Massachusetts Institute Of Technology
Artificial Intelligence Laboratory

A.I.Memo No. 1084 January, 1989

The Performance of a Mechanical Design "Compiler"

Allen C. Ward
Warren P. Seering

ABSTRACT

A mechanical design "compiler" has been developed which, given an appropriate schematic, specifications, and utility function for a mechanical design, returns catalog numbers for an optimal implementation. The compiler has been successfully tested on a variety of mechanical and hydraulic power transmission designs, and a few temperature sensing designs. Times required have been at worst proportional to the logarithm of the number of possible combinations of catalog numbers.

This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Funding for the work was provided in part by the Industrial Technology Institute of Ann Arbor, Michigan, and in part by the Office of Naval Research under University Research Initiative contract N00014-86-K-0685.

1 Introduction

Among our research goals is the development of "mechanical design compilers"; that is, programs which take as input a schematic (or other high-level description) of a mechanical design, plus specifications and a cost function, and return a description of the optimal implementation of the design, sufficiently detailed to support manufacture. Such programs should decrease design time and cost, increase design quality, and allow designers to explore more alternatives in greater depth.

We have chosen to work initially in the domain of mechanical and hydraulic power transmission systems built from cataloged components. For this domain we have substantially accomplished our goal; this paper presents the evidence for that success, and discusses its limitations.

After mentioning some related work, we provide a very brief over-view of the compiler, intended only to allow the reader to understand our performance evaluation. For a more detailed introduction to the compiler, and in particular the theory on which it rests, see [1]; for full details see [2]. We then examine the capabilities of the compiler in three different respects: 1) the range of design problems it has been tested on; 2) its reliability; and 3) its efficiency or time complexity.

2 Related Work

We have found no other programs identified as "mechanical design compilers" by their creators. [3] and [4] discuss programs which offer the designer a schematic language, but which perform analysis only. We argue in [1] that the traditional "constraint propagation" methods they use are inadequate to represent essential mechanical design information.

[5] and [6] discuss programs able to find the optimal selection of a single component, given constraint and cost equations. These use "hill-climbing" optimization routines, with heuristics to modify the hill-climbing process when they get stuck. The "hill-climbers" are called by supervisory programs which can represent combinations of components. The supervisory programs supply the hill-climbers with specifications, based on "expert knowledge", and in [5] using iteration to improve the initial guesses.

The approach these programs use has some disadvantages. They appear to require substantial effort to set-up each configuration of components; a "domain" in [5] is equivalent to a single schematic for our system. They use nominal values; manufacturing tolerances, and variations in operating conditions are not explicitly represented. The "hill-climbing" search process can become stuck on local optima. Because the set of artifacts implicitly represented by the design is continually changing, all calculations must be repeated at each step, and correcting one deficiency can introduce others. For these reasons we have chosen a radically different approach.

3 Overview of the Compiler

Figure 1 illustrates our approach. Our data base is built up (block 1) from "basic sets" of artifacts. Each basic set is represented by a single catalog number. The set consists of the individual artifacts one might receive by ordering that catalog number. For example, on ordering Dayton motor number 2N103, we will receive any one of an effectively infinite variety of motors, each slightly different because of manufacturing tolerances, each with its own serial number; these motors make up the basic set denoted by catalog number 2N103.

The basic sets are modeled by an engineer, using equations and specifications in a special "labeled interval" specification language. For example, the speed regulating characteristics of Dayton motors 2N103 might be represented by $\langle \mathbf{A} \overset{only}{[\ ]}\ RPM\ 1740\ 1800 \rangle$. This specification tells us that we are *assured* (**A**) that for any motor we might get by ordering number 2N103, the the RPM will take on *only* ($\overset{only}{[\ ]}$) values between 1740 and 1800, under normal loading.

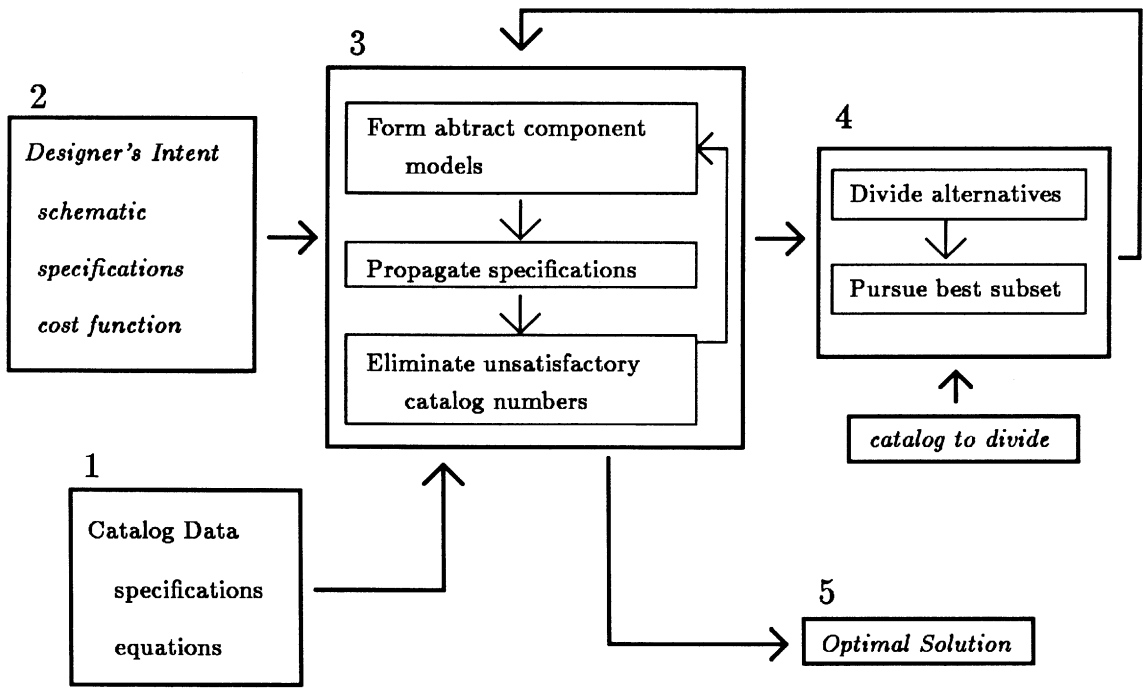


Figure 1: The Compiler Block Diagram

The engineer groups the catalog numbers into a hierarchical structure, and the compiler **abstracts** (block 3) the information about the basic motor sets to form descriptions for higher levels in the hierarchy. For example, the next level up might be all the 1800 rpm three-phase motors represented; these have varying degrees of speed regulation, so the set as a whole might only guarantee speed regulation between, say, 1700 and 1800 rpm: $\langle \mathbf{A} \overset{only}{[\ ]} \mathit{RPM}\ 1700\ 1800 \rangle$. Finally, a schematic symbol (Figure 2) represents the whole hierarchy of catalog numbers, and therefore the union of their basic sets. The motor symbol might initially represent all of the electric motors listed in the Dayton catalog[1].

The compiler's user, a mechanical designer, **composes** new designs by pointing at schematic symbols (block 2). The system automatically makes appropriate connections, asking for help if needed to resolve ambiguities; for example, in adding the first cylinder to the schematic of Figure 3, the compiler would have to ask which valve to attach it to. Having defined such a design schematic, the user may assign it a symbol of its own, for recall or use in more complex designs.

The compiler automatically **eliminates** catalog numbers which are incompatible with *any* implementation of the connected components (block 3). For example, on connecting the motor schematic to one representing a 220-volt power supply, the system automatically eliminates any 110 volt motors.

After building the schematic, the user provides specifications. These specifications describe sets of operating conditions; $\langle \mathbf{R} \overset{every}{\leftrightarrow} \mathit{speed}\ 0\ .2 \rangle$ applied to a cylinder means that the speed of the cylinder shaft is "Required" (**R**) to take on every value ($\overset{every}{\leftrightarrow}$) in the interval from 0 to .2 feet per second.

[1]The currently implemented catalogs only include a subset of the Dayton catalog.

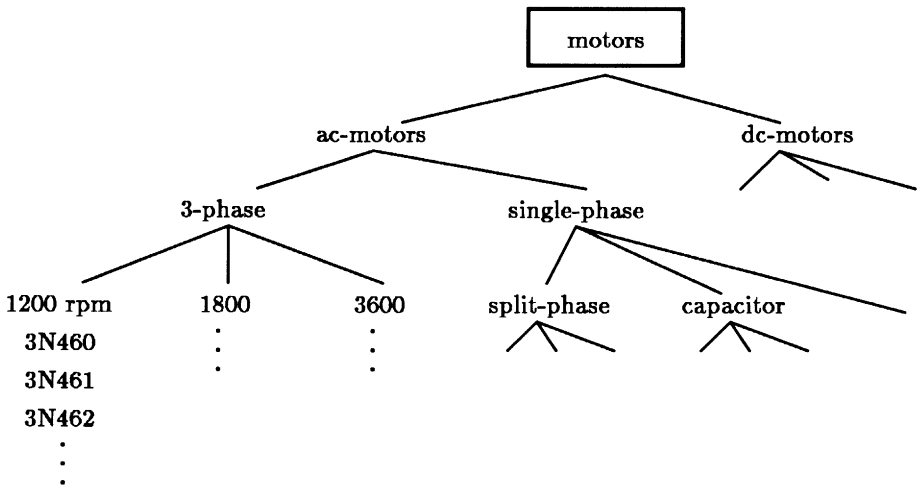


Figure 2: A hierarchy of motors

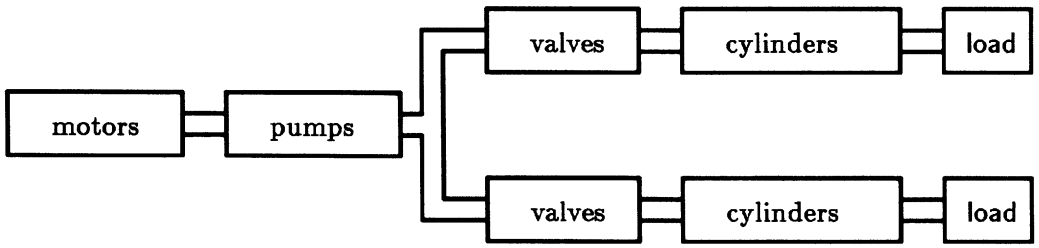


Figure 3: A Hydraulic Power Train

In this example, the maximum output pressure available from any of the pumps, together with the highest of the range of forces required, sets a minimum diameter requirement on the cylinders. These, together with the speeds required, establish flow requirements. This use of equations and specifications to form new specifications is **propagation** (block 3); it can be regarded as a generalization of "the constraint propagation of intervals" [7]. More specifically, the constraint propagation of intervals corresponds to one of 21 propagation operations employed in the compiler.

The propagated specifications for flow, horsepower, torque, and so on cause further eliminations, leaving subsets of the original catalog numbers. Descriptions for these subsets are then abstracted to produce new specifications, which trigger further propagation and elimination.

When the cycle of abstraction, propagation and elimination ceases, a variety of alternative combinations of catalog numbers often remains. The user then provides a cost function, for example the weighted sum of the price and weight of the components. He also directs the compiler to split one of the catalogs in half, for example to look at 3600 and 1800 rpm motors separately (block 4)[2]. The compiler then generates two daughter designs, one for each motor set; the abstraction

[2]Having the user guide the search in this way improves efficiency; catalogs could be selected for splitting randomly, or by a heuristic.

operators formulate new specifications describing the new, smaller motor sets. These specifications trigger another cycle of eliminations.

Repeating this splitting process generates a binary best-first search tree. The compiler always splits the leaf of the tree offering the lowest possible cost. The search continues until a single catalog number remained for each component.

The output of this compiler thus consists primarily of catalog numbers. Given these numbers and the schematic, most mechanics could probably buy the components and construct the system without further input from an engineer. A future, more complete compiler would provide a drawing of the base-plate. The most complete compiler possible would instruct an automated manufacturing system to build the design.

4 Some Examples

In this section I discuss in general terms my experience with the compiler. Figure 4 shows some component types, with the primary variables used to model them.

The component models now used include specifications for most of the non-geometric criteria that the vendors discuss in the "engineering" sections of their catalogs. Formulating a representation for a component type requires the engineer to extract a precise model, in our formal specification language, from the usually vague and often inconsistent catalog data. He must compromise between simplicity and completeness. For example, we have chosen to represent the efficiency of mechanical transmissions as a range of possible values, from 90 to 98 percent. We could instead have entered an equation relating efficiency to speed; manufacturing variations would then be represented by interval specifications on the coefficients of that equation.

Once the form of the precise model has been determined, a simple program can be instructed to translate the manufacturer's catalog into the labeled interval specification language. Entering further catalog numbers for components of this type is then a typing exercise. It generally takes about one day to decide the form of the specifications and equations for a new kind of component, generate the transformation procedure that converts the catalog to the desired form, and test the results.

The system has been tested on a few temperature measurement system design problems and more than a dozen different arrangements of power transmission components. Figure 5 shows some of these, with machines in which the power trains might be used.

Let us now consider in more detail the two-cylinder hydraulic system example of Figure 3. The catalogs for the components shown include the following numbers of alternatives: 7 types of electrical supply (omitted from the schematic), 36 motors, 13 pumps, 3 valves, and 12 cylinders. There are thus 4,245,696 possible combinations; of these, because our catalogs are still sparse, only 505,440 remain after the eliminations caused by connecting the components.

After composing the schematic, the user then enters load specifications, for example:

Load-1: $\langle \mathbf{R} \overset{every}{\leftrightarrow} speed\ 0\ .2\rangle$, $\langle \mathbf{R} \overset{every}{\leftrightarrow} force\ 0\ 1000\rangle$
Load-2: $\langle \mathbf{R} \overset{every}{\leftrightarrow} speed\ 0\ .15\rangle$, $\langle \mathbf{R} \overset{every}{\leftrightarrow} force\ 0\ 3000\rangle$.

For the first load, this means that the system must provide every speed from 0 to .2 feet per second, with forces from 0 to 1000 pounds.

The compiler uses these specifications, and those built into the catalogs, to eliminate unsatisfactory alternatives and to generate further specifications. For example, the linear horsepower equation is built into the "load" component, $hp - \frac{(force)(speed)}{550} = 0$. The compiler incorporates an inference rule which can be written

$$\langle \mathbf{R} \overset{every}{\leftrightarrow} x\ x_l\ x_h\rangle \& \langle \mathbf{R} \overset{every}{\leftrightarrow} y\ y_l\ y_h\rangle \& G(x,y,z) = 0 \longrightarrow \langle \mathbf{R} \overset{every}{\leftrightarrow} z\ \textsc{Range}(G, \langle x\ x_l\ x_h\rangle, \langle y\ y_l\ y_h\rangle)\rangle.$$

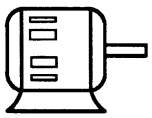
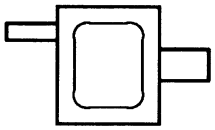
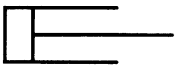
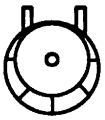
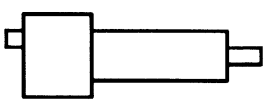
parts	variables represented	
 electric motors	power type	rpm
	voltage	efficiency
	current	torque
	power	
 mechanical transmissions	torque-out	power-out
	torque-in	power-in
	rpm-out	efficiency
	rpm-in	
ball screws	rpm	force
	lead	critical rpm
	speed	buckling load
	torque	length
 hydraulic cylinders	diameter	flow
	pressure	speed
	force	
 hydraulic pumps	rpm	displacement
	flow	total efficiency
	power	volumetric efficiency
	pressure	
hydraulic valves	flow-in	valve-coefficient
	flow-out	flow-return
	pressure-out	pressure-in
	power-out	power-in
 speed controlled motors	power type	rpm
	voltage	efficiency
	current	torque
	power	

Figure 4: Some test parts

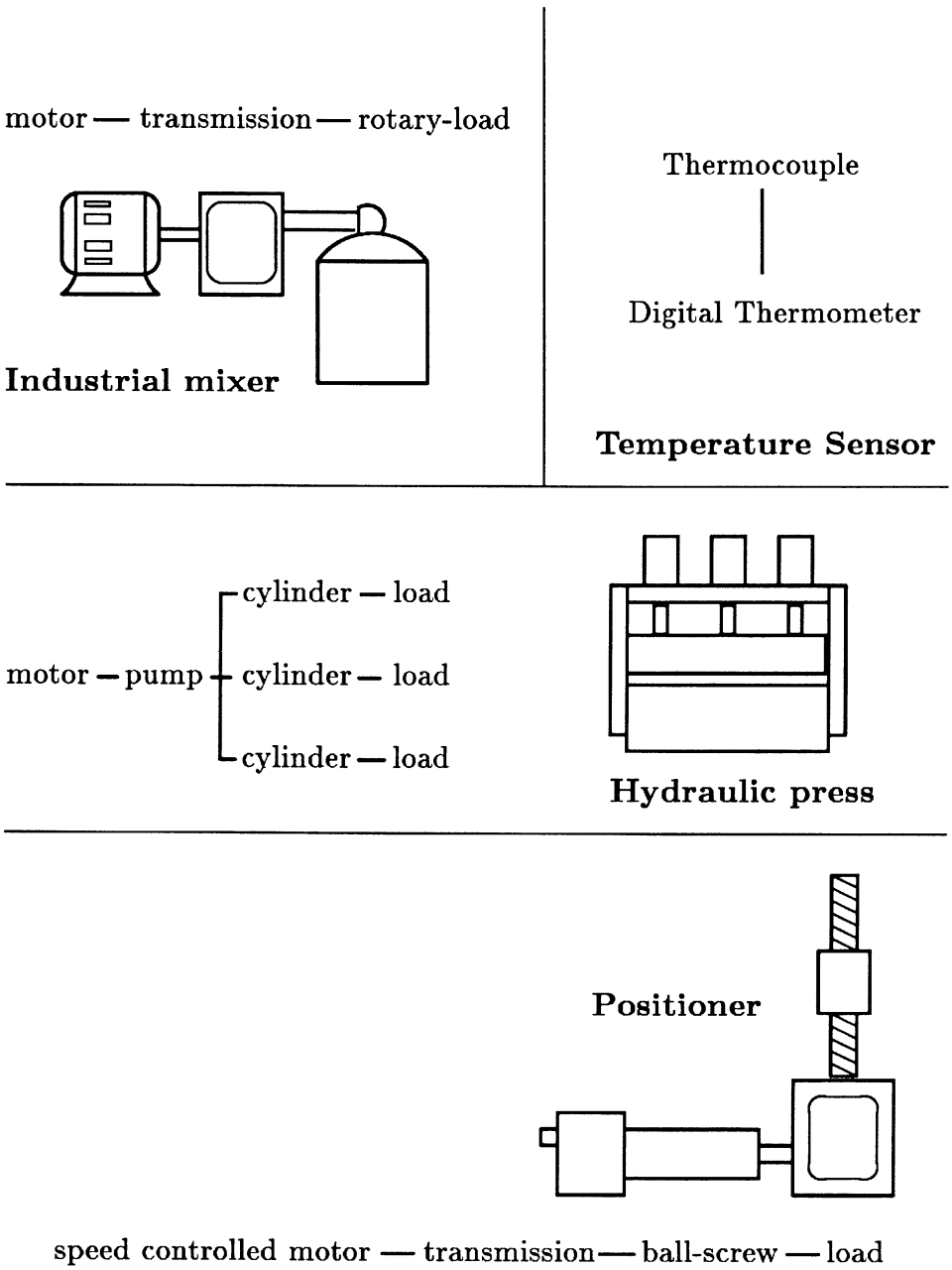


Figure 5: Some test designs

The left hand side of the rule matches the input data and the equation:

$$\begin{array}{lcl} \langle \mathbf{R} \overset{every}{\leftrightarrow} speed\ 0\ .2 \rangle & \sim & \langle \mathbf{R} \overset{every}{\leftrightarrow} x\ x_l\ x_h \rangle \\ \langle \mathbf{R} \overset{every}{\leftrightarrow} force\ 0\ 1000 \rangle & \sim & \langle \mathbf{R} \overset{every}{\leftrightarrow} y\ y_l\ y_h \rangle \\ hp - \frac{(force)(speed)}{550} = 0 & \sim & g(x,y,z) = 0. \end{array}$$

The RANGE function on the right side of the rule is one of three operations on equations and intervals discussed in [1]. In effect, it solves the equation for the hp, forming $hp = \frac{(force)(speed)}{550}$. It then determines the range of the horsepower subject to the constraints that force and speed are restricted to the intervals [0 1000] and [0 .2]. The numerical results of RANGE are thus identical to those produced by the "constraint propagation of intervals". (The other operations discussed in [1] can be thought of as inverses to RANGE.) But the new specification which would be formulated by the right hand side of the rule , $\langle \mathbf{R} \overset{every}{\leftrightarrow} hp\ 0\ .36 \rangle$, is not a "constraint" in the usual sense of a limit on the values. Rather, it says that the cylinder must have available to it power flows from 0 to .36 horsepower; higher powers are acceptable as well.

These specifications eliminate many potential implementations; for example, motors unable to supply the required horsepower, adjusted by the efficiency of the pumps. The designer then splits the catalog for one of the components, for example one set of cylinders, generating daughter designs. One daughter design has only large cylinders, the other only small; this starts a new cycle of abstraction and elimination.

On the particular data given, the compiler searched 71 daughter designs, generating 15,663 new specifications in the process. The cost function used was price plus one half weight. The design run took about 20 minutes, a normal time for the program to complete a hydraulic problem of this size. Optimization of the code will speed this considerably. The output for this problem included:

```
The optimum solution, with cost 441.97, is:
For POWER-SUPPLY, US-3PH-220 with cost 0
For MOTOR, 3N593 with cost 192.72
For GEAR-PUMP, TYPE-103 with cost 133.0
For VALVE, TYPE-1 with cost 50.0
For CYLINDER, 1.25 with cost 6.25
For VALVE-2, TYPE-1 with cost 50.0
For CYLINDER-2, diameter 2.0 with cost 10.0
```

5 Assessing Program Reliability

We have used basic set theory, predicate calculus, and analysis to develop formal correctness proofs of most of the individual compiler operations; see [2]. Such proofs add greatly to the reliability of the program, and to our understanding, but they are no better than the assumptions on which they rest; the program must still be tested empirically. We have done dozens of "runs", with varying specifications, on more than a dozen different arrangements of components. We evaluate these runs by determining why particular alternatives are eliminated, and by examining the "optimal solutions" resulting.

The system appears to eliminate only invalid designs. It frequently surprises us, but we always find either a correctable bug, or that our understanding of the design problem was incomplete.

We are also quite sure that the designs selected are "optimal" with respect to the cost function, but our confidence here is based on the simplicity and clarity of the optimization process rather than on empirical results. Finding an optimum solution by hand is extremely slow on even these simple design problems, and no optimization program we know of can easily be set up for problems of this kind. Even an exhaustive check of combinations of components would still involve sets of

operating conditions, hence require most of the mechanisms of the compiler and not constitute an independent check. The most we can say, as a human designers, is that the designs produced look like they could well be optimal.

A subtler question is whether the program eliminates all the implementations it should—whether its rule set is complete enough to guarantee that the designs it produces will work. It is not, in three senses. First, we know that there are propagation operations we have not yet implemented. We implement operations only as needed, because new operations slow the system and require testing. Second, as we discuss later, the compiler does not propagate every specification it could.

Third, and pragmatically most important, the selected design can always be unsatisfactory because of criteria not represented in the component models. Our formalism imposes restrictions on the criteria it can represent. In particular, equations must be algebraic, and have three variables, though intermediate variables can be used to break up complex equations. We must be able to solve for each variable, and the resulting functions must be continuous and monotonic. The equations must be "instantaneously true"; they cannot values which occur at different times. Values must be non-negative. Specifications must be stated as equations, cost expressions to be minimized, or "hard-edged" intervals. Finally, variables must be divided into only two "causal categories"—parameters, which are fixed at manufacturing, and state variables, which change during operation.

These restrictions limit expressive power. Lack of differential equations probably prevents the system from compiling servo-system designs, or detecting vibration problems. Speed controller catalogs often provide ratios between the highest and lowest controllable speeds, thus relating two different operating conditions. An attempt to model automobile seat design failed because seat-back position is neither a state variable nor a parameter.

Nonetheless, within the domain and problems we have implemented the system appears to select correct designs. It is at present probably less reliable than a very skilled designer working on familiar problems, because very skilled designers make use of information omitted from the catalogs. However, it is probably more reliable, faster, and more likely to produce an optimal design than the average designer.

6 Time Complexity

How long does it take to solve these design problems? "About 20 minutes for a problem involving half a million alternatives" is correct but not very useful, since this depends mostly on implementation and hardware. What we really want to know is how the time required to solve the problem increases as the size of the problem increases.

6.1 Theoretical Results

We will consider two measures of the size of the problem. The first is the total number of possible alternatives, where an alternative is a combination of catalog numbers without regard to feasibility. This is proportional to C^n, where n is the number of components in the design, and C the average catalog length for each component.

The program searches for an optimal solution by creating a binary search tree; the forks in the tree are generated by dividing the catalog for a single component into two parts, splitting the "artifact space". The program then pursues the "most promising" daughter design. There is no guarantee that the "most promising" decision will be correct, and unless it is correct most of the time, back-tracking may require time at least proportional to the number of alternatives.

The situation grows even worse when we consider the other measure of size, that is the number of equations involved. The compiler subsumes the conventional constraint propagation of intervals, and it can be shown [7] that the constraint propagation of intervals can run forever. For example, suppose we have two equations, $x = y$ and $x = 2y$, and we start with intervals $0 \leq x \leq 1$ and

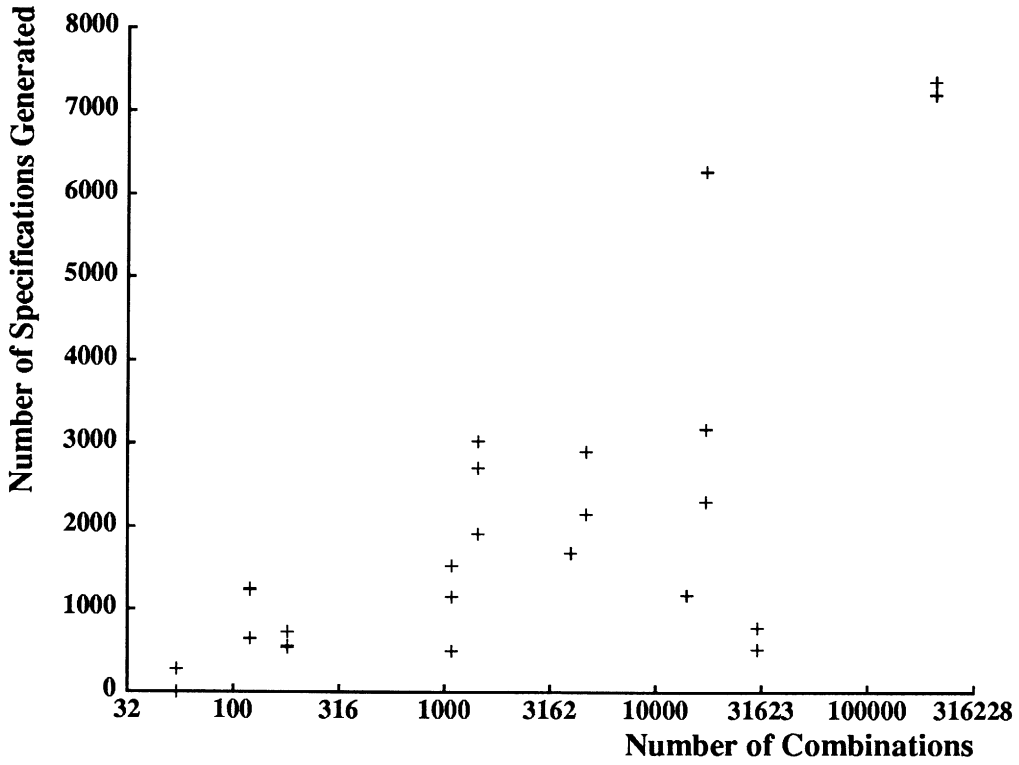


Figure 6: Specification generation vs alternatives for a variety of designs

$0 \leq y \leq 1$. We first conclude from the second equation that $0 \leq y \leq .5$, then from the first that $0 \leq x \leq .5$, then $0 \leq y \leq .25$ and so on. We never arrive (barring round-off error) at the solution, $x = y = 0$.

It may be possible to avoid such pathological cases in real design problems, but even much simpler forms of constraint propagation can require time double-exponential in the number of variables involved, and therefore singly exponential in the number of equations[7].

6.2 Empirical Results

Fortunately, the system actually performs much better than the worst case theoretical projections. In figures 6 and 7, we have used the number of specifications generated by the searching compiler as the measure of time; this measure is independent of the particular hardware and software implementation. Most of the compiler's operations take time proportional to the number of specifications generated. One, the elimination of alternatives, can at worst take time proportional to the number of specifications generated times the average length of the catalogs.

Figure 6 shows a semi-logarithmic plot of the number of specifications generated against the number of alternatives. At worst, the number of specifications generated grows according to the logarithm of the number of alternatives.

Figure 7 shows a plot of the number of equ ations involved in the design against the number of specifications generated; growth is no worse than linear with the number of equations. This, in turn, is linear in the number of components in the design.

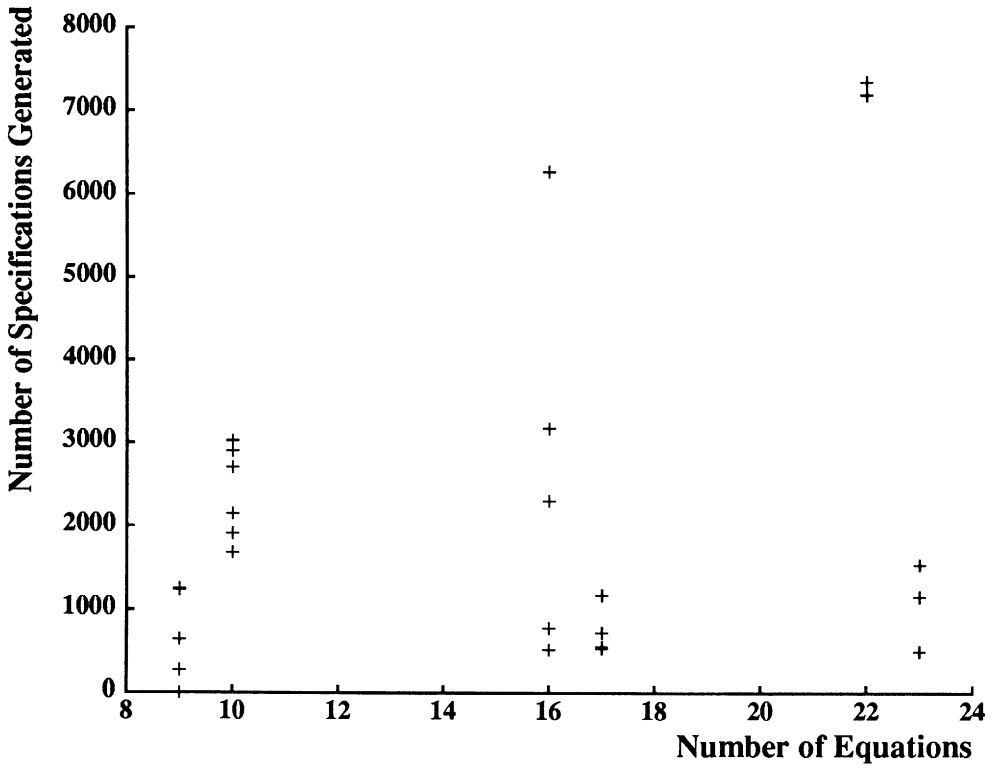


Figure 7: Specification generation vs equations for a variety of designs

6.3 Explaining the difference

There seem to be five principal reasons why the empirical results are so much better than the worst case predictions.

First, note that eliminating a single catalog number eliminates many alternatives, since that catalog number is involved in a combinatorial set of alternatives.

Second, the artifact space is organized, for example by horsepower. A single specification can eliminate many catalog numbers. More importantly, the optimal solution generally involves the smallest (or nearly the smallest) of the devices meeting the horsepower requirement. Since these are clustered together in the search space, only a few branches of the search tree need be followed.

Third, the equations used to describe mechanical components establish a fairly sparse network between variables. In particular, all information passed between components is channeled into a small number of "port variables", such as rpm and torque. (These components have been selected for manufacturing and cataloging in part because they have relatively simple connections with the rest of a design.) This sparseness helps limit the growth in execution time as a function of the number of equations.

Fourth, some of the propagation operations are correct only if each input specification is independent of the other variables in equation used. The compiler in fact requires independence for all propagation operations, thus preventing infinite loops of the kind discussed above.

Fifth, the compiler propagates only the "strongest" specifications, for example the tightest required limits.

These last two reasons involve restrictions on the constraint propagation process. We have not proven that these restrictions cannot cause failures to eliminate, but have not observed any such errors in practice.

7 Conclusions

To summarize, the compiler has been tested on a range of mechanical and hydraulic power transmission designs; new designs can be entered by the designer in minutes. Results have been correct and optimal for the tested problems. Time required for solution grows reasonably slowly as the problem grows. These results are evidence that the theory outlined in [1] is both essentially correct, and useful.

References

[1] Allen C. Ward and Warren Seering. Quantitative Inference in a Mechanical Design Compiler. memo 1062, MIT Artificial Intelligence Laboratory, November 1988.

[2] Allen C. Ward. *A Theory of Quantitative Inference for Artifact Sets, Applied to a Mechanical Design Compiler.* PhD thesis, Massachusetts Institute of Technology, 1989.

[3] David Serrano and David Gossard. Constraint management in conceptual design. In *Knowledge Based Expert Systems in Engineering, Planning and Design.* Computational Mechanics Publications, 1987.

[4] R. J. Popplestone. The Edinburgh Designer system as a framework for robotics: the design of behavior. *Artifical Intelligence for Engineering Design, Analysis, and Manufacturing*, 1(1), 1987.

[5] Kenneth L. Meunier and John R. Dixon. Ierative respecification: A computational model of hierarchical mechanical system design. In *ASME Computers in Engineering Conference.* ASME, 1988.

[6] Jack Mostow, Lou Steinberg, Noshir Langrana, and Chris Tong. A domain-independent model of knowledge-based design: Progress report to the National Science Foundation. Technical Report Working Paper 90-1, Rutgers University, 1988.

[7] Ernest Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32, 1987.

CS-TR Scanning Project
Document Control Form

Date : 4/9/95

Report # AIM-1084

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 12(17-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1) UN #'ED TITLE PAGE	
(2-12) PAGES #'ED 1-11	
(13-14) SCANNER TARGET, DOD	
(15-17) TRGTS(3)	

Scanning Agent Signoff:

Date Received: 4/19/95 Date Scanned: 4/20/95 Date Returned: 4/27/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

completed 8/3/89

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AIM 1084	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD-A209540
4. TITLE (and Subtitle) The Performance of a Mechanical Design "Compiler"		5. TYPE OF REPORT & PERIOD COVERED memorandum 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Allen C. Ward and Warren Seering		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0685
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE January 1989 13. NUMBER OF PAGES 12
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

constraint propagaton
design
quantitative inference
qualitative reasoning

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A mechanical design "compiler" has been developed which, given an appropriate schematic, specifications, and utility function for a mechanical design, returns catalog numbers for an optimal implementation. The compiler has been successfully tested on a variety of mechanical and hydraulic power transmission designs, and a few temperature sensing designs. Times required have been at worst proportional to the logarithm of the number of possible combinations of catalog numbers.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 4/20/1995

M.I.T. Libraries
Document Services