in this area should also be driven by the design of devices with highly interconnected parts and integrated functionality (as described above).

The traditional approach to specification is to describe a desired function in terms of a simple, identifiable piece of behavior, such as a quantity remaining constant or one quantity being proportional to another. It may be possible to extend this approach to more complicated kinds of functions by developing better techniques for characterizing behavior. Alternatively, the "design by example" approach described above may provide a means of specifying a broader range of functions. Rather than attempting to categorize different types of standard behaviors, the"design by example" approach relies on methods for identifying when two behaviors are the same. Identifying similar behaviors may prove more general than creating definitions of behaviors.

Perhaps the most important area for future work is geometric reasoning. Most of the automated synthesis techniques described in this chapter have only limited geometric reasoning abilities. However, for mechanical design, geometry is of primary importance. Much of the behavior of a mechanical device is determined by the geometry of its parts. The geometric reasoning techniques of Joskowicz and Addanki (1988) and Stahovich et al. (1998) are a starting point for this work, but there is much left to be done. Geometric reasoning will likely pose the biggest challenges in applying AI to synthesis. However, success in this area will lead to substantially more powerful and more general synthesis tools.

REFERENCES

Bachant, J. and McDermott, J. (1984). "R1 revisited: four years in the trenches," *The Artificial Intelligence Magazine*, **5**(3):21–32.

Barker, V. E. and O'Connor, D. E. (1989). "Expert systems for configuration at digital: XCON and beyond," *Communications of the ACM*, **32**(3):298–318.

Bennett, J. S. and Englemore, R. S. (1984). "SACON: a knowledge-based consultant for structural analysis." In *AAAI-84*, AAAI Press, Menlo Park, CA.

Bhatta, S. R. and Goel, A. (1997). "Learning generic mechanisms for innovative strategies in adaptive design." *Journal of the Learning Sciences*, **6**(4):367–396.

Bobrow, D. G. and Winograd, T. (1977). "An overview of KRL, a knowledge representation language," *Cognative Science*, **1**(1):3–46.

Brachman, R. J. (1979). "On the epistemological status of sematic networks." In *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Finder (ed.), Academic Press, New York.

Brachman, R. J. and Schmoze, J. G. (1985). "An overview of the KL-ONE knowledge representation system," *Cogntive Science*, **9**(2):171–216.

Britt, D. B. and Glagowski, T. (1996). "Reconstructive derivational analogy: a machine learning approach to automated redesign," *Artificial Intelligence for Engineering Design*, **10**:115–126.

Brown, D. C. and Chandrasekaran, B. (1986). "Knowledge and control for a mechanical design expert system," *IEEE Computers*, **19**(7):92–100.

Buchanan, B. G. and Shortliffe, E. H., Eds. (1984). *Rule-Based Expert Systems. The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley, Reading, MA.

Cagan, J. and Agogino, A. M. (1991a). "Dimensional variable expansion – a formal approach to innovative design,"*Research in Engineering Design*, **3**:75–85.

Cagan, J. and Agogino, A. M. (1991b). "Inducing constraint activity in innovative design," *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing*, **5**(1):47–61.

Chandra, K. S. D. N., Guttal, R., Koning, J., and Narasimhan, S. (1992). "CADET: a case-based synthesis tool for engineering design," *International Journal of Expert Systems*, **4**(2): 157–188.

Cover, T. and Hart, P. (1967). " Nearest neighbor pattern classification,"*IEEE Transactions on Information Theory*, **13**:21–27.

Davis, R. (1982). "Expert systems: Where are we? and Where do we go from here?" *Artificial Intelligence Magazine*, **3**(2):3–22.

Davis, R., Buchanan, B., and Shortliffe, E. (1977). "Production rules as a representation for a knowledge-based consultation program," *Artificial Intelligence*, **8**:15–45.

Davis, R. and Lenat, D. B. (1982). *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York.

Davis, R., Shrobe, H., and Szolovits, P. (1993). "What is a knowledge representation?" *Artificial Intelligence Magazine*, **14**(1):17–33.

de Kleer, J. (1977). "Multiple representations of knowledge in a mechanics problem solver." In *Proceedings IJCAI-77*, Morgan Kaufmann, San Francisco, CA, pp. 290–304.

de Kleer, J. (1979). "Causal and teleological reasoning in circuit recognition," Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge.

de Kleer, J. and Brown, J. S. (1984). "A qualitative physics based on confluences,"*Artificial Intelligence*, **24**:7–83.

Dixon, J. R., Simmons, M. K., and Cohen, P. R. (1984). "An architecture for the application of artificial intelligence to design." In *21st Design Automation Conference*, IEEE Computer Society Press, pp. 634–640.

Domeshek, E. A. and Kolodner, J. L. (1992). "A case-based design aid for architectural design." In *Artificial Intelligence in Design '92*, J. S. Gero (ed.); Kluwer Academic, Dordrecht, The Netherlands, pp. 497–516.

Dym, C. L. (1994). *Engineering Design: A Synthesis of Views*. Cambridge University Press, Cambridge, MA.

Dym, C. L. and Levitt, R. E. (1991). *Knowledge-Based Systems in Engineering*. McGraw-Hill, New York.

Faltings, B. (1990). "Qualitative kinematics in mechanisms," *Artificial Intelligence*, **44**: 89–119.

Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J. (1971). "On generality and problem solving: a case study using the DENDRAL program." In Meltzer, B. and Michie, D., editors, *Machine Intelligence 6*, B. Meltzer and D. Michie (eds.); Edinburgh University Press, pp. 165–189.

Fikes, R. and Kehler, T. (1985). "The role of frame-based representation in reasoning," *Communications of the ACM*, **28**:904–920.

Forbus, K. D. (1980). "Spatial and qualitative aspects of reasoning about motion." In *Proceedings AAAI-80*, AAAI Press, Menlo Park, CA.

Forbus, K. D. (1984). "Qualitative process theory," *Artificial Intelligence*, **24**:85–168.

Forbus, K. D., Nielsen, P., and Faltings, B. (1991). "Qualitative spatial reasoning: the clock project," *Artificial Intelligence*, **51**(3):417–471.

Garrett, Jr., J. H. and Jain, A. (1988). "ENCORE: an object-oriented knowledge-based system for transformer design," *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, **2**(2):123–134.

Gautier, P. O. and Gruber, T. R. (1993). "Generating explanations of device behavior using compositional modeling and causal ordering." In *Eleventh National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA.

Goel, A., Bhatta, S., and Stroulia, E. (1997). "Kritik: an early case-based design system." In *Issues and Applications of Case-Based Reasoning in Design*, M. L. Maher and P. Pu (eds.), Lawrence Erlbaum, Mahwah, NJ, pp. 87–132.

Gruber, T. R. and Gautier, P. O. (1993). "Machine-generated explanations of engineering models: A compositional modeling approach." In *1993 International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Francisco.

Hamilton, S. and Garber, L. (1997). "Deep Blue's hardware-software synergy," *Computer*, **30**(10):29–35.

Hart, T. P., Nillson, N. J., and Raphael, B. (1968). "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, **SSC-4**(2):293–326.

Hayes-Roth, F., Waterman, D. A., and Lenat, D. B. (1983). *Building Expert Systems.* Addison-Wesley, Reading, MA.

Ivezic, N. and Garrett, J. H. Jr. (1998). "Machine learning for simulation-based support of early collaborative design," *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, **12**(2):123–140.

Iwasaki, Y. and Simon, H. A. (1986). "Causality in device behavior," *Artificial Intelligence*, **29**:3–32.

Jamalabad, V. R. and Langrana, N. A. (1998). "A learning shell for iterative design (L'SID): concepts and applications," *Journal of Mechanical Design*, **120**(2):203–209.

Joskowicz, L. and Addanki, S. (1988). "From kinematics to shape: an approach to innovative design." In *Proceedings AAAI-88*, AAAI Press, Menlo Park, CA, pp. 347–352.

Joskowicz, L. and Sacks, E. (1991). "Computational kinematics," *Artificial Intelligence*, **51**: 381–416.

Karnopp, D. and Rosenberg, R. (1975). *System Dynamics: A Unified Approach.* Wiley, New York.

Kedar-Cabelli, S. and McCarty, T. (1987). "Explanation-based generalization as resolution theorem proving." In *Proceedings of the Fourth International Workshop on Machine Learning*, Morgan Kaufmann, San Francisco, CA, pp. 383–389.

Kehler, T. P. and Clemenson, G. D. (1984). "An application development systems for expert systems," *System Software*, **3**(1):212–224.

Kolodner, J. L., Ed. (1993). *Case-Based Reasoning*, Morgan Kaufmann, San Francisco.

Korf, R. E. (1985). "Depth-first iterative deepening: an optimal admissible tree search," *Artificial Intelligence*, **27**(1):97–109.

Korf, R. E. (1988). "Search: a survey of recent results." In *Exploring AI*, H.E. Shrobe (ed.), Morgan Kaufmann, San Francisco, pp. 197–237.

Kota, S., Erdman, A. G., Riley, D. R., Esterline, A., and Slagle, J. (1987). "An expert system for initial selection of dwell linkages." In *ASME Design Automatiion Conference*, ASME, New York.

Kuipers, B. (1986). "Qualitative simulation," *Artificial Intelligence*, **29**:289–388.

Mackworth, A. K. (1977). "Consistency in networks of relations," *Artificial Intelligence*, **8**(1):99–118.

Mahadevan, S., Mitchell, T. M. Mostow, J., Steinberg, L., and Tadepalli, P. V. (1993). "An apprentice-based approach to knowledge acquisition," *Artificial Intelligence*, **64**: 1–52.

Maher, M. L., Balachandran, B., and Zhang, D. M. (1995). *Case-Based Reasoning in Design*, Lawrence Erlbaum, Mahwah, NJ.

McDermott, J. (1981). "R1, the formative years," *Artificial Intelligence Magazine*, **2**(2): 21–29.

Mitchell, T. M. (1977). "Version spaces: a candidate elimination approach to rule learning." In *Fifth International Joint Conference on AI*, Morgan Kaufmann, San Francisco, CA, pp. 305–310.

Mitchell, T. M. (1982). "Generalization as search," *Artificial Intelligence*, **18**(2):203–226.

Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill, New York.

Mitchell, T. M., Keller, R., and Kedar-Cabelli, S. (1986). "Explanation-based generalization: a unifying view," *Machine Learning*, **1**(1):47–80.

Mittal, S. and Dym, C. L. (1986). "PRIDE: an expert system for the design of paper handling systems," *IEEE Computers*, **19**(7):102–114.

Mostow, J., Barley, M., and Weinrich, T. (1989). "Automated reuse of design plans," *Aritificial Intelligence in Engineering*, **4**(4):181–196.

Newell, A., Simon, H. A., and Shaw, J. C. (1963). "Empirical explorations with the logic theory machine: a case study in heuristics." In *Computers and Thought*, E. Feigenbaum, and J. Feldman (eds.), McGraw-Hill, New York.

Orelup, M. F. and Dixon, J. R. (1987), "Dominic II: More progress towards domain independent design by iterative redesign." In *Proceedings Intelligent and Integrated Manufacturing Analysis and Synthesis*, Winter Annual Meeting of the ASME, 67–80.

Papalambros, P. and Wilde, D. J. (1988). *Principles of Optimal Design*. Cambridge University Press, Cambridge.

Parker, D. (1985). "Learning logic," Technical Report TR-47, MIT, Cambridge; MA.

Quinlann, J. R. (1986). "Induction of decision trees," *Machine Learning*, **1**(1):81–106.

Quinlann, J. R. (1993). *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Mateo, CA.

Rosen, D., Riley, D., and Erdman, A. (1991). "A knowledge based dwell mechanism assistant designer," *Journal of Mechanical Design*, **113**:205–212.

Rumelhart, D. E. and McClelland, J. L. (1986). *Parallel Distributed Processing: Exploration in the Microstructure of Cogntion*. Vols. 1 and 2, MIT Press, Cambridge, MA.

Russell, S. and Norvig, P. (1994). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs; NJ.

Schwabacher, M., Ellman, T., and Hirsh, H. (1998). "Learning to set up numerical optimizations of engineering designs," *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, **12**(2):173–192.

Shea, K., Cagan, J., and Fenves, S. J. (1997) "A shape annealing approach to optimal truss design with dynamic grouping of members," *ASME Journal of Mechanical Design*, **119**(3): 388–394.

Shortliffe, E. H. (1974). "MYCIN: a rule-based computer program for advising physicians regarding antimicrobial therapy selection," Ph.D. Dissertation, Stanford, University, Stanford, CA.

Shrobe, H. (1993). "Understanding linkages." In *Proceedings AAAI-93*, AAAI Press, Menlo Park, CA, pp. 620–625.

Simmons, R. and Davis, R. (1987). "Generate, test, and debug: combining associatational rules and causal models." In *International Joint Conferences on Artificial Intelligence*, Morgan Kaufmann, San Francisco, pp. 1071–1078.

Stahovich, T. F. (2000). "LearnIT: an instance-based approach to learning and reusing design strategies," *Journal of Mechanical Design*, **122**(3):249–256.

Stahovich, T. F. and Bal, H. (2001). "LearnIT II: an inductive approach to learning and reusing design strategies," *Research in Engineering Design*, submitted.

Stahovich, T. F., Davis, R., and Shrobe, H. (1996). "Generating multiple new designs from a sketch." In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, AAAI Press, Menlo Park CA, pp. 1022–1029.

Stahovich, T. F., Davis, R., and Shrobe, H. (1997). "Qualitative rigid body mechanics." In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA.

Stahovich, T. F., Davis, R., and Shrobe, H. (1998). "Generating multiple new designs from a sketch," *Artificial Intelligence*, **104**:211–264.

Stahovich, T. F., Davis, R., and Shrobe, H. (2000). "Qualitative rigid-body dynamics," *Artificial Intelligence*, **119**:19–60.

Stahovich, T. F. and Kara, L. B. (2001). "A representation for comparing simulations and computing the purpose of geometric features," *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, **15**(2):189–201.

Stahovich, T. F. and Raghavan, A. (2000). "Computing design rationales by interpreting simulations," *Journal of Mechanical Design*, **122**(1):77–82.

Stallman, R. M. and Sussman, G. J. (1976). "Forward reasoning and dependency-directed

backtracking in a system for computer-aided circuit analysis," Technical Report Memo 380, AI Lab, MIT, Cambridge, MA.

Stefik, M., Bobrow, D. G., Mittal, S., and Conway, L. (1983). "Knowledge programming in LOOPS: report on an experimental course," *Artificial Intelligence*, **4**(3):3–14.

Subrahmanyam, P. A. (1986). "Synapse: An expert system for VLSI design," *IEEE Computers*, **19**(7):78–89.

Subramanian, D. and Wang, C.-S. E. (1993). "Kinematic synthesis with configuration spaces." In *The 7th International Workshop on Qualitative Reasoning about Physical Systems*, pp. 228–239.

Subramanian, D. and Wang, C.-S. E. (1995). "Kinematic synthesis with configuration spaces," *Research in Engineering Design*, **7**:193–213.

Ulrich, K. T. (1988). "Computation and pre-parametric design," Technical Report 1043, AI Lab, MIT, Cambridge, MA.

van de Brug, A., Brachant, J., and McDermott, J. (1986). "The taming of R1," *IEEE Expert*, **1**(3):33–39.

van Melle, W., Shortliffe, E. H., and Buchanan, B. G. (1984). "Emycin: a knowledge engineer's tool for constructing rulebased expert systems." In *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. E. H. Shortliffe and B. G. Buchanan (eds.), Addison-Wesley, Reading, MA.

Waltz, D. (1975). "Understanding line drawings of scenes with shadows." In *Psychology of Computer Vision*, P. H. Winston (ed.), MIT Press, Cambridge, MA.

Williams, B. C. (1984). "Qualitative analysis of MOS circuits," *Artificial Intelligence*, **24**(1–3):281–346.

Williams, B. C. (1990). "Interaction-based invention: designing novel devices from first principles." In *AAAI-90*, AAAI Press, Menlo Park, CA.

Winston, P., Binford, T., Katz, B., and Lawry, M. (1983). "Learning physical descriptions from functional definitions, examples, and precedents." In *Proceedings of the National Conference on Artificial Intelligence*, AAAI Press, Menlo Park, CA, pp. 433–439.

Winston, P. H. (1992). *Artificial Intelligence*. 3rd ed., Addision-Wesley, Reading, MA.