85-31

Dominic: A Domain-Independent Program for Mechanical Engineering Design

Adele E. Howe *
Dept. of Computer and Information Science, University of Massachusetts

John R. Dixon
Dept. of Mechanical Engineering, University of Massachusetts

Paul R. Cohen
Dept. of Computer and Information Science, University of Massachusetts

Melvin K. Simmons
Knowledge Based Systems Branch, Corporate Research and Development
General Electric Company

March 9, 1986

Abstract

We describe an Artificial Intelligence (AI) program for mechanical engineering design. The program, called DOMINIC, characterizes design as best-first search through a space of possible designs. DOMINIC is a general architecture for a class of mechanical engineering design problems; within its *redesign* framework, in which a design is iteratively modified and improved, one can design a variety of mechanical devices. DOMINIC's performance on two design problems is evaluated, and a battery of experiments with DOMINIC is discussed.

Introduction

Design is pervasive. It has been referred to as the core of all professional activity, and called the "proper study of mankind" [1]. It is an underconstrained, ill-structured problem – a search for alternative solutions in a large space of possibilities [2]. Mechanical engineering design is the activity of creating the concepts and the detailed instructions that specify the manufacture of material parts, products, and systems [3]. It is often iterative – a series of actions and evaluations that cause a progression from vague specifications to a physical implementation [4, 5].

Artificial intelligence systems implement design as a search through a design abstraction space by successively applying operators to the problem specification to satisfy the goals and constraints of the design. Described so generally, design may be viewed as state-space search [6], where the states represent the artifacts under design or their components (e.g., [1]). But

*This work has been partially supported by a grant from the General Electric Company to the University of Massachusetts, Amherst.

many of the issues that limit search as a technique for problem-solving and planning [7] also limit this view of design (e.g., mutually constraining goals, control of explosive search spaces). Thus, the AI approach to design is following the trend toward a variety of knowledge-based methods in several expert domains, such as VLSI layout and timing, electrical circuit analysis, and automatic programming [8]. Research into knowledge-based expert systems for mechanical engineering design is reported in [9, 10, 11].

The domain of mechanical design is wider than that of many other expert tasks: a designer may be asked to design an enormous range of artifacts in different domains. We want fairly general AI systems that can design in many domains, yet we recognize the need for domain-specific knowledge to control the search of the space of possible designs. We have built a system for mechanical design in part to explore this *power/generality* tradeoff. Dominic treats design as *best-first search*, allowing us to experiment with the incorporation of domain-specific knowledge into its evaluation function. Other general architectures for mechanical engineering design are described in [12] and [11]. Their work differs from ours in a number of respects. Perhaps most important, it incorporates much more domain-specific expert knowledge, whereas we rely primarily on weak methods. We expect their systems to handle fairly complex design problems, while we handle a wider range of simpler problems. No single point on the power/generality spectrum is the "right" one; our approach is to work toward powerful design from the generality end of the spectrum. We believe that Mittal et al. and Brown and Chandrasekaran are working nearer the other end of the spectrum. A second difference is that we do not address the issue of decomposing design problems into subproblems. We focus instead on the problem of iteratively redesigning a single object or simple physical system.

In the next sections, we describe Dominic in overview, then detail its flow of control and knowledge representation, then give an example of Dominic at work, and analyze its performance in two domains.

Dominic

1. Overview

Dominic solves design problems that can be represented by a fixed, finite set of *design variables*. These variables represent the physical parameters of a design (e.g., the dimensions and material of a part are design variables). Every point in the space of designs is a combination of the values of each design variable. Dominic moves from one design to another by changing one or more of these values. The space of designs represented by n design variables is n-dimensional. Adding another dimension – the "goodness" of each design – defines a hill. The space can then be searched by hillclimbing. This requires a method to calculate the "goodness" of a design given the values of its design variables, which in turn requires some way to relate one's *design goals* (such as product life, cost, or heat transfer) to the design variables, and to combine the degrees to which individual design goals are achieved into an overall degree of goodness for the design. Dominic works as just described. The space of possible designs is circumscribed by a set of design variables. A *dependency table* records how changes in each design variable affect the level of satisfaction of each goal. Design changes are based on these dependency table

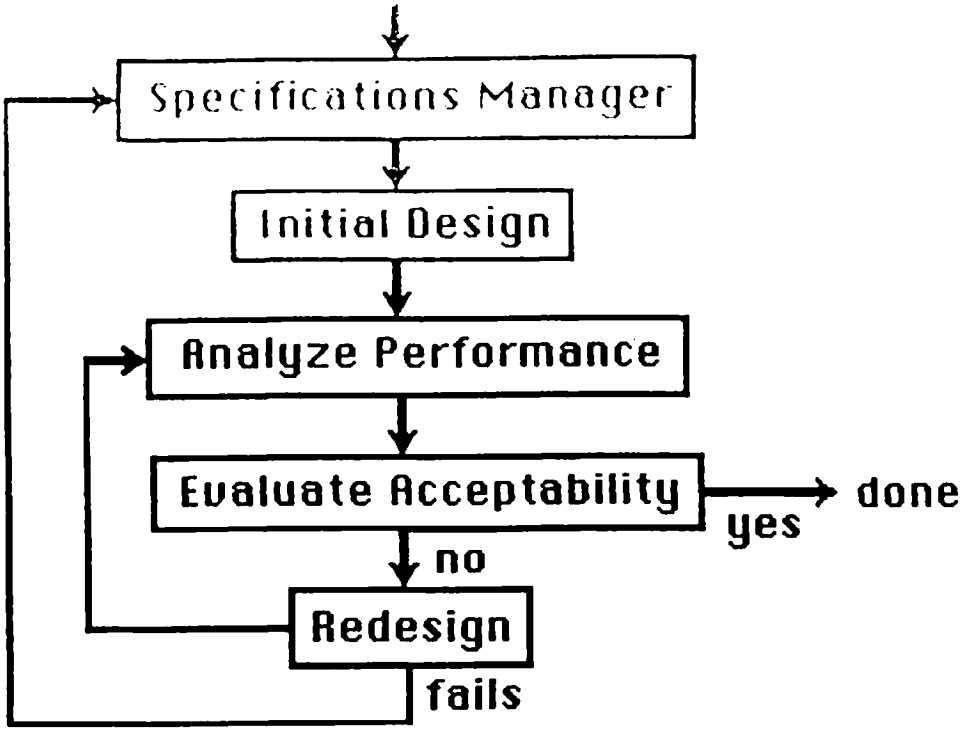

Figure 1: Evaluate-and-Redesign Model

values. The level of satisfaction of each goal is assessed, and these levels of satisfaction are heuristically combined to produce, for each point in the space of designs, an overall evaluation. Dominic starts at one point in this space called the *initial design*, and iteratively changes the values of design variables to move to other points that have successively higher evaluations. As it moves through the space of designs, Dominic gains information about the dependencies between design variables and design goals and updates its dependency table with each iteration. It quits when it achieves a satisfactory design.

2. Flow of Control

The process we just described is quite general, therefore Dominic can, in principle, solve a great many design problems. The process iteratively evaluates and changes a design (redesigning it) until the design meets the constraints and the goals of the task. As other researchers have recognized [13, 14], design is frequently *re*design; that is, rather than create a design from first principles, one modifies an existing design to conform to new requirements or eliminate problems. We call this the Evaluate-and-Redesign architecture [3].

The Evaluate-and-Redesign model has five steps: problem specification, initial design, evaluation, acceptability, and redesign (Fig.1).

Design problems are specified by a user. Then Dominic either computes an initial design from routines supplied by a domain expert, or accepts an initial design from the user. The expected performance of the design is then assessed as a combination of the levels of performance on each of the problem's goals. These, in turn, are compared to *desired* levels of performance (determining levels of *satisfaction*). If the design is acceptable, then the process is complete. Otherwise the design is *redesigned* until it is acceptable.

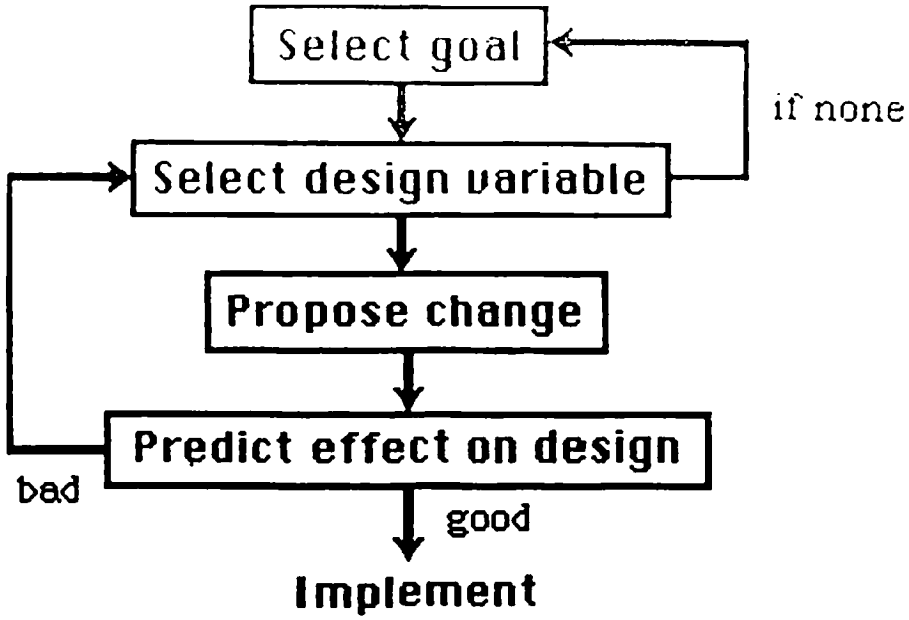

Figure 2: Redesign Process

The redesign module determines which design variables to change and how much to change them. Redesign has five steps. (Fig. 2).

The program first selects a goal for attention. Goals are ordered according to priorities specified by the user, and need of attention, determined during the evaluation phase (Fig. 1). The top-ranked goal is selected. Next, a design variable is selected to produce a change in the goal. Dominic uses heuristic knowledge, provided by the domain expert, about which design variable to change first to produce change in a particular goal. This knowledge tells Dominic which design variables produce the most change in the target goal with the least detrimental effect on the other goals.

Changes in the values of design variables are computed from knowledge about the *dependencies* between design variables and design goals. Dependencies note how the level of satisfaction of a goal covaries with values of a design variable. The redesign module proposes a new value for the selected design variable, confirms that the change does not violate user-specified *constraints* on the value of the variable, and predicts the effects of the change on the overall design using, again, dependency information. If the proposed change violates constraints or lowers the value of the design overall, then a revised change is proposed. If all proposed changes to the design variable are poor, the variable is withdrawn and another design variable is tried, or, if necessary, another goal is selected.

3. Knowledge Representation

Domain Representation This section describes the knowledge and knowledge representations that Dominic uses during the design process. A domain expert interacts with Dominic to provide basic knowledge of a new domain. Dominic acquires other knowledge (about depen-

dencies) itself as it solves problems. This knowledge, along with problem-specific information given by the user, forms the knowledge base needed by Dominic to solve problems in a new domain.

A domain is described by *problem specifications, design variables, and design goals.* Problem specifications are fixed requirements or constraints (e.g., an object must be less than two feet wide). Design variables specify physical dimensions or objects in the final design. They are the parameters that define a design and are manipulated during the redesign process. The actual width of the object or its distance from another object are design variables. Design goals are parameters that measure the quality of the design. Product cost and expected lifetime, heat transfer and speed of rotation are design goals. Shortly, we will describe how Dominic assesses how well each of these goals is satisfied by a design, and combines these individual degrees of satisfaction into an overall evaluation.

Redesign Knowledge The redesign module employs heuristic and numeric knowledge about design variables and goals to propose design changes. *Satisfaction indexes* for each goal assess how well the goal is satisfied by a design. Dominic uses the levels of satisfaction of goals, as well as their *importance* (specified by a domain expert or user) to decide which of several goals to work on. The satisfaction index for the *cost* goal, for example, is a monotonically decreasing function of cost: one cannot imagine increased cost increasing one's satisfaction! Satisfaction indexes for other goals are more complex: as product life increases, so does one's satisfaction; then for some range of product life, satisfaction is at a maximum; but if the product has too long a life, satisfaction begins to decrease, on the assumption that the excessively long life could be traded for improvement in some other goal (e.g., cost).

The degree of satisfaction in each goal is calculated from its satisfaction index, then these numbers are combined heuristically to give an overall degree of satisfaction for the whole design. This combination is sensitive to differences in the importance of goals. The combining function captures knowledge of the form, "If all the goals are maximally satisfied, then this is the best design; if all but the unimportant goals are maximally satisfied, and they are nearly satisfied, then this is the next best situation ... ," and so on.

Heuristic knowledge guides which design variable to change. A *dependency order list* for each goal specifies which design variables to change first, to improve the level of satisfaction for that goal. Dependency order lists are supplied by the domain expert; they reduce the number of design variable changes that would otherwise be considered.

The *dependency table* is an array of numeric relations (called *dependencies*) between design variables and goals. Dependencies relate how the design evaluation, via goals, is affected by changes in design variables. The redesign module uses dependencies to determine how much to change a design variable to produce a desired change in a goal, and to predict the effect of the proposed design change without having to run costly analysis routines. Dominic has access to analysis routines, but it runs them only after a change has been proposed, its effect predicted, and the change is accepted. This way, Dominic does not invoke expensive analyses for each change it *considers*, only for those that are predicted by the dependency table to be beneficial. Clearly, the quality of the information in the dependency table affects Dominic's performance. At the start of a design problem, dependencies are roughly estimated by a domain expert. But as Dominic solves problems, they are updated empirically from design changes. This is possible

because Dominic runs analyses after a proposed change is implemented, so it gains information about the true effect of a particular change in a design variable on each goal.

Dependencies are represented as logarithmic derivatives of the form:

$$\frac{Ynew}{Yold} = \left|\frac{Xnew}{Xold}\right|^{D}$$

in which *Ynew* is the desired goal value, *Yold* is the current goal value, *Xold* is the current design variable value, *Xnew* is the design variable value needed to produce the desired goal value and *D* is the dependency relating the goal and the design variable. The required design variable change, *Xnew*, is computed from this equation given the desired and current goal values, the current design variable value, and the goal/design variable relationship.

In summary, our design for Dominic was dictated by several goals. Design problems are given a simple representation in terms of design variables, goals, and problem specifications; and the space of designs is searched by a simple hill-climbing algorithm, a weak method that is sensitive to the accuracy of its evaluation function. By adopting this weak model of design, we make Dominic a very general, but potentially inefficient, program. Efficiency is improved by adding heuristic knowledge. Technically, Dominic then becomes a best-first search algorithm, because control decisions depend on more than just potential improvements in the design. Dominic's domain-specific, heuristic knowledge includes dependencies, dependency order lists, the relative importance of goals, satisfaction indexes, and the function that combines satisfaction levels for individual goals into an overall level of satisfaction. We want it to be easy to change design domains by "pulling out" one set of knowledge and "plugging in" another. Thus, we ask for as little domain-specific knowledge as is necessary to limit search. An example of this trade-off is dependency information: Dominic relies heavily on its dependency table, so inaccurate dependencies can mislead it. But rather than insist on accurate dependency information for all pairings of goals and design variables (a lot of information), we allow the user to give fairly rough dependency information (or none at all) and let Dominic revise it as it gains information from analysis routines. As we said in the introduction, we emphasize generality over power in Dominic. If an expert can give precise dependency information for a domain, then Dominic will use it and perform efficiently; if not, the program may have to solve a few problems in a domain to "tune" its dependency table before turning in expert performance.

4. Performance

Domains Our primary goal in developing Dominic was to explore design issues for a general architecture mechanical engineering design. Dominic has been tested successfully in two domains: standard v-belt drive systems and aluminum extruded heat sinks.

A v-belt is a simple belt and pulley system for transferring power from one pulley to another (Fig. 3). It has three major parts: two pulleys (called drive and load) and the belts. The problem specifications are minimum center distance, maximum center distance (distance between the two pulleys), drive speed, load speed, load power, and minimum system life. A v-belt design is described by design variables for the drive diameter, load diameter, belt type, belt length and number of belts. The goals are cost, life, load speed, force on the shaft, and belt velocity.

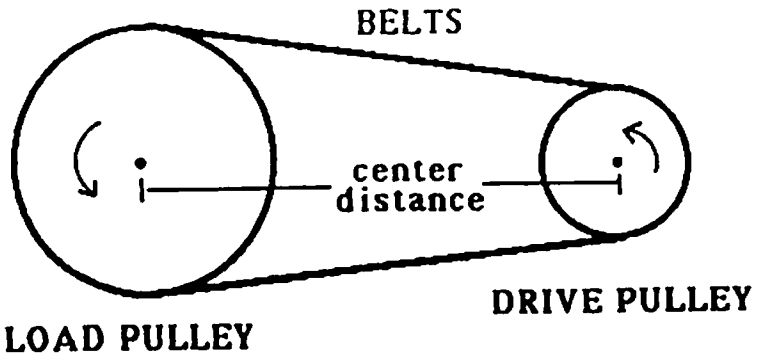

Figure 3: V-belt Drive System

Dependencies relate goals to design variables; for example, life is approximately proportional to the number of belts raised to the fourth power, and load speed is independent of the number of belts.

The second domain is the design of aluminum heat fin arrays (Fig. 4). Heat fins dissipate heat from electrical or electronic devices. The problem specifications are the length, width, thickness of the base, target temperatures, and maximum allowable height. The design variables are the length, thickness, and spacing of the fins. The goals are heat transfer and cost.

Summary of Results Overall, Dominic's performance on problems in both domains is good. In each domain, Dominic's design compared favorably to designs produced by a domain-specific expert system (VEXPERT, [9] for v-belts and XENIF, [10] for heat sinks) and a human expert. For example, on a typical v-belt problem, Dominic produced a slightly different v-belt design from either the human expert or the domain specific expert system. The design met the life requirement of 8000 hours exactly at a lower cost. On a typical heat sink problem, Dominic's design had a slightly better heat flow than the other two, at a slightly higher cost. Typical results are presented in Fig. 5.

Dominic is relatively easy to use. We found that it required little effort to specify a new domain. Dominic's initial domain was v-belt. Adding the heat sink domain took about three hours with the domain expert and about 40 hours for the system developer, acting as knowledge engineer. However, most of that time was spent making FORTRAN heat transfer analysis routines communicate with LISP, Dominic's implementation language.

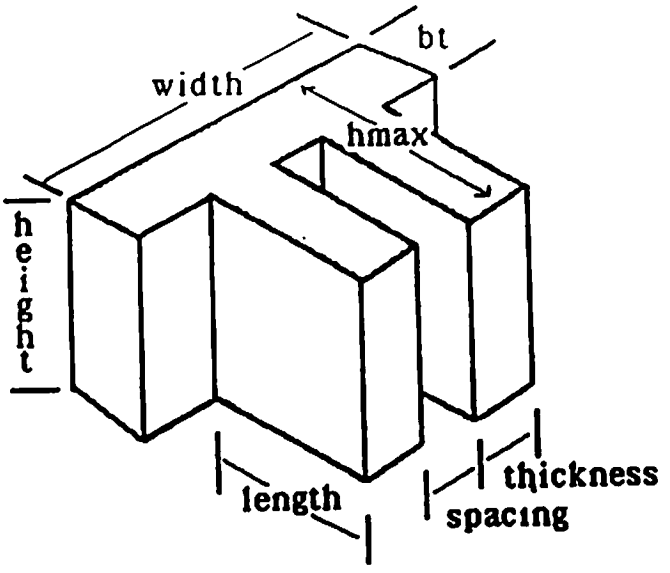

Figure 4: Heat Fin Array

Comparative V-Belt Designs

	Expert	Vexpert	Dominic
D1	30	22.4	17.6
D2	18	13.2	10.5
L	140	100	100
M	12	15	16.5
COST	1834	1630	1578
LIFE	4586	8436	8000

Comparative Heat Sink Designs

	Expert	Xenif	Dominic
Thickness	0.001	0.002	0.001
Height	0.0352	0.0491	0.0376
Spacing	0.007	0.0123	0.0094
Cost	1.10	0.95	1.11
Heat Flow	50.0	50.0	51.30

Figure 5: Typical Performance Results

5. Analysis

The final phase of development of a knowledge-based system is usually a performance test. We felt that because Dominic dealt with a complex, poorly understood domain, design, a simple performance test would be uninformative. Dominic's development was guided by our intuitions about the design process. Testing the completed system without testing alternative configurations did not prove that our development decisions were most appropriate. We thus undertook a series of experiments to explore alternative configurations for Dominic.

Dominic was varied along 5 dimensions: How are initial designs selected? How is the dependency table updated? How are dependency order lists used? What is the structure of satisfaction indexes and how are they used? and How are design variables selected for change? These questions were selected as the most crucial to the redesign architecture. Each denotes a dimension along which the program was altered as many ways as practical.

Initial design selection was varied to test the consequences of less facilitative initial designs. Initial designs were chosen in three ways: by routines that produce initial designs; by us, to be relatively close to the *target design* – the best design for the problem; or by us to be relatively far from the target design.

In another experiment, the dependency table was not allowed to change during program execution, to test the effect of the dependency table on performance and evaluate the importance of updating the dependency knowledge with information gained by running the analysis routines.

Dependency order lists specify the order in which to try design variables if you want to change a design goal. We tested the effects of ignoring this heuristic knowledge. Instead, Dominic selected the design variable that had the largest effect on the goal it was trying to change, according to the dependency table. (Note that this is a pure hill-climbing strategy.)

The satisfaction indexes for goals can be divided into regions – qualitative evaluations of how well the goal is satisfied. In Dominic these regions are *good* or *unacceptable* with levels in between. We varied the number of qualitative levels, and also the quantitative distance between them, to evaluate the effects on the time required to produce a design and the final quality of the design.

Finally, seven different strategies for selecting design variable changes were explored. Strategies control the speed of goal change and the resistance to possibly detrimental changes. For example, one strategy supports changing a goal as much as is needed to be evaluated as *good* (quick changes) provided the proposed change will not produce an overall decrease in design quality (high resistance to detrimental effects); another strategy advocates improving a goal by a single satisfaction level (slower changes) regardless of the overall effect on the design (no resistance to detrimental effects). We explored whether strategies were differentially appropriate for the v-belt and heat sink domains, and whether they could prevent a design from converging.

We performed 17 experiments varying a single dimension on each of three v-belt problems and two heat sink problems, and 20 combination experiments on one problem from each domain, for a total of 125 experiments. The results can be summarized with some observations.

Dominic rarely didn't work at all. In nearly all the experiments, Dominic produced at least an acceptable design. It seems that the architecture is robust enough, or the search space is constrained enough, so that varying any single dimension does not cripple the system.

The unmodified program performed best. Dominic's development was guided by our intuitions about the design process, and it appears, from these results, that our intuitions were justified.

The program performed best with initial designs that were not extremely far from the target design. Because initial designs are usually computed from an initial design routine, we were concerned that Dominic would be unable to recover from a poor design routine. In both domains, Dominic produced good designs quickly, given initial designs that were either very close to the best design or at a moderate distance. However, its performance was unacceptable when the initial design was distant from the best design.

As Dominic solves a design problem, the dependency table is updated. Our experiments showed that when the dependency table was not updated, Dominic produced poor designs for v-belt problems and inefficient solutions for heat sink problems. By not updating the dependency table, its values become inaccurate. Roughly, the dependencies that hold between goals and design variables in one part of the design space do not hold in other parts of the space, so Dominic needs to update the dependencies as it moves through the space. Because Dominic relies on dependencies to select among proposed design changes, disallowing updates causes the program to produce unacceptable designs, or to oscillate around reasonable dependency values until it stumbles onto them. The heat sink domain is less affected by a static dependency table because it has fewer goals and design variables and is extremely constrained by manufacturing considerations.

We found that if one ignores the expert's intuitions about which design variable to select (i.e., ignores the dependency order list), difficulties result in domains with many variables, like v-belt. The dependency order list provides expert heuristic guidance to limit the search of design variables and best select variables which will produce the most effect with the least detrimental repercussions. By ignoring the list, the program selects design variables without accounting for their full effect on the design.

Changing the number of levels on the satisfaction indexes influences the size of design changes. Enlarging the distance between levels forces the program to produce a more significant improvement in a particular goal to, in turn, produce any improvement in the overall evaluation. Large changes in design variables produce strong, frequently undesirable, repercussions to the overall design. Conservative changes potentially cause the design to improve more slowly, but less frequently produce undesirable effects from which the program must recover. So, conservative changes in design variables are usually better than drastic ones.

The strategies for selecting among design variable changes affected Dominic's performance differently in the two domains. We believe that individual domain differences, such as this, may necessitate tailoring redesign strategies to particular domains. For this reason, we explored the possibility of Dominic "tuning" its strategies automatically to a new domain.

Note that the five dimensions above constitute a multidimensional space of the kind that Dominic searches. This leads to the intriguing possibility of having Dominic configure itself, by best-first search through a space of configurations. One of us [15] tried this experiment with some success. Dominic begins in a poor configuration and produces poor designs, and then redesigns its configuration, using the quality of its designs (speed of convergence, tendency to oscillate, etc.) to evaluate the "goodness" of a configuration. It is actually able to improve its configuration to the point that it produces adequate designs. This result has important

consequences for our approach, which is to design as well as possible given relatively little domain-specific information. It appears that we may be able to keep Dominic near the generality end of the power/generality spectrum if it is able to adapt itself to new domains.

Discussion Dominic currently designs at an expert level in two domains. It has been tested in one other mechanical engineering domain. It is a general program: we believe it will design any system that can be represented by goals and design variables. When Dominic itself was represented this way, it was able to improve its own configuration. But this generality comes at a price. Dominic probably cannot design efficiently in much larger search spaces (e.g., ten goals and design variables), unless we incorporate more heuristic knowledge. Nothing prevents this, but our aim has been to start with a general architecture, since design programs must be general. Dominic makes the most of the expert guidance it receives, but it is robust if that knowledge is missing or inaccurate.

The current version of Dominic has limitations. All design variables must have continuous numeric values. Final designs in the v-belt problem included fractional numbers of belts. While the evaluation routines can handle such a concept, it is difficult to actually construct a v-belt system with 2.3 belts. Additionally, the system cannot select symbolic changes. For example, in the v-belt domain, different types of belts are available. Consequently, Dominic was unable to change the belt type and simply assumed that the one chosen by the initial routine would be adequate.

Dominic has difficulty with mutually constraining variables. In the v-belt problem, the size of the load diameter and the drive diameter are related to each other by a fixed ratio which affects loadspeed. Since Dominic changes only one variable at a time, changes must be sufficiently small to not violate the ratio.

Both of these problems are being addressed in the next version of Dominic. They will require more heuristic knowledge of Dominic's users, and thus reduce its flexibility, but will increase its power. In addition, we are designing a module that uses heuristics to alter the redesign strategy to best suit the problem. For example, if a mutually constraining situation is identified, the redesign strategy may be changed to relax the constraint and allow temporary violations or move through the design space only in directions that preserve the constraint.

6. Conclusion

Dominic can be characterized as a best-first search algorithm. It solves mechanical engineering design problems formalized in terms of problem specifications, goals, and design variables. It has successfully demonstrated its capabilities in two rather different domains. Additionally, we have examined alternate configurations of the program and validated our program development decisions. Further work will make Dominic more powerful but, if possible, no less general.

References

1. Simon, H.A., *The Sciences of the Artificial*, MIT Press, Cambridge, MA. 1967.

2. Krick, E.V., *An Introduction to Engineering and Engineering Design*, John Wiley & Sons, NY, 1967.

3. Dixon, J.R., Simmons, M.K., Cohen, P.R., "An Architecture for Application of Artificial Intelligence to Design", *Proceedings of ACM/IEEE 21st Design Automation Conference*, Albuquerque, NM., 1984.

4. Gibson, J.E., *Introduction to Engineering Design*, Holt, Rinehart, & Winston, NY, 1968.

5. Asimow, M., *Introduction to Design*, Prentice–Hall, Inc., Englewood Cliffs, NJ, 1962.

6. Barr, A., Feigenbaum, E., *Handbook of Artificial Intelligence*, William Kaufmann, Inc., Los Altos, CA, 1981.

7. Cohen, P., Feigenbaum, E.,*Handbook of Artificial Intelligence*, William Kaufmann, Inc., Los Altos, CA, 1981.

8. Mostow, J. "Towards Better Models of the Design Process", *AI Magazine*, 6:1, pp.44–56, 1985.

9. Dixon, J.R., Simmons, M.K., "Expert Systems for Engineering Design: Standard V-Belt Design as an Example of the Design-Evaluate-Redesign Architecture", *Proceedings of the 1984 ASME Computers in Engineering Conference*, Las Vegas, NV, August 12-15, 1984.

10. Kulkarni, V.M., Dixon, J.R., Simmons, M.K., Sunderland, J.E., "Expert Systems For Design: The Design of Heat Fins as an Example of Conflicting Subgoals and the Use of Dependencies", *Proceedings of the ASME Computers in Engineering Conference*, Boston, MA, Aug.4-8, 1985.

11. Brown, D.C., Chandrasekaran, B., "An Approach to Expert Systems for Mechanical Design", *Proceedings of Trends and Applications*, National Bureau of Standards, Gaithersburg, MD, May 25-26, 1983.

12. Mittal, S., Dym, C.L., Morjoria, M., "PRIDE - An Expert System for the Design of Paper Handling Systems", In *Applications of Knowledge Based Systems to Engineering Analysis and Design*, Winter Annual Meeting of the American Society of Mechanical Engineers, Miami, FA, Nov 17-22, 1985.

13. Sussman, G.J., "Electrical Design: A Problem for Artificial Intelligence Research", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

14. Mitchell, T., et.al. "An Intelligent Aid for Circuit Redesign", *Proceedings of the National Conference on Artificial Intelligence*, Washington, D.C., 1983.

15. Howe, A. "Learning to Design Mechanical Engineering Problems", EKSL Working Paper 86-01, University of Massachusetts, 1986.