

Sanjay Mittal and Agustin Araya
Knowledge Systems Area
Intelligent Systems Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA. 94304

Abstract:

Many design problems can be formulated as a process of searching a "well-defined" space of artifacts with similar functionality. The dimensions of such spaces are largely known and are constrained by relations obtained from the implicit functionality of the designed artifact. After identifying the kinds of knowledge that mediate the search for acceptable designs, a computational framework is presented that organizes the required knowledge as design plans. A problem solver is described that executes these plans. The problem solver extends the notion of dependency-directed backtracking with an advice mechanism. This mechanism allows information from a constraint failure to be used as advice in modifying a partial design. An expert system for designing paper transports inside copiers has been successfully built based on this framework.

1. Introduction

Increasing attention is being paid to the development of knowledge-based systems for design, especially of mechanical systems [Dym 1985, Gero 1985]. The expectation is that these computer systems can improve the quality of designs and shorten the time required to find satisfactory designs.

Some of the major stages in designing a complex system are: i) a definition stage where precise functional specifications are developed from the requirements; ii) a generation stage where many satisfactory designs may be created; and iii) an evaluation stage where these different designs are compared or optimized by some criteria. These stages are not necessarily sequential because the latter stages can provide feedback to earlier ones. In this paper we shall be primarily concerned with the middle stage, i.e., the generation of designs that satisfy some functional specification.

The general problem of designing artifacts that satisfy some arbitrary functionality is not well understood [Mostow 85]. However, there seem to be many design problems where the search space has been largely defined by the expert designers (or can be obtained from them). This means that the kinds of dimensions of the design space are by and large known, i.e., the kinds of design parameters are known. Furthermore, the design parameters of the search space are constrained to produce artifacts which have the "same" functionality. We shall call problems with these two properties as being *well-defined*.

In this paper we present a framework for building computer programs that can assist in the design of systems that have well-defined search spaces. The framework rests on the key observation that given such spaces, the process of generating alternative designs is largely a process of searching these spaces. This is not to suggest that the space is small, or that it does not vary in details, or that substantial reasoning may not be needed for finding satisfactory designs. On the contrary, the search process is guided by knowledge about how to define partial designs in this space and knowledge about how to modify a partial design when the constraints are violated. Furthermore, the search may be ordered by heuristic knowledge obtained from experience.

The proposed framework organizes these different kinds of knowledge into design plans. These plans are carried out by a problem solver that can engage in exhaustive search if the knowledge is insufficient. The problem solver extends the notions of dependency-directed backtracking with an advice mechanism. This mechanism allows advice based on a failed constraint to reorder the generators at a prior decision point allowing rapid convergence in many cases.

Based on this framework we have successfully built an expert system called PRIDE [Mittal et. al. 1986] for the design of paper transports inside copiers. In this paper we shall focus on the ideas behind the design framework and not the expert system itself. We start by describing an example of an artifact with a well-defined design space. The next section makes our notion of design-as-search more precise. The subsequent three sections describe the framework itself. We conclude with a discussion of some of the questions raised by our work.

2. Knowledge about the Artifact Being Designed

We begin with a simplified example taken from the domain of paper handling systems inside copiers and duplicators.

An example of an artifact

A paper handling system in a copier is used to transport paper from an input to an output location, avoiding certain obstructions. One kind of paper transports are built from the pinch-roll technology. In this technology, a "baffle" is used to guide the paper along a certain path and "roll stations" are placed along this path to move the paper (see Figure 1). Roll stations consist of one or more pairs of rolls mounted in corresponding shafts. Each pair, in turn, consists of a driver roll, which is powered, and an idle roll, which spins freely.

A typical design problem specifies the velocity and angle of the paper at the input and output locations of the transport, maximum acceptable skew of the paper while being transported, characteristics of the papers that will be transported (e.g., length, weight, etc), and so on. The problem is to determine the shape of the baffle, the number, position and kinds of roll stations, the properties of drivers and idlers, and many other properties of these and other components.

Different kinds of knowledge

There might be several kinds of artifacts, based on different technologies, that can exhibit the "same" functionality. For instance, paper transports can also be built from belt-transport technology.

For each technology, it is necessary to know the kinds, and numbers, of parts (or components) and how those parts compose or interact to form the artifact. Parts might be further decomposed into other parts.

Certain parts might have alternative decompositions into subparts, and it is necessary to know the conditions under which each alternative is more suitable.

Parts have "relevant" properties, i.e., properties that can affect the functionality of the artifact. (e.g. width and diameter of a driver roll, which may affect the velocity with which the paper moves while passing through the station). When parts interact with other parts of an artifact, they can exhibit certain relevant behaviors (e.g., velocity of a driver, skew of the paper), which depend on properties and behaviors of these or other parts.

Corresponding to each property, one needs to know what the plausible values are for that property, e.g., the different known diameters of a drive roll may be 10, 20, 40 mm; the width of a driver can be between 5mm and 50mm in increments of 1mm; the baffle gap can be between 2 and 10mm in increments of 0.5 mm; etc. Certain properties of parts can only take values from a pre-existing set of values. This is the case when it is desirable to select parts from existing ones. For other properties it might be known how to design them taking into account the given specifications and the properties and behaviors of other parts.

3. Design as Knowledge-guided search

The process of designing such an artifact can be usefully viewed as a search of a multi-dimensional space of possible designs. The dimensions of such a space are the parameters of the artifact, i.e., the structural relationships between the parts and the properties of the individual parts. For example, in the case of a paper transport, some of the dimensions would be "input velocity of the paper coming into the transport", "lengths and widths of the different kinds of paper", "length of the paper path", physical characteristics of each of the driver and idler roll at each station such as diameter, width, material, and velocity, etc.

Typically such design spaces are very large and searching for suitable designs can be very time consuming. Two major factors contribute to this. First, significant computation may be involved in defining a point in the space, i.e., assigning values to the different parameters. Because the space is quite sparse, in that there are far fewer acceptable designs than the ones ultimately rejected, most of the search effort may be expended in finding solutions that will be rejected later on. One approach to mitigate this problem is to analyze partial designs as early as possible, instead of waiting

for the complete design.

This brings us to the second cost, i.e., the computation in evaluating a design for suitability. Many of the analysis techniques are time-consuming and a design may pass one analysis only to be rejected later by another one. By appropriately ordering the generation of the design and its evaluation for suitability, some of the wasteful computation may be avoided.

Given this complexity, experienced designers use knowledge of various kinds to direct their search. As discussed in the previous section, one obviously needs to have a great deal of knowledge about the artifact itself. Here we will discuss some of the knowledge used in exploring the space and directing the search.

Ordering Knowledge.

A simple, yet powerful piece of knowledge is information that creates an order in which decisions get made. Use of such ordering information is quite prevalent [Mostow 85]. However, the characteristics of the search space which create such order are not well understood. The ordering knowledge may be simply based on the dependencies between decisions. For example, in our example problem, decisions about roll station placement depend so intrinsically on the length of the paper path that they have to be made later.

A different kind of order is created by structuring the space hierarchically. By this we mean that instead of having the complete space explicitly defined, decisions along some dimension open up sub-spaces. Thus, different choices at some level could lead to very different sub-spaces being opened up for design. A simple example from paper transport domain involves choice of technology. Depending on the technology chosen such as rolls or belts, very different design spaces are opened up for further exploration.

Constraints between parameters.

The parameters of the design artifact are not independent. Often, they are constrained by relations. Some of these constraints may be derived from the explicit specifications of the particular design problem. For example, the locations and angles of the input and output of the paper transport constrain the shape of the paper path.

A different set of constraints is derived from the intrinsic properties of the structure and behavior of the artifact being designed. All paper transports must satisfy some basic constraints on velocities, frictions,

and forces acting on a moving paper, otherwise they will fail in their essential functionality. For example, the distance between two consecutive roll stations must be less than the smallest paper that will be transported by the paper handling system, otherwise for certain sections of the path the paper will no longer be under the control of any station. Both kinds of constraints determine the suitability of a design in terms of providing the desired functionality.

The way these constraints are used is crucial in determining how efficiently the design process operates. It is well known that a generate and test model in which the constraints are primarily used to test the generated solutions will be quite inefficient. More powerful problem solvers such as dependency-directed backtracking [deKleer et al. 79] also have some well-known deficiencies. Some of these deficiencies can be compensated by using appropriate knowledge, in terms of "ordering" information based on how the variables are constrained.

We have found it useful to make a distinction between tight and loose coupling between a set of variables. In the case of tightly coupled variables, a search procedure that tries to assign a value to one of these variables and then propagate it over the constraints may have to back up many times before finding a consistent solution. However, in the case of loosely coupled variables, it is often possible to find a partial order in which the variables are decided which will work with relatively small amounts of backtracking.

Advice for Modification.

A major piece of knowledge that expert designers seem to use when the design fails some acceptability condition (constraint) is how to modify the design. Consider a dependency-directed backtracking problem solver in contrast. It knows enough to back up to a relevant decision point but does not have any way of deciding how to modify its decision. Good designers, on the other hand, not only know where the relevant prior decision points are but also analyse the failure to decide how to modify their past decisions. Being able to advise a prior decision point (and a problem solver in general) is crucial in reducing the search. In the best case, the advice would enable a previous decision to be modified in exactly the way needed to fix the current constraint failure. In general, the advice may only help partially. In the framework we have developed, and described in the rest of the paper, this ability to advise plays a central role in problem solving and is an important

advance over most of the earlier approaches.

4. Structuring Design Knowledge as Plans

In the previous section we identified four major kinds of knowledge that are needed during the design process: defining the dimensions of the design space; choices along each dimension; constraints on these choices; and advice for modifying some design choice. In addition, there were heuristics on ordering the decisions, structuring the space, and ordering the choices for some dimension that aid in making the design process be more effective. These different pieces of knowledge can be effectively integrated into knowledge structures that we shall call design plans. In this section we introduce the different plan elements and describe their structure. The next section discusses how they are used in problem solving.

Goals.

Plans are organized around *goals* for making design decisions about a set of design parameters. Each goal is responsible for a few of these parameters, i.e., it represents one or more decision points from a problem solving viewpoint. A goal also defines some of the dimensions of the design space. By this we mean that only by scheduling a goal does the design sub-space defined by that goal become ready for exploration.

In our paper transport domain, some typical goals would be "Design Paper Transport", "Design Paper Path", "Design Driver Roll", and "Design Driver Width". The first of these is a top-level goal, which can recursively expand into a tree of sub-goals (Figure 2). Each of these goals defines a space of partial designs. As we move down the goal tree fewer dimensions are considered. Thus, the goal "Design Driver Width" is concerned with only one design parameter, whereas the goal "Design Driver Roll" is concerned with all parameters of a driver roll. The former is a sub-goal of the latter. Each goal explicitly specifies the design parameters it is responsible for. Goals also specify the design parameters on which they depend. For example, the goal "Decide number and location of roll stations" specifies that it depends on knowing the paper path length. The dependency information may be either statically described or dynamically determined from the particular design method that is being tried or both.

Design Methods.

Design goals have different design methods associated with them, which specify alternate ways to make decisions about the design parameters of the goal.

These methods capture the knowledge about the possible values of properties of components, as well as knowledge about the behavior of components. The role of the design methods is then to generate partial designs.

The knowledge about carrying out a goal may be available in many different forms. This diversity is reflected by the different kinds of methods that exist in our representation. One kind of methods are *generators* which specify a set, or range of values to be generated. They can also encode heuristics about ordering the values, initial guesses, etc. For example, a generator method for driver width is shown in figure 3. It shows both the range of values as well as the initial choice heuristic.

Another kind of methods are *calculations* which apply some mathematical function over a set of previously decided parameters. A calculation may be viewed as a combination of a generator and an equality constraint. This method always produces the same value for the same set of its input parameter values. Some of the other kinds of methods are *procedures* (which embed arbitrary computations) and *constrained generators* (which can look ahead to the constraints on the goal to generate values).

There is another set of method types which primarily provide control knowledge on the use of other methods. A simple example are *conditional methods* (also called rules) which allow some conditions to be specified on the suitability of applying a method. The action part of a rule must be a method. Other examples of such control methods are *rule groups* and *conjunctive methods*. An important property of control methods is that they make explicit the separation between two kinds of knowledge: one for making design choices and the other for selecting a suitable set of choices or ordering the different sets of choices.

Sub plans. Another kind of control method is called a *subplan*. These methods specify a set of goals that must be carried out in order to satisfy the higher level goal. The actual order in which the goals are carried out is specified by the input and output dependency descriptions attached to a goal. The subplan method is the only mechanism for creating goal trees. This has some important consequences. First, alternate plans for decomposing a goal into sub-goals can be easily represented. For example, very different sub-plans exist for a goal if different technologies are available for the implementation of the goal's specifications. Second,

given that a subplan method is like any other method, it can be embedded inside control methods. This allows, for example, plan selection knowledge to be represented inside control methods.

Finally, subplan methods and other more direct methods can be simultaneously specified for the same goal. In other words, a goal may be achieved in different ways. One way may be to decompose it into smaller sub-problems. Another way might be to use previously designed pre-packaged solutions. For example, the goal for "Design driver roll" may have one method which decomposes the goal into sub-goals: "design diameter", "design width", "decide tolerances", "decide material", etc. A driver designed in this way may need to be manufactured from raw stock. Another method may be a generator which selects from some standard off-the-shelf driver rolls. Typically, this latter method would be tried before the more general subplan and be so specified.

Statically no distinction can be made between goals which have sub-goals and those which have direct methods. During the execution of the plan, however, some differences arise. The primary difference arises from the fact that a sub-goal is responsible for a subset of the specifications of its super-goal. In such cases, the most specific goal is held responsible for the shared design parameter during problem-solving, which is described in the next section.

In addition to the method types described above, we also specify an abstract problem solving protocol that must be followed by a method. Thus, new method types can be created. In fact, the current set has evolved over the course of representing the knowledge about paper transports.

Design Constraints.

The third major element of a plan are constraints on the design parameters. These constraints are attached to some goal. Typically, they would be associated with the goal for the less constrained variable, as heuristically determined by experts. However, they can be as well attached on separate goals which then depend on the goals for the constrained parameters. Notice, that much of the ordering in the plan arises from where the constraints are attached. This is because the parameters in a constraint are also used to order the goal during run-time scheduling. As we discussed in the previous section, this is very appropriate because much of the ordering seems to come from the constraints on a parameter.

We view a constraint as an object which basically specifies a relation between a set of design parameters. These relationships may reflect the conditions on the underlying structure or behavior of the artifact or they may be derived from the specifications of an individual problem. In the next section we elaborate on how constraints are used.

Advice for modification.

The last major element of a design plan is advice to the problem solver. We have identified the need for many different kinds of advice. In this paper we will focus on only one kind of advice, namely, *modify parameter advice*. This is the advice attached to constraints and activated when constraints fail. These advice descriptions can be obtained in two ways. For certain kinds of constraints one can analyze the expression and determine which parameters must be modified and how to satisfy the constraint. In many other cases, the experts know from experience which parameter may be more easily modifiable and the system can determine how much to change the parameters in order to satisfy the constraint.

In our framework we can represent both kinds of advice. This implies that part of the constraint protocol is being able to automatically analyze the failure. Once a piece of advice is created, no difference is made between the heuristic (produced by the expert) and direct (produced by the system) advice.

Some of the other kinds of advice we have found useful are *processing advice* which advises the problem solver itself to give up or suspend a particular exploration path; *selection advice* which causes a particular plan to be aborted in favor of another; and *modify specification advice* which advises the user (or another system) to change some problem specification.

5. Problem Solving using the Plans

We start by describing the basic problem solver that tries to carry out these design plans. Later we will briefly describe the more extended version which supports a more comprehensive design process. The basic problem solver comprises three major parts: i) a goal scheduler which uses an agenda to post goals, try them out, suspend them if needed, and revise them; ii) a dependency net which is created dynamically (this data structure associates a designed parameter with the goal which designed it and the goals which directly depend on it); and iii) a set of protocols which each of the plan elements is expected to follow. The protocols can be viewed as falling in two groups: *initial design* and

revision.

Initial Design Protocol.

Before a goal is run, its preconditions are checked. These are computed both from the input parameter dependencies as well as direct dependency on other goals. The latter is a heuristic way of ordering goals which reflects processing considerations.

The activated goal tries methods from its list of design methods to find the first that runs successfully. A method could cause a goal to suspend by surfacing some new dependencies. Most methods fail or succeed right away. Subplan methods, on the other hand, post new goals and suspend the higher goal. If all methods fail, then the goal fails. Notice, that if the goal was embedded in a subplan method, and all but the top goal are, this failure propagates to the method and up.

Once a method succeeds, the constraints are tried. If all constraints are satisfied, the goal succeeds. If a constraint fails, however, the problem solver (often working with the user) will either relax the constraint or try to satisfy it by revising the partial design.

Revise Design Protocol.

In order to revise the design the problem solver has to: i) determine what design parameter(s) to modify, ii) determine which goal to backtrack to, and iii) try to effect the change. The first piece of information comes from the advice attached to constraints. Given the advice, the dependency net is examined to determine the goal which can handle the advice. This goal is then activated in a "revise" state.

The revised goal adds the advice as a new constraint. It then asks the previously executed method to revise itself if it can. Different methods handle advice differently. A generator tries to generate a different value which conforms to the advice. A calculation, on the other hand, can revise itself only by creating a new piece of advice which may cause the problem solver to backup further. If the original method fails, then the goal searches among its other methods for the first method that succeeds. If none of the methods succeed then the advice has failed and control returns to the original point of failure. Often there are other pieces of advice that can be tried. If a method does succeed in producing a value then the constraints are checked again. If the constraints are satisfied then the advice has succeeded and design will proceed, eventually reaching the goal which originally failed and continuing beyond if the advice was appropriate.

Notice that at the revised goal, some constraints which originally succeeded may now fail. This can create new advice causing the problem solver to back up further. Also, some new constraints may have been added which can fail. In fact the calculation methods effectively propagate the advice backwards by this mechanism.

Illustration of the advice mechanism

We shall illustrate how the advice mechanism works with the help of a simple example. Consider the following two constraints on three variables x , y , and z .

$$x + y + z > 10 \quad (C1)$$

$$x + y + z < 20 \quad (C2)$$

Furthermore, let us assume that independent of these constraints, we also know the sets from which each of the three variables can take values.

$$x: \{1, 3, 5\} \quad (4)$$

$$y: \{2, 4, 6, 8\} \quad (5)$$

$$z: \{1 \dots 100\} \quad (6)$$

One way to represent this problem in our framework is to have separate goals for x , y , and z . Let us call them G_x , G_y , and G_z . Each of these goals will have a single method, which is a generator incorporating the choice sets in (4) - (6) respectively. Let us name the methods M_x , M_y , and M_z . Also assume that there is no knowledge about initial guesses for these variables in the generators. Constraints $C1$ and $C2$ can be either attached to one of these goals or a fourth one. Let us say we adopt the latter representation and call the goal with the constraints G_c . [A discussion of the differences between the two choices are beyond the scope of this paper.]

In the initial design phase, the goals G_x , G_y , and G_z will be trivially satisfied (because no constraints are attached to them) by making the following choices.

$$x = 1; y = 2; \text{ and } z = 1$$

However, goal G_c will fail because while $C2$ is satisfied, $C1$ is not. Constraint $C1$ can generate many different advice for modification:

$$x \uparrow, > 7 \quad (A1)$$

$$y \uparrow, > 8 \quad (A2)$$

$$z \uparrow, > 7 \quad (A3)$$

$$x \uparrow \ \& \ y \uparrow \ (A4), \text{ etc.}$$

The advice $A1$ means "increase x such that it is greater than 7". In this example, we will only consider advice

that tries to change one variable at a time. The advice A1 when sent to the problem solver will cause goal Gx to try to revise itself. However, the method Mx at Gx cannot find a value for x that is greater than 7, so this advice will fail. Goal Gc will then send advice A2, which also fails. Next A3 is tried which succeeds in modifying z to 8 and now the constraints are satisfied.

Notice that the revision of z will cause all goals dependent on z to be "undone" and retried. Also, even though we started with arbitrary values for the three variables, we were able to quickly find a solution. The generators keep track of the choices they have made, so the same value will not be generated again in the same context (see section 6 for more on the context mechanism).

Suppose we were to impose a new constraint on z at this point:

$$z > 10 \quad (C3)$$

This constraint will fail creating an advice,

$$z \uparrow, > 10 \quad (A5)$$

This advice will cause the value of z to change to 11. The change in z will undo goal Gc which will recheck its constraints. The constraints C1 and C2 are still satisfied, so this new solution will be accepted. Notice, that if wanted to preserve the previous solution, this new constraint would be imposed in a subcontext, allowing both solutions to be explored further.

Example of design revision from Pride

Let us consider another example which is drawn from the paper transport domain. After the shape of the path to be followed by the paper has been defined, it is necessary to determine the number of roll stations and their locations. The placement of the stations has to satisfy various kinds of constraints [Mittal and Stefik 86].

In the design phase, a heuristic is used to propose the number of stations. Using this information, a method is applied which determines ranges of placements of stations such that the relevant constraints are satisfied. If it turns out that no such placement exist because for any placements there are constraints that are not satisfied, then a redesign episode takes place. A piece of advice is generated indicating, for instance, that the number of roll stations should be increased. This requires undoing the previous decision (and all the decisions that depended on it) and making a new decision using the advice. This is illustrated in figure 4.

Discussion.

Some important properties of our problem solver are novel and crucial to its success. Our problem solver augments a weak-method, i.e., dependency-directed backtracking, with an advice mechanism. In other words, the dependencies between design parameters are used in determining a relevant decision point to back up to. Furthermore, the failed constraint(s) is analyzed to determine a piece of advice for the revised decision. *Thus the problem solver is not only capable of searching its entire design space but still does so intelligently and directed by advice from failures.* Moreover, this general search method is integrated in a framework which is knowledge-rich. This means that if knowledge exists about ordering goals or making plausible choices, it can be profitably used. Recourse is made to the general method only where sufficient knowledge does not exist or is incomplete.

Finally, notice that our approach avoids another typical shortcoming of purely knowledge-based approaches which rely on heuristically determined order between goals. In our scheme even if two goals were ordered the wrong way, the advice mechanism would produce the correct result in one round of revision. This is because the advice mechanism allows constraints imposed later in design to be propagated back as advice. The same mechanism can also be used to do a rough design followed by a more precise design.

Limitation.

Even though the problem solver we have described can perform arbitrary search, it will clearly be too inefficient in some cases. One such situation arises in cases of tightly coupled variables. That is, if there is a set of variables which are so inter-constrained that no local propagation of values or advice will suffice to efficiently find a consistent solution, then one might want to look for other problem solving methods for that subproblem. For example, in the paper transport design, the roll placement problem has this property. It is important to emphasize that these special problem solvers can still be embedded in our overall framework by embedding them inside design methods. The example discussed earlier illustrated this point. *This implies that the overall problem solving may still proceed as a process of solving loosely-coupled sub-problems with some backtracking, with the tightly-coupled decisions localized as a single decision-point, but still capable of being revised from the outside.*

6. Extended Problem Solver

We briefly describe two other components of the problem solver that play a major role in supporting the overall design process but are not essential in understanding how the problem solver works.

Multiple design contexts.

We provide a facility for maintaining multiple design contexts [Mittal et. al. 1986]. A design context contains a complete description of the artifact being designed, a complete description of the state of the design plan corresponding to that design, and the state of the problem solver.

The advising mechanism makes use of the multiple contexts mechanism. Specifically, when the design problem solver processes an advice, it can do so in a separate context. This ensures that if a specific advice fails to revise the design satisfactorily, the system can back up to the context in which the advice was originated and continue with a different advice.

The ability to create multiple partial designs and keep them distinct is crucial in exploring different choices simultaneously. For example, at certain choice points, one can explore the different choices simultaneously by creating a sub-context for each choice. We have chosen not to do so because of the size of the design space, i.e., the number of choice points and choices at each point are far too many. Ultimately, some incorporation of ATMS [deKleer 1986] ideas may be worthwhile.

User control of the search.

Pragmatically, the user and the automated problem solver have to work together. This is because of the complementary nature of their strengths. Most automated problem solvers can tirelessly search a design space, manage the dependencies, selectively undo parts of the design, and consistently check the constraints. However, they rarely have enough knowledge to avoid unnecessary work. Human problem solvers, including experts, are rarely systematic in the above activities, but often have knowledge that lets them avoid or minimize the search. It seems natural, therefore, that there be a way for the human user to steer the problem solver in more suitable regions of the search space.

We provide many entry points for a user to interact with the problem solver. The advice mechanism turns out to be quite suitable for many such interactions. Thus, a user can easily enter a piece of advice. This means

that the user can choose to advise arbitrary goals and thereby affect the course of design.

Another natural place is in the selection of advice. A failed constraint typically has alternative advices on how to satisfy it. However, it is often hard for the system to decide which advice is more likely to succeed. We allow the user to not only change the order of the advice but also change its content in some cases.

There are many situations where the design methods are incomplete in their description of the design space. In such situations, it is natural for the user to be able to make a design decision and let the system do the rest. In fact it is possible for the user to not only make the decision but also handle the ensuing advice from a constraint failure at some subsequent goal. On a very pragmatic basis, these 'hooks', along with the multiple context facility, allow a user to work with the system in exploring a design space and looking at alternatives quite rapidly.

7. Discussion and Conclusions

The framework described in this paper has been successfully used to build a knowledge-based system, called *Pride*, for designing paper transports inside copiers and duplicators [Mittal et. al. 1986]. A prototype version of *Pride* has been ready and in field test for over a year now. It has been tested on real design problems from previous and ongoing copier projects. It has been successful in not only producing acceptable designs but also in analyzing designs produced by engineers and identifying shortcomings in their designs.

The notion of plans for representing design knowledge was independently developed by Brown and Chandrasekaran [Brown and Chandrasekaran 1985]. Our framework, however, is more general in many ways. First, we impose fewer restrictions on the kinds of artifacts we can handle. Second, we provide a problem solver that can search the design space more thoroughly. Finally, our multiple contexts mechanism allows different design alternatives to be simultaneously explored.

Many interesting research issues are still unresolved in the work we have presented. For example, we have not explored the limitations of the advice mechanism. In particular, we have not looked at the general case where many constraints can simultaneously fail and the problem caused by conflicting advice. Another area of investigation is a categorization of constraint types and the constraint satisfaction methods that may be most suitable for each

type.

Another interesting issue we are investigating is the relationship between the structure and function of the artifact on one hand and the design plans on the other. This seems to be important both from the point of view of acquiring additional knowledge as well as generating the design plans more automatically.

As was indicated in the introduction, the proposed framework supports the "generation of alternative designs" stage of the overall design process. We are trying to extend the framework to cover the other stages also. In particular, we want to study the processes involved in the comparison of designs according to a set of criteria. Also, we want to extend the advice mechanism to support the feedback processes between the different stages.

Acknowledgements

The Pride project is a joint effort between Xerox PARC and Xerox RBG (Reprographics Business Group). Mahesh Morjaria, George Roller and many other engineers at RBG have collaborated on this project from the start. Felix Frayman has contributed many ideas and programming effort to the project. Mark Stefik has supported the work both as the manager of Knowledge Systems Area at PARC and as a research colleague. Daniel Bobrow, Felix Frayman, Ken Kahn, Mark Stefik, and the referees provided invaluable feedback on earlier drafts of the paper.

References

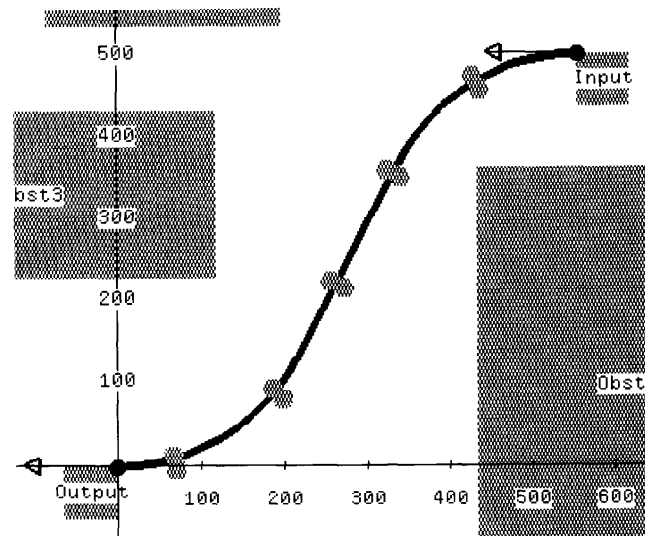
- Brown, D. C., and B. Chandrasekaran. "Expert Systems for a Class of Mechanical Design Activity". In J. Gero, ed., *Knowledge Engineering in Computer-Aided Design*. North Holland, Amsterdam, 1985.
- Dym, C. L., (ed). *Applications of Knowledge-Based Systems to Engineering Analysis and Design*. ASME, New York, 1985.
- Gero, J., (ed). *Knowledge Engineering in Computer-Aided Design*. North Holland, Amsterdam, 1985.
- de Kleer, J., J. Doyle, G. L. Steele, and G. J. Sussman. "Explicit Control of Reasoning". In P. H. Winston and R. H. Brown, eds., *Artificial Intelligence: An MIT Perspective*, MIT Press, Cambridge, 1979.
- de Kleer, J. "An Assumption-based TMS". *Artificial Intelligence* 28:2 (1986) 127-162.

Mittal, S., C. L. Dym, and M. Morjaria. "PRIDE: An Expert System for the Design of Paper Handling Systems." To appear in *Computer* (Spl. Issue on Expert Systems for Engineering Applications). July, 1986.

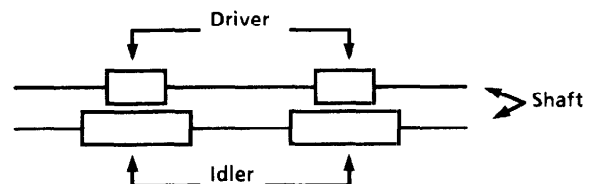
Mittal, S., and M. J. Stefik. "Constraint Compaction: Managing Computational Resources for Efficient Search." Technical memo, Xerox Palo Alto Research Center, Palo Alto, CA, April, 1986.

Mittal, S., D. G. Bobrow, and K. Kahn. "Virtual Copies: At the boundary between classes and instances." To appear in *Proc. ACM Conf. on Object-Oriented Programming Languages, Systems and Applications (OOPSLA)*. Portland, Oregon, September, 1986.

Mostow, J. "Towards Better Models of the Design Process." *AI Magazine*, Spring 1985.



a) Side view of paper path and roll stations



b) Front view of roll station

FIG 1: Views of a Paper Handling System

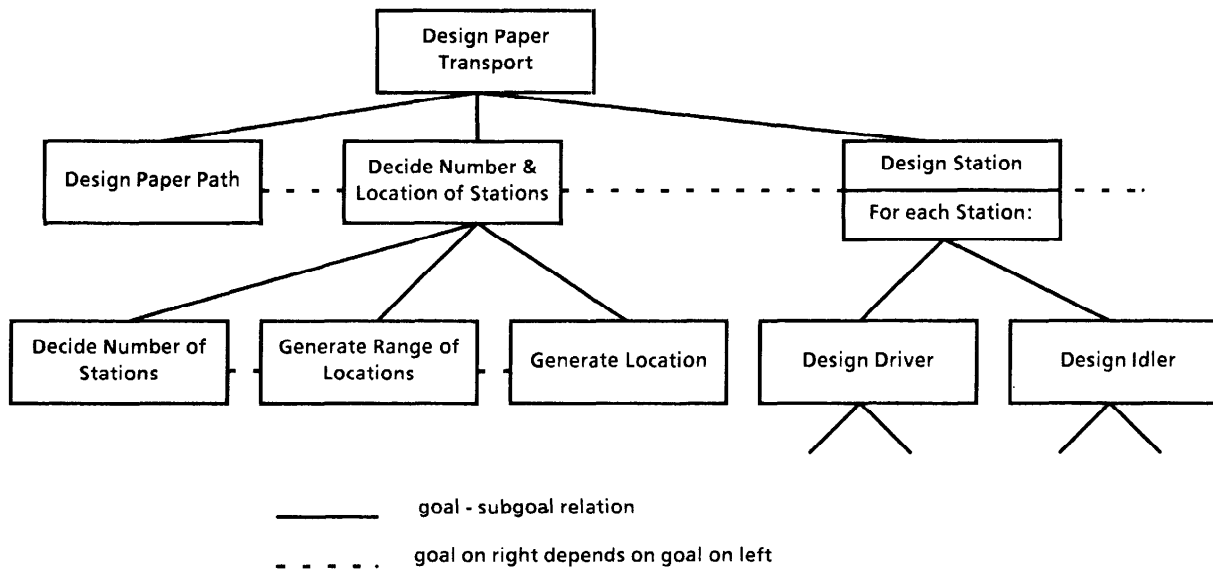


Fig 2: Part of goal tree for Paper Handling Systems

Idler width generator

parameter : Idler width
 min value : 10mm
 max value : 100mm
 step : 1mm
 initial value : if driver width known
 then $2 * \text{driver width}$
 else 40mm

Fig 3: Generator for Idler Width

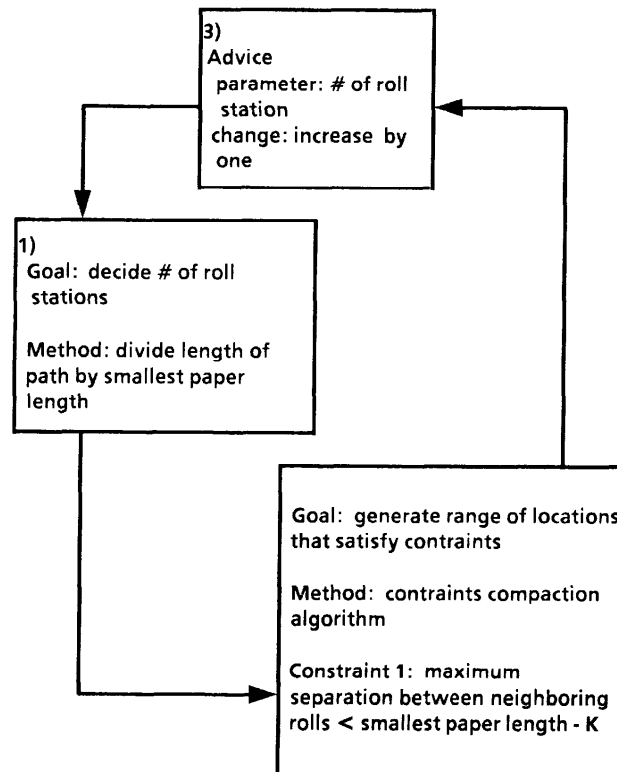


Fig 4: Advice Example