Constraint Reasoning and Planning in Concurrent Design

V. Krishnan *
D. Navinchandra +
P. Rane *
J. R. Rinderle *

CMU-RI-TR-90-03 2

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

28 February 1990

+ Robotics Institute, CMU

* Department of Mechanical Engineering, CMU

Table of Contents

Constraint Reasoning and Planning in Concurrent Design

Abstract

By concurrent design we mean, in part, concurrent consideration of a broad range of life-cycle constraints concerning, for example manufacturing and maintenance. The multitude of constraints arising from these considerations make it difficult to identify satisfactory designs. An alternative to explicitly considering all constraints is to determine which of the constraints are relevant, redundant or inconsistent and to consider only those which impact design decisions.

The proposed approach is based on two simple ideas: (1) Constraints provide a uniform representation for a variety of life-cycle concerns, and (2) Interval methods applied to constraints can be used to identify critical constraints, eliminate redundant constraints and to narrow the space of design alternatives.

The application of the *necessary* and *sufficient* intervals of constraints and constraint propagation techniques are used to classify constraints in this way and to focus design activity. Regional monotinicity properties are used to identify critical constraints.

A related aspect of concurrent design problems is the large number of complex constraints which have to be satisfied to complete a design task. As it is impossible to guarantee the simultaneous solution of a large set of design constraints, we have investigated algorithms for planning and simplifying such constraint problems.

Chapter 1

Introduction: Concurrent Design

The practice of design is frequently sequential in nature. In the design of a jet engine turbine disk, for example, the aerodynamic shape of the blade might first be determined, later modified to satisfy structural constraints, and then further modified to satisfy manufacturability and maintenance considerations.

It is not surprising that such a situation exists, since there are few individuals capable of bringing a full range of life-cycle concerns to bear during design. Nevertheless, the fact that manufacturing and maintenance considerations are introduced only on an ad hoc basis during preliminary design gives rise to fundamental design deficiencies. It is the purpose of concurrent engineering design to include a broad range of functional and life-cycle concerns during preliminary design phases. While it is possible to obtain an appearance of concurrence by rapidly iterating through the basic sequential design process, we seek a greater degree of concurrency by attempting to identify critical life cycle concerns early and to use those concerns to direct design decisions.

1.1 Representing Life-Cycle Concerns

Life-cycle concerns impose required relationships among features of the design to effect functionality, manufacturability, reliability, and servicibilty. In the context of engineering design, these required relationships can be thought of as constraints among design features. Constraints may embody a design objective (e.g. weight), a physical law (e.g. $F = ma$), geometric compatibility (e.g. mating of parts), production requirements (e.g. no blind holes), or any other design requirement. We express constraints as algebraic relations among feature parameters (e.g. hole diameter, wall thickness, stress level). Collectively, the constraints define what will be an acceptable design. Constraint based representations provide a uniform representation for a variety of design considerations including function, geometry, production and disposal. Because there is a single, uniform representation for all constraints there is no differentiation between functional, geometrical, manufacturing, and other, so called, life-cycle constraints. Methods used to refine the design by processing constraints are applied uniformly to all life-cycle constraints: All constraints, whether they be behavioral, geometrical or those which have traditionally been considered *down-stream*, have equal impact on design decision making. It is for this very reason that our approach achieves concurrency.

Although constraints are a general mechanism to represent design considerations, it is not possible to identify all design constraints at the time the design problem is first proposed. This is because the set of relevant constraints depends on the design context. If the geometry of the designed artifact is such that casting is an appropriate manufacturing method, then casting constraints are required. Alternatively, a set of machining constraints is necessary if the part is to be machined. Similarly, there are constraints that

are dependent on material, assembly methods, and a host of other considerations. The relevant constraints depend on the current design features. The features themselves may be completely defined aspects of a detailed design or they may be partially specified characteristics of a general configuration. Because constraints are required relationships among feature parameters they may be retracted, augmented or refined as the design evolves. The design can be thought of as being complete when the set of constraints stabilize and when all the constraints have been satisfied. If we assume for the remainder of this article that constraints can be expressed as algebraic relationships among feature parameters, then we can say that a design is complete when all variables have been assigned values and when all the constraints have been satisfied simultaneously.

1.2 Automation in Constraint Based Design

The design of certain small mechanical components can often be effectively accomplished by executing an established design procedure. Automating such procedures is a common approach to automating the design of components such as gears, however, as the complexity of the component itself, or the scope of considerations increase, it is difficult to identify practical design procedures.

An alternative to coding comprehensive design procedures is to identify isolated procedures and to determine sequences for executing the procedures until all design variables are established. Certain rule based design systems and some *constraint propagation* systems follow this paradigm. Each rule or constraint can be thought of as a procedure indicating how some specific design decision or variable can be determined once other design parameters are known. The rules, or constraints, can then be executed, in sequence, until the design is complete. This method, which requires a dependence ordering or topological sorting of constraints, is the basis for a family of commercial products. It is effective for certain classes of designs, most notably when the degree of component interaction is small and when the design is fundamentally serial in nature. It is *not* effective when the description is fundamentally *what needs to be achieved* rather than *means for achieving it.* Such a descriptive (rather than prescriptive) approach to design specification is advantageous because the designer does not need to consider how a specific constraint will be satisfied, thereby facilitating the inclusion of constraints which cannot readily be interpreted as procedures. Furthermore, descriptive specification does not impose a causality to the constraints. For example, if a simple disk was constrained to weigh a specific amount, the constraint itself could be interpreted as a procedure to determine diameter, or thickness, or density given the other two. In this way the constraints imposed on a design serve as both a specification of the design and as a set of procedures for determining design parameters. The design task, in affect, is reduced to that of satisfying the given constraints.

Constraint satisfaction, however, is not easy. Design constraints are usually numerous, complex and highly non-linear. Satisfying a large set of arbitrarily complex equality and inequality constraints is, in essence, the non-linear programming problem and, in general, is not solvable. Although the general problem cannot be solved, much can be done to assist the designer. It is possible to provide the designer with insights about critical interactions among features, redundant requirements and inconsistencies. Such information is useful to the designer even if the constraints are ultimately solved numerically because a purely numerical solution does not facilitate understanding the subtleties of a specific design task.

A large body of research exists on solving constraint problems. The SKETCHPAD [Sutherland 83]system was an early effort on solving constraints by propagation and relaxation. Mackworth [Mackworth 77] introduced algorithms for maintaining consistency in a network of constraint relations. The ThingLab research effort [Borning 79] lead to ideas on propagating constraints across part-whole hierarchies of objects. A constraint representation formalism was introduced by Sussman and Steele [Sussman 80]. Recently, Gosling [Gosling 83] presented a planning technique which, coupled with propagation, helps solve algebraic constraints. Other relevant work on solving sets of algebraic equations has come from Popplestone [Popplestone 80] and Serrano [Serrano 87]. These research efforts provide a core of solution techniques for handling and propagating variables with exact values. Unfortunately, many if not most of the engineering design constraints are expressed as inequalities. The very nature of constraints is such that they often do not prescribe specific values for design parameters but rather prescribe ranges for the values. To accomodate inequality constraints while maintaining uniformity of representation, we choose to represent all feature parameters as interval values. Conventional parameter assignment, e.g. $L = 1.5\,in.$, can be expressed as assignment to a narrow interval, e.g. $L = [1.495, 1.505]$, explicitely representing the scale of indifference or perhaps manufacturing tolerance. Inequality constraints, e.g. $L \geq 10ft.$, may b e expressed as an open interval, e.g. $L = [10, \infty]$[1], making explicit the (as yet) unbounded range for a parameter value. Interval specification is also convenient for expressing constraints which are left implicit such as the positive value requirement, e.g. $L \leq 10ft. \rightarrow L = [0, 10]$. Relationships among feature parameter values are also conveniently expressed as interval assignments, e.g. $L_1 \geq L_2 \rightarrow L_1 - L_2 = [0, \infty]$.

The ideas presented in this paper are based on treating design parameters as intervals. The notion of interval arithmetic was developed by Moore [Moore 66, Moore 79]. The value of interval based methods for design has also been recognized by Ward [Ward 89]. The interval representation of values generalizes the notion of equality assignment, provides a mechanism to deal with tolerances and adds flexibility, making it possible to capture incompleteness and uncertainty in a design.

A design which is not yet complete may have some parameters which have not been assigned exact values and there may be some uncertainty about the final design characteristics. Intervals may be used to express upper and lower bounds on parameter values, making it possible to estimate some properties of the artifact before exact values are assigned. This information can sometimes be used to guide preliminary design, augmenting the *rules of thumb* or *back of the envelope calculations* commonly employed by designers. Furthermore, interval based representation is a convenient framework for implementing and bounding *order of magnitude* analysis and default sizing. In this way many levels of specificity may be used simultaneously at any point in the design process. By representing all levels of specificity as intervals and using a uniform technique for propagating the intervals through the constraints, we are able to evaluate the constraints on the design and provide the designer valuable feedback about potential constraint violations.

---

[1]This is an open interval and should correctly be written as $[y, +\infty)$. Because the only open intervals which we consider are unbounded and because the apparent mismatch in open interval delimiters is often confusing we have chosen to ignore this fine distinction.

Interval Methods for Constraint Reasoning

Introduction

The desire to address a broad range of life cycle issues in a nonsequential fashion during the early design stages introduces two main difficulties: 1) The multitude of constraints obscure the designer's understanding of the design by increasing the dimensionality of the problem. and 2) The dimensionality of the problem increases the complexity of finding a design solution. It is not clear to the designer which constraints are important and which ones are irrelevant. There is a need to sift the model and identify

1. The active constraints that decide where the solution lies.
2. The critical constraints that determine the solution and preserve the integrity of the model, and
3. The irrelevant constraints which can be deleted from the model.

Reasoning with a set of multi dimensional nonlinear constraints, is not only a formidable task to the human designer but also a drudgery to his creative and flexible mind. However it is essential that the trends and trade-offs inherent in the physics of the device be available to the designer in the early stages of the decision making process. There is a need for a systematic methodology to

1. Identify the critical considerations that direct the design and
2. Reason with the constraint based design model to garner insights about the model without having to choose specific values for the design parameters.

One approach to identifying the relevant and critical constraints is Monotonicity Analysis. Unfortunately most engineering design constraints do not exhibit the global monotonicity required for the application of Monotonicity Analysis. It is therefore necessary to exploit *regional* properties of functions and reassemble the regional information to draw global inferences. We seek methods to

- Extract and utilize regional information
- Address constraint evaluation, and propagation of decisions in the preliminary stages of design.

We propose an Interval Analysis based methodology to accomplish these objectives.

In this chapter we explain the interval based methodology to reason with constraints. First, we describe the concepts of constraint activity,criticality and dominance and the identification of these properties using Monotonicity Analysis. Conditions under which monotonicity analysis succeeds and its limitations also come under this discussion. Secondly, we introduce interval methods to represent, manipulate and propagate regional information and we briefly discuss the mathematics of interval methods along with some problems associated with them. Thirdly, we discuss the utility of interval methods in handling constraint sets with many inequalities(as are common in design). We also address the refinement of decisions through Interval propagation amongst constraints thereby detecting inconsistencies in the preliminary stages. Fourthly , we describe the enhancements to interval methods to detect regional constraint dominance, activity and criticality. Lastly, we delve on the applications of interval methods to global optimization problems.

1.3 Context and Definitions

Consider a stage of design when the concept and the configurations to meet the design requirements have already been synthesized and studied, resulting in a set of constraints.[2] This set of constraints is referred to as a model. A model in which all variables are restricted to physically realizable values is said to be *bounded.*[3]

The goal is to obtain a satisfactory design that optimally satisfies the design objective. The following definitions are in order.

Let x be a vector = $(x_1, x_2, x_3, \ldots x_{k-1}, x_k, x_{k+1}, \ldots x_n)$ where $x_1, x_2, \ldots x_n$ are the design variables.
A function $f(x_1, x_2, \ldots x_n)$ is monotonically increasing with respect to x_i, if an increase in x_i does not result in a decrease in f.

A monotonic variable is one that is monotonic in *every* function in the model.

An *active* constraint is one whose presence influences the solution of the model. An active constraint is a relevant constraint but need not be tight i.e satisfied as an equality at the design solution.

Constraint 1 *dominates* Constraint 2 if the feasible region of constraint 1 is a subset of the feasible region of constraint 2. Satisfaction of constraint 1 implies satisfaction of constraint 2.

An active constraint satisfied as an equality constraint at the design solution is called a critical constraint. A critical constraint is an active constraint, but an active constraint need not be critical.[4]

The following example clarifies these definitions.

---

[2]By constraints, we mean, a required relationship among design objectives and variables. We limit our discussion to algebraic constraints.

[3]In particular, design variables should not reach the values 0 or ∞.

[4]Not all tight constraints are critical. Consider Minimizing $f=(x-5)^2$, subject to $x^2=25$; The minimum is at 5 but is not changed when the constraint is removed. So the equality constraint is not active and therefore not critical.

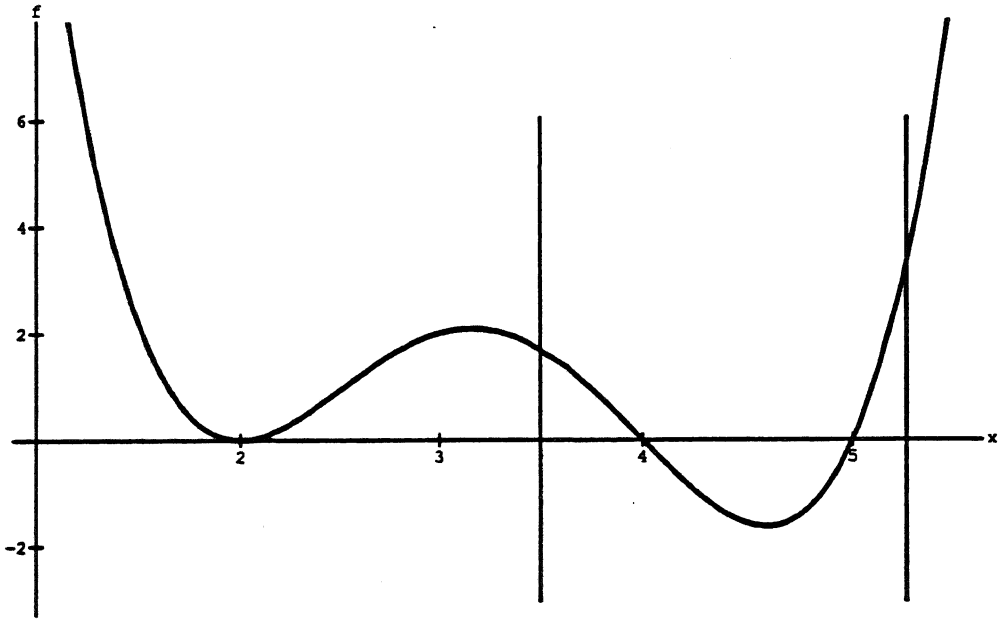

Figure 1-1: Plot of $f = (x-2)^2(x-4)(x-5)$ subject to constraints $c_1 = x-3.5 \leq 0$ *and* $c_2 = x-5.25 \leq 0$

- The function f is monotonically increasing with respect to the design variable x in the region x=2.0 to x=3.0. The unconstrained minimum occurs at x=4.6
- Now let the constraint c_1: $x - 3.5 \leq 0$ be introduced. The constrained minimum is at x= 2.0; c_1 is an active constraint, as its presence influences the solution. c_1 is not a critical constraint as it is not satisfied as a tight constraint at the design solution.
- The constraint c_1 dominates the constraint c_2: $x - 5.25 \leq 0$, as satisfaction of c_1, means satisfaction of c_2. So the dominated constraint c_2 can be deleted from the model.

1.4 Monotonicity Analysis

Monotonicity analysis has been investigated by Wilde et. al, [Papalambros and Wilde 88]. to verify and simplify the formulated model. They argue that improper bounding usually arises from unnoticed monotonicity in the mathematical functions of the model and expound a set of principles that summarize the boundedness requirement of the mathematical model . These are the First and Second Monotonicity principles, that can be used to identify the critical and relevant constraints in the model.

Monotonicity Principle1: In a well constrained objective function, every monotonic variable in the

objective should be bounded by at least one active constraint.[5]

Monotonicity Principle2 : Every monotonic nonobjective variable in a well-bounded problem is either

1. Irrelevant and can be deleted from the problem together with all constraints in which it occurs, or
2. Relevant and bounded by two critical constraints,[6] one from above and one from below.

The utility of the Monotonicity Principles is in proving criticality and irrelevancy of constraints. This can result in the deletion of constraints and reduction in size and complexity of the model if the variables are globally monotonic. The Monotonicity Principles are used to solve the hydraulic cylinder problem given below. [Papalambros and Wilde 88].

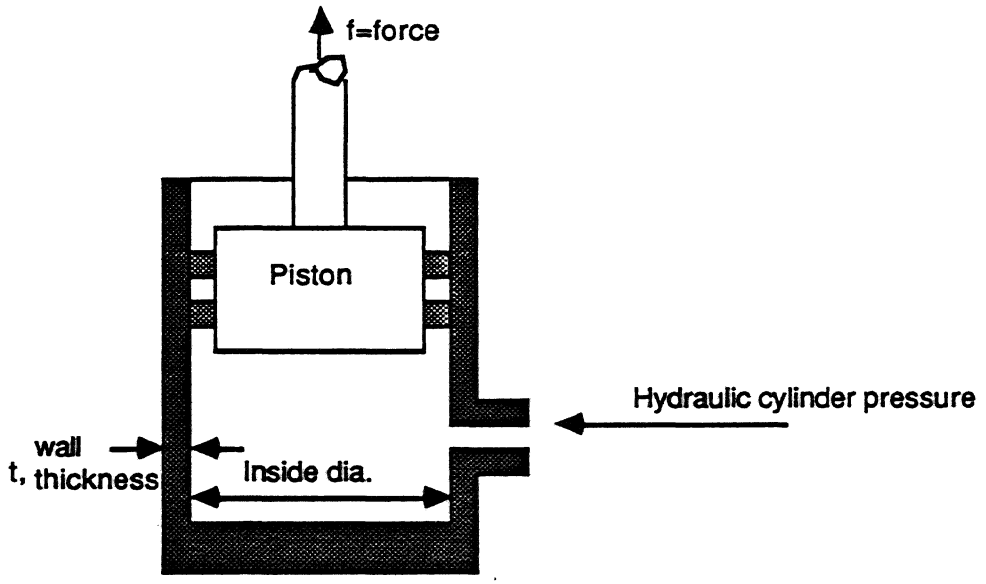

Figure 1-2: Hydraulic Cylinder

Notations : d_o = outside dia.; d_i = inside dia.; s = hoop stress; t = thickness;

Goal : To design the hydraulic cylinder so as to meet the following *functional specifications*

- F= Load Handling Capacity $\geq$ 22 pound wt.(10 kgs)
- P= Pressure $\leq$ 3.5 psi.

Minimize $d_o = d_i + 2t$, subject to the **Constraint Set**

1. $-F + 22 \leq 0$

---

[5] If a set of constraints bind the monotonic objective variable, the dominant constraint in this set is a critical constraint.

[6] [Papalambros and Wilde 88] use the term "active" to mean what we are calling "critical".

2. $p - 3.5 \leq 0$

3. $s = \frac{p\, d_i}{2t}$

4. $s \leq S_{yp} = 8000(psi)$ (Material Strength constraint)

5. $F = \frac{p\ \pi d_i^2}{4}$

6. $t \geq 3$ (mm) (Manufacturing constraint)

All the variables are monotonic in this model. Reasoning using Monotonicity principles proceeds as follows.

- Because the stress s does not appear in the objective, from Monotonicity Principle 2, if s is relevant, then it must be bounded by one constraint from above and one from below . Constraint 4 binds s from above . The only other constraint in which s appears, constraint 3, should bind s from below. So constraint 3 becomes a directed equality. constraint 3 :-

  $$s \equiv > \frac{(p\, d_i)}{2t}$$

  thereby binding s from below.

- The objective variable d_i cannot be bounded from below by constraint 3, due to the directionality on constraint 3 imposed by step 1. The only other constraint in which d_i occurs, constraint 4 , must be critical , i.e satisfied as an equality.

- The criticality of constraint 4 makes F and p relevant variables. By Monotonicity Principle 2, these relevant variables have to be bounded by two critical constraints: one from above and one from below. So constraints 1 and 2 are critical yielding p=3.5(psi) and F=22(lbs). >From constraint 5, d_i=72 (mm).

- From Monotonicity Principle 1, one of the two constraints, 4 or 6 has to be critical to bind t from below. Using s=8000(psi), and results from step 3, constraint 4 becomes $t \geq 0.015$(mm). Therefore constraint 6 becomes critical and t=3mm.(since satisfying constraint 6 implies satisfaction of constraint 4 but not otherwise.)

- The **Solution** : p=3.5(psi), F=22(lbs), t = 3(mm), d_i = 72(mm).

Thus Monotonicity Analysis was able to provide a closed form solution to the hydraulic cylinder problem due to the global monotonicities of objective and constraints with respect to the design variables. However if such global monotonicities do not exist, it is not possible to obtain a closed form solution using Monotonicity Principles. Consider the design of the cylinder for a high Pressure application where the cylinder must be thick walled. Then the maximum hoop stress for a thick-walled cylinder, at the inner diameter, is given by:

$$\frac{s}{p} = \frac{d_o^2 + d_i^2}{d_o^2 - d_i^2}$$

ince $d_o = d_i + 2t$ we have:

$$\frac{s}{p} = \frac{(d_i+2t)^2+d_i^2}{(d_i+2t)^2-d_i^2}$$

we use Al alloy A96061 with S_{yp}=8000(psi) and p=4000 (psi) then the $s \le S_{yp}$ yields,

$$\frac{s}{p} \le \frac{S_{yp}}{p}$$

id therefore

$$2 \ge \frac{(d_i+2t)^2+d_i^2}{(d_i+2t)^2-d_i^2}$$

hich becomes $4t^2 + 4d_i t - 2d_i^2 \ge 0.0$. This resultant constraint is not globally monotonic with respect to

easoning using Monotonicity Principles will not work because d_i is not a globally monotonic variable.

lthough the newly introduced constraint was not globally monotonic, it still is monotonic in the regions $< d_i$ and $t \ge d_i$. If $t \ge d_i$ is unreasonable in the domain of application, then the solution can be obtained y solving the problem in one region. Else, the problem is solved separately in the two regions, and the olution assembled to obtain the global solution. Most design objectives are too complicated to be lobally monotonic, but do vary monotonically over regions. Similarly in real design problems, different onstraints may become active and dominant in different regions; hence great leverage can be obtained by cploting regional information. We need means for representing, abstracting and manipulating regional ıformation. The need is met by the application of Interval methods.

.5 Interval Methods

ıterval Methods provide a convenient framework to characterize regional properties of objectives and onstraints. An interval is a set $[a, b]$ such that all real numbers between a and b are included in the set. ıtervals can be operated on by set theoretic operators such as intersection, union and subset. An interval f a function provides upper and lower bounds for the range of the function, when its arguments span an ıterval. For eg. the interval of the function $(x^2 + y)$ for the interval $x = [1, 4]$ $y = [5, 10]$ is $[6, 26]$. This nplies that all the values taken by the function $(x^2 + y)$ for the given range of arguments are above 6 and elow 26.

ıterval arithmetic is used as the basis for evaluating algebraic relations containing interval variables, ielding interval results. Interval arithmetic operators are defined on the upper and lower bounds of the perands. The interval on $(x^2 + y)$ in the above example, was determined by expanding the square perator and applying the following interval arithmetic formulae. [Moore 66]

$$[a, b] + [c, d] = [a + c.\ b + d] \quad (1)$$

$$[a, b] - [c, d] = [a - d,\ b - c] \quad (2)$$

$$[a, b] \times [c, d] = [min(ac, ad, bc, bd),\ max(ac, ad, bc\ bd)] \quad (3)$$

$$[a, b] \;/\; [c, d] = [a, b] \times [1/d, 1/c]^{7} \tag{4}$$

By using Interval methods, we can characterize regional monotonicities, regional feasibilities etc. of design constraints. For example consider the newly introduced constraint in the hydraulic cylinder problem:

$$c = 4t^2 + 4d_i t - 2d_i^2 \geq 0.0;$$

$$\frac{\partial(c)}{\partial(d_i)} = 4t - 4d_i$$

In the region t=[3, 10] and d=[35, 40],

- The interval on the partial of c is calculated by using the principles of interval arithmetic to be = [–148,–100] < 0.0 So the constraint is monotonically decreasing with respect to the inner diameter, d_i.(Regional Monotonicity)
- The interval of the function c itself in this region = [–2744, –450] < 0; So the constraint is violated in the whole of this region. (Regional infeasibility).

We shall see in a later section how the properties, regional constraint activity, criticality and dominancy can be redefined through intervals.

1.6 Conservativeness of Interval Calculations

The four basic arithmetic operators, are monotonic with respect to each of their arguments. Further more, the sum and difference operators have known monotonicities and therefore it is possible to determine a priori which combinations of interval endpoints must be evaluated to determine the maximum and minimum interval of the function. The multiplication operator however, may have a strictly increasing or decreasing monotonicity depending on whether or not the arguments are positive or negative. In this case we must evaluate the multiplication function at each combination of interval end points and select the minimum and maximum to determine the interval. Nevertheless, these basic operators compute intervals which are *precisely* the interval that occurs.

Although the four basic arithmetic operators produce exact intervals, the representation of higher level functions in terms of these basic arithmetic operators introduces some difficulty. Consider the function of $y = (x–2)^2$ over the interval x = [0, 10]. The function itself is not monotonic over that interval. Since subdivision into monotonic intervals would require in the general case difficult solution procedures we prefer to express the function in terms of the monotonic arithmetic operators, i.e. $y = (x-2)(x-2)$. In this case the square operator has been replaced with multiplication, however, implicit in the definition of interval multiplication is that the two arguments may vary independently. This is clearly not the case for the square function which we are discussing, however, since the independence of the arguments is less restrictive, the result of applying the conventional interval arithmetic will result in an interval on y which is conservative in the sense that the actual interval on the function will lie within the computed interval. In this case the interval computed using interval arithmetic is [–16, 64] which includes the actual interval

[7]The division operator is not valid when the interval of the divisor includes zero.

of [0, 64]. The conservative interval calculation destroys the one to one correspondence between intervals on arguments and intervals on functions. This is important in the context of design because it is often necessary to determine what range of arguments will satisfy a range on the function itself. The extent to which the computed interval deviates from the actual interval is critical to the degree to which strong inferences can be made regarding intervals on variables.

There are some specific techniques intended to mediate against the expansion of intervals. One such approach is the centered form of functions based on a fourier expansion of the intervals and is described in [Moore 79]. Other heuristics, for example, to deal with even exponents are also useful. There are several ad-hoc methods to obtain less conservative intervals, often exact intervals. Since the computation of intervals is not the focus of our research, it will not be discussed at greater length here.

1.7 Constraint Propagation in Design

Intervals can be effective for representing and reasoning about design parameter values. It is also possible to propagate interval values through a set of constraints and detect potential constraint violations. By propagating design decisions through constraints it is possible to determine how the various design parameters affect one another. In the process, redundant constraints are identified and eliminated. The intervals of the various parameters are also refined in this process.

Consider, for example, a DC motor. The torque (T in-oz) is related to speed (ω rad/sec) as shown in Figure 3 and as given by the constraint:

$$T = 100 - \frac{1}{5}\omega$$

Assume that the torque must be at least 30 in-oz (.21 N-m) and must not exceed 75 in-oz (.53 N-m) and that the speed may assume any value between 150 and 400 rad/sec. The given interval, [30, 75 in-oz],[8] in conjunction with the motor characteristics imposes upper and lower bounds on speed of 125 and 350 rad/sec as shown in Figure 3. Intersecting this interval with the original interval we obtain a refined interval on speed, [150, 350 rad/sec]. This new interval is propagated through the constraint, once again, to find upper and lower bounds on torque, [30, 70 rad/sec]. This interval on torque and the corresponding interval on speed indicate that the original specifications requiring torque to be less than 75 in-oz and speed to be less than 400 rad/sec were not necessary. By propagating intervals it was possible to identify redundancies and therefore simplify the design task without making specific commitments about any of the design parameters.

The process of propagating intervals through constraints can be continued through long chains of constraints. The process provides a means for determining bounds on design variables thereby delimiting a feasible space for the final design. Propagation can be done through chains of constraints resulting in a successive narrowing of parameter intervals. Continuing our example, assume the power of the motor (given by $Power = \omega T$) is required to be less than or equal to 8500 in-oz/second (60 W), that is, in the

[8] The S.I. units are not generally repeated in the interval notation to avoid confusion.

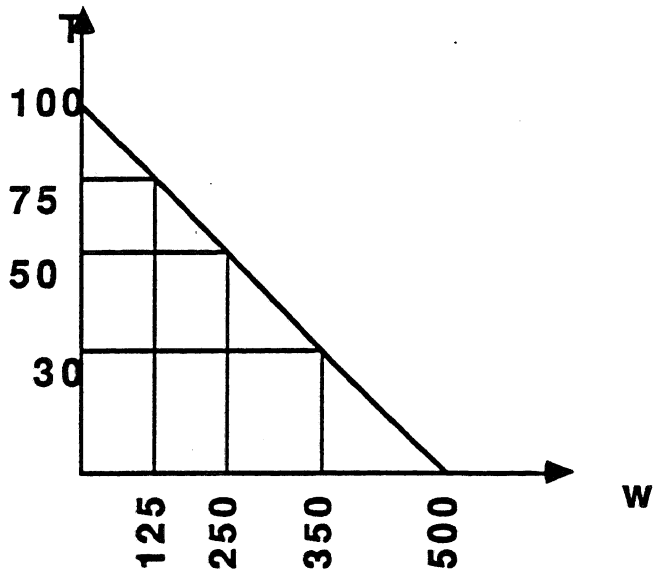

Figure 1-3: Torque Speed Characteristics of a D.C Motor

interval [0, 8500]. Holding the interval on Power and propagating T and ω through the two constraints yields the interval [222, 252] for the speed and the interval [30, 38.2] for torque. This narrowing requires about 20 iterations. By propagating intervals successively, any variable can affect any other variable as long as there is a chain of constraints connecting them. Propagation can occur in any direction; it is not the case that one variable in a constraint must be selected, a priori, as being dependent while all others are regarded as independent. As constraints are propagated and as intervals narrow, specifications may be found to be inconsistent with other constraints thereby identifying violations[9] and redundancies before specific design decisions are made. Interval propagation makes it possible to gain insight about a design without having to choose specific values for the design parameters.

Working with intervals, in this way, allows one to simultaneously consider a wide range of alternatives and to examine interactions among design parameters before the design is completed. The example also shows how it is possible to narrow design choices without actually committing to any single operating point. We believe that the ability to draw important inferences about a design problem, early in the process is important in concurrent engineering. Later in this paper, we show how interval based propagation methods can be used to provide a designer feedback about the likely violation of life-cycle constraints, even when the design is incomplete. We believe this is a viable way of achieving concurrency in design.

---

[9]That is, when an interval is reduced to a null set.

1.8 Interval propagation

In this section we delve in more detail on the propagation of intervals through a set of constraints and the evaluation of intervals through necessary and sufficient conditions. Consider the evaluation of intervals using the basic interval arithmetic operations. For example, let V_3 be an interval calculated from the equation V_1 ***op*** $V_2 = V_3$. Where, ***op*** is one of the four basic interval arithmetic operators. This operation guarantees that for any value in the intervals V_1 and V_2 the result of applying ***op*** will be in V_3. In other words, the result is *necessarily* in the interval V_3.

After a constraint expression is evaluated the new interval is propagated. For example, when a new interval is determined for a variable, that variable's current interval is updated by intersecting it with the calculated necessary interval. If the intersection is null, then the original interval or the constraint is said to be inconsistent. This kind of propagation can be carried through complex equations using the interval arithmetic operators. The process guarantees the result will necessarily be in the calculated interval.

In the DC-motor example, we have the equation $Power = \omega T$. Assume we know the interval on Power: [8000, 25000] (inch-oz/sec) and Torque [30, 75] (inch-oz). We seek an interval on ω such that, for any values of power and torque (within their intervals) the speed, ω, falls within the interval. In other words, we seek an interval in which ω has to *necessarily* be in. As shown in Figure 4 , the speed must fall between *a* and *b*. The interval may be computed using the basic interval arithmetic to evaluate the constraint, expressed terms of the variable in question: $\omega = \frac{Power}{T} = \frac{[8000, 25000]}{[30, 75]} = [106, 833]$.

1.9 Necessary and Sufficient Intervals

The fact that a variable falls within a necessary interval does not guarantee that all constraints can be satisfied, i.e, *necessary* does not imply *sufficient*. Consider for example the situation depicted in Figure 4. Although ω[106, 833 rad/sec] is necessary, an arbitrary value in this interval is not sufficient to satisfy the power requirement for arbitrary values of allowable torque since the rectangle of valid torques and speeds extends beyond the bounding power curves. Even when we know that both torque and speed fall within the necessary intervals it is still necessary to check that the power constraint is satisfied. There may, however, be an interval on speed which guarantees that the power requirement will be satisfied whenever torque requirements are met. We say that such an interval is *sufficient* to satisfy the constraint. For example, the interval on speed *sufficient* to satisfy the power requirement is shown in Figure 4. For any value of torque and speed in their respective intervals, the power will always be between 8000 and 25000 in-oz/sec.

The concept of necessary and sufficient intervals can be very useful to the designer. If two constraints each have associated with them a necessary interval on the same argument and those necessary intervals do not overlap it is not possible to simultaneously satisfy both constraints. The ability to identify a constraint contradiction of this sort early in the design cycle makes it possible for the designer to determine appropriate relaxations of these constraints.

The interval of some variable, sufficient to satisfy a constraint insures that a constraint will be satisfied whenever all of the other variables fall within their necessary intervals. Therefore, if the necessary

interval of one constraint falls completely within the sufficient interval of a second constraint, then that second constraint will be unconditionally satisfied whenever the first is and therefore the second constraint does not need to be considered explicitly. Identifying a redundant constraint of this sort is similarly useful to the design since only those constraints which are truly binding need be considered.

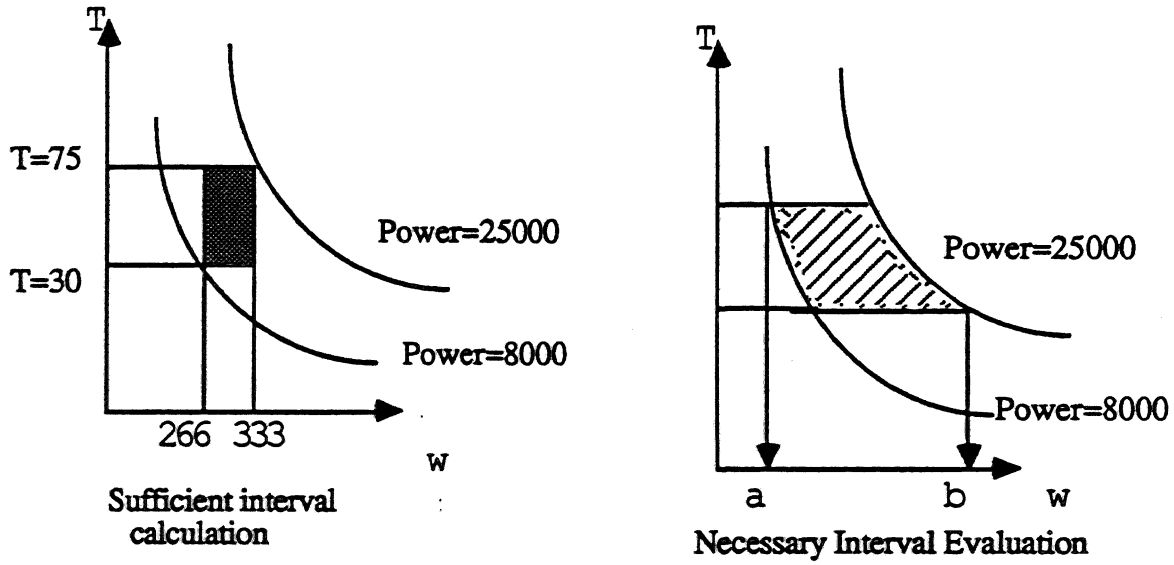

Figure 1-4: Constant Power Curves for a D.C Motor

In the DC-motor example, where $Power = \omega T$, suppose that the interval on Power is prescribed as: [8000,25000] (inch-oz/sec) and Torque [60, 70] (inch-oz). The sufficient interval on ω is [133, 357]. Suppose the Torque-Speed characteristics of the d.c motor are given as before:

$$T = 100 - \frac{1}{5}\omega$$

The interval on ω necessary in light of torque, is [150, 200] (rad/sec). Since the necessary interval due to the Torque Speed characteristics constraint is a subset of the sufficient interval on the power constraint, the Torque speed characteristics constraint dominates the power constraint. This is one way of showing constraint dominance. We shall provide a stronger necessary condition that dominant constraints obey in a later section.

The necessary and sufficient intervals are of course not unique.

- Any sub-interval of a sufficient interval is also a sufficient interval.
- A super-interval of any necessary interval is also a necessary interval.
- The union of all sufficient intervals are contained in any necessary interval.

1.10 Calculation using the Sufficiency Condition

In this section we present a method to evaluate the intervals on a variable based on a sufficient condition.

Our goal is as follows: Given the constraint V_1 *op* V_2 = V_3, to determine V_2 such that for any value in given V_1,the application of op yields a result in specified V_3. In other words, what should V_2 be so that for any value in V_1 , V_1 *op* V_2 lies in the specified interval V_3?

To obtain the unknown interval V_2, we express the relation V_1 *op* V_2 = V_3 in terms of the interval we want to be within : in this case V_3. Assume V_2 = [v_{2l}, v_{2u}], where v_{2l} and v_{2u} are the values we seek. Now consider the relation V_1 *op* V_2 = V_3 . The left hand side can be evaluated in terms of the unknowns v_{2l} and v_{2u} by applying the interval operators. These two unknowns can be found by solving the interval equation, as demonstrated by the following example : Consider the D.C motor case in which Power is required to be in the interval [8000, 25000] and the interval on Torque is given to be [30, 75]. Our goal is to find an interval on ω such that *Power* = ωT is satisfied for all ω and T in their respective intervals. Following the above procedure: Let $\omega = [\omega_l, \omega_u]$. Applying the equation *Power* = ωT; substitute the intervals for Power, Torque and ω we get [8000, 25000] = $[\omega_l, \omega_u]$ [30, 75] from which it follows that [8000, 25000] = [30 ω_l,75 ω_u], since torque and speed are known to be positive definite. Equating the upper and lower limits of the interval equation, 8000= ω_l 30 gives ω_l= 266 and 25000 = ω_u 75 gives ω_u= 333 . These limits on speed give the sufficient interval.

Now consider the dual case, that of finding an interval on speed sufficient to satisfy the torque requirement of [30,75] given that power falls within the stated interval of [8000, 25000]. Now, V_3 is Torque and V_1 is Power. Expressing the equation in terms of Torque, Torque = Power/ω; we get [30, 75] = [8000, 25000]/ $[\omega_l, \omega_u]$, from where it follows that 30 = 8000/ω_u and 75 = 25000/ω_l. The limits on speed are hence: ω_u = 266 and ω_l = 333 , which is a nonsense interval. There is no interval on speed sufficient to guarantee that there is some valid torque for any power in the stated interval.

This example not only demonstrates the simple methodology to evaluate sufficient intervals but also illustates the asymmetric nature of sufficiency intervals and their conditional existence. The existence of a sufficient intervals has an impact on the design decision making. For example, if we are designing a d.c motor, the existence of a sufficient interval means the ability to accomodate any motor in the given torque range or the need to use a particular motor in the given torque range. Conditions for existence of sufficient intervals, resolutions and retractions needed for their existence are topics of current research.

1.11 Interval Criticality, Dominance, Activity

Constraints in design may not be globally monotonic, globally active, dominant or critical, but certainly are regionally. The concepts of constraint criticality, dominance and activity, defined over regions, are therefore, more effective in identifying the critical constraints and pruning the insignificant ones. Interval Methods, with which we represented and manipulated regional information in the previous sections, can again be used to characterize dominant, critical and active constraints regionally.

Interval Dominance Constraint $c_1 < 0$ dominates constraint $c_2 < 0$ in the interval I, if the interval $(c_2 - c_1)$

is necessarily <0.

Constraint c_1 dominates c_2, if the feasible region of c_1 is a subset of the feasible region of c_2. In particular if $c_2<c_1$, then c_1 dominates c_2, since satisfaction of $c_1 < 0$, implies that $c_2 <0$. So if $c_2-c_1<0$, then constraint c_1 dominates c_2. Regionally, this becomes interval (c_2-c_1) is necessarily <0 ,for dominance of c_1 over c_2.

Interval criticality An inequality constraint that is satisfied as a tight constraint for a variable x_i, over the interval I is termed Interval Critical.[10].

Interval Activity A constraint that influences the solution within the interval I is termed Interval Active.

The interval dominance conditions stated in this section are stronger conditions than the ones stated using necessary and sufficient conditions in Section 5.[11]. Consider the application of the interval dominance conditions to the d.c motor problem where the following constraints apply:-

c_1... $Power = \omega T < 25{,}000$;

and the motor Torque-speed characteristic constraint

$$c_2 \ldots \quad T = 100 - \frac{1}{5}\omega$$

with Torque constrained to be in the interval [30,75];

The sufficient interval from c_1 on ω is [0,333], while the necessary interval on ω using c_2 is [125, 350]. Since the necessary interval from c_2 is not a subset of the sufficient interval from c_1, dominance of c_2 over c_1 cannot be proved by the theory of necessary and sufficient intervals.

However if we used the interval dominance conditions stated in this section,

the Interval (c_1-c_2) = Interval(ωT-25,000 - (25 - 0.2 ω))

= Interval($(T+0.2)\omega - 25{,}025$) is necessarily < 0 for $\omega < 333$;

Intersecting with the necessary interval on ω from c_2 we get, ω = [125,333]; in this interval, constraint 2 dominates constraint 1.

Let us consider the utility of these properties in solving the hydraulic cylinder problem.

Hydraulic Cylinder Revisited The two constraints involving thickness were:

1. c_1 $4t^2 + 4d_i t - 2d_i^2 \geq 0.0$;
2. c_2 $t \geq 3.0(mm)$;(from a manufacturing consideration)

One of these two constraints must be critical, according to Monotonicity Principle 1. Suppose that for this heavy load application, $t > d_i$ is inapplicable. Then in the interval $t < d_i$ the new constraint is

---

[10]Over a region, a model is automatically bounded by the set constraints, defining the region. So an interval critical constraint is one that dictates what the regional solution is, rather than one that keeps the model bounded

[11]The interval dominance conditions are sufficient conditions, but not necessary conditions, as they are based on interval calculations.

monotonically decreasing; the new constraint cannot bind d_i from below. The monotonic objective variable is again bound from below by the relationship between force and d_i, and so d_i=72 (*mm*), *F*=44,000(*lbs*) *p*=4000 (*psi*).
The Constraint c_1 becomes $4t^2 + 288t - 10368 \geq 0.0$;

$$c_2 - c_1 = 10365 - 287t - 4t^2;$$

$$\text{interval}[\, c_2 - c_1 \,] \text{ for t=[3, } t_u]$$

$$= [10365 - 287t_u - 4t_u^2,\ 10365 - 287t_l - 4t_l^2]$$

which is necessarily > 0 for $t_u \leq 27mm$.
So in the interval [3, 27] constraint 1 dominates constraint 2.
At $t = 27(mm)$, constraint c_1 becomes tight and dominates c_2.
The solution is t=27 (*mm*), d_i = 72 (*mm*) *and* d_o = 126 (*mm*), for this is the smallest value of t for which both constraints are satisfied.

Thus, despite the absence of global monotonicities, the design solution was achieved by combining regional information. This suggests that one could use interval methods not only for constraint reasoning but also for optimizing the objectives. In fact interval methods guarantee that the resultant interval obtained is an inclusion, i.e. the result includes and binds all values of the function in this region. Interval Methods based algorithms use this assuredness property of intervals to obtain the global optimum. In the next few sections we will investigate the application of interval based methods to

- Obtaining the globally optimum solution and
- Combining constraint reasoning with global optimization.

1.12 Global Optimization

Global Optimization of a nonlinear, nonconvex objective subject to nonlinear constraints is yet an unsolved problem. There is no *single best* method to accomplish this goal of attaining the global optimum. The problem with most traditional nonlinear programming techniques is that they are local methods. They can get stuck in local valleys, and there is no guarantee that the solution is globally optimum. Under strong assumptions about the function involved, like convexity etc. the solution can be assured to be globally optimum.

Interval methods have been used to solve the global optimization problem [Ratschek and Rokne 88]. The rationale behind these approaches for unconstrained optimization is as follows:

- Use interval methods to represent regional information.
- Exploit the bounds provided by the interval method as a part of a branch and bound search strategy.
- Combine with a subdivisioning procedure, that helps accelerate the search, by yielding tighter bounds.

To solve the constrained optimization problem, these methods subdivide the constrained design space into halves, until they get a part of the space which satisfies all the constraints. Due to the the extreme

conservativeness of interval calculations, and the non linearity of the constraints, it is very difficult to obtain a region that satisfies all the constraints through interval calulations.
Consider the two constraints $f_1 \geq (x-2)(x-3)$ *and* $f_2 \leq 5x-6.25-x^2$. as shown in Figure 5.

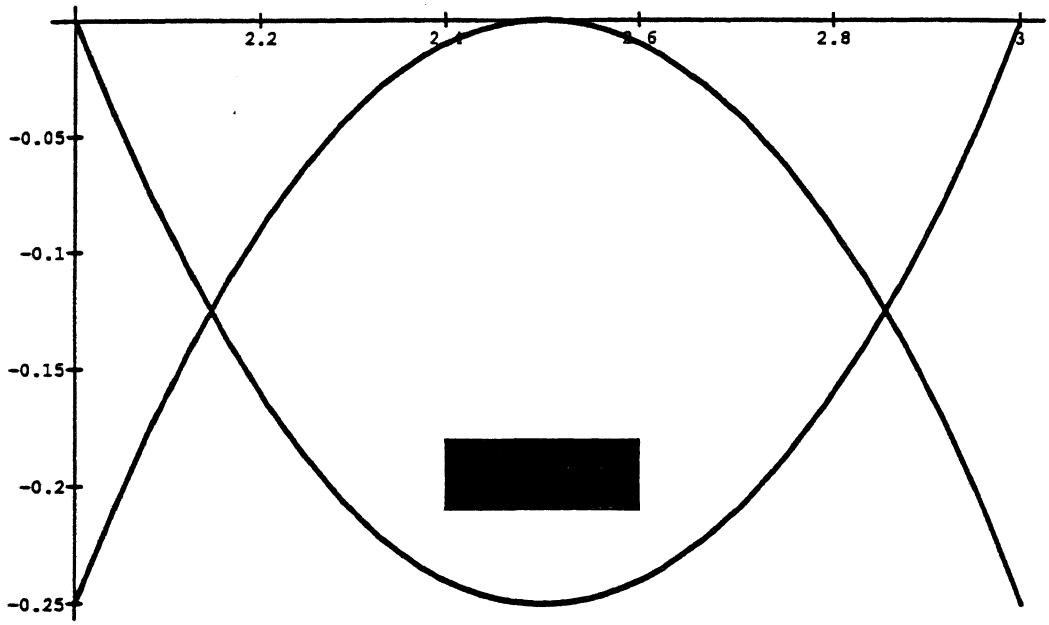

Figure 1-5: $f_1 = (x-2)(x-3)$ *and* $f_2 = 5x-6.25-x^2$.

Consider the small window represented by x=[2.4, 2.6] and y= [–0.21,–0.18]. Although the rectangle shown is clearly feasible, doing interval analysis on this rectangle cannot show that this rectangle is feasible.
Interval[$(x-2)(x-3)$] for x=[2.4, 2.6] = [–0.36, –0.16].
Y- Interval[$(x-2)(x-3)$] for x=[2.4, 2.6] = [–0.05, 0.18], which is not necessarily greater than zero. So according to interval analysis, this region may not be completely feasible.
Despite subdividing the originally available region to very small regions we cannot prove that the small region is going to satisfy both the constraints, although in reality it does. So the region which satisfies all constraints is going to be difficult to obtain, and may be too small to be of any engineering use.

On the other hand it is not necessary that each and every constraint be satisfied a priori in every region through interval calculations. A large portion of the constraints are dominated in some regions and can be deleted from that region. So great gains may accrue if constraints are reasoned within each of the regions, and model simplified , before optimizing in that region.
Combining constraint reasoning with interval based methods for global optimization, results in the Interval Variable approach for global optimization.

1.13 Interval Variables Approach

Unlike conventional interval algorithms which keep sub-dividing the design space until all constraints are satisfied, the Interval variables approach simplifies the model regionally by reasoning from a variable point of view. The interval dominancy, criticality and activity conditions, as defined in an earlier section, are used to reason from the variable point of view. The Interval Variable approach, by itself, may not be able to solve all the problems completely. In such cases it may be used as a pre-processor that simplifies the model.

Interval Variables approach

1. Form the adjacency matrix for the model.[12]
2. Consider nonobjective variables first, giving preference to those variables that figure in the least number of constraints.
   - If the variable occurs in only one constraint, and is regionally monotonic in that constraint, eliminate the constraint and the variable.
   - If the variable occurs in several constraints, and is regionally monotonic with respect to each of the constraints, then call these constraints, nonobjective conditionally critical,(exactly one constraint in this group is critical). Apply Interval Dominance conditions to all pairs of constraints in this conditionally critical set to identify the critical constraint. While considering the various pairs, dominance relations are propagated,i.e. If A dominates B, and B dominates C, then A dominates C; The pair A,C need not be tested for dominance.
   - If the nonobjective variable is not monotonic, subdivide the design space further, to obtain desired monotonicities.

Consider monotonic objective variables, starting first with variables that occur in the least number of constraints. For each variable,

- Partition design space to obtain desired monotonicities.
- List all the constraints with the desired monotonicities under an objective conditionally critical set.
- Test for interval dominancy of constraints.
- Delete the dominated constraints. Apply Monotonicity Principle 1 to obtain the constraint which is regionally critical.

In a larger sense, the interval variable approach is similar to active set strategies, as it tries to solve the problem by finding out the critical constraints. However instead of expecting a few constraints to be critical throughout the design space, the interval variables approach looks for regional criticality. In highly nonlinear situations, such as design constraints and objectives, we believe this results in significant benefits.

---

[12]Adjacency matrix lists the numbers of the constraints and the variables that occur in each of the constraints. It is described in the next chapter.

1.14 Weldment Design using Interval Variables approach

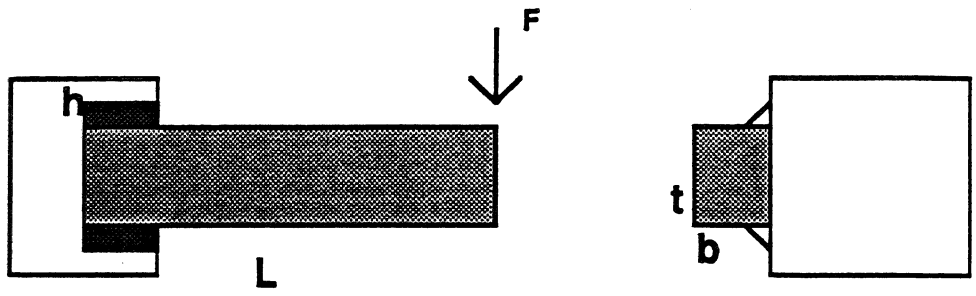

Figure 1-6: Design of Bar Weldment

The rectangular bar supporting a vertical force on one end is to be welded at the other end to a metal column. [Ragsdell and Philips 76]. The distance of the metal column to the force F is denoted by L. The four design variables in the problem are weld length l, weld rod thickness h, bar width b and bar depth t. The objective to be minimized is the sum of weld, stub and bar costs

$f = C_1 h^2 l + C_2 lbt + C_3 bt$

where $C_1 = 1.1047$; $C_2 = 0.6735$; $C_3 = 0.04811$. subject to

$g1 = -hl + 1.5211 \leq 0.0$ ... shear stress

$g2 = -bt^2 + 16.8 \leq 0.0$ .. bending stress in bar

$g3 = -b + h \leq 0.0$ .. weld width < bar width

$g4 = -bt^3 + 9.08 \leq 0.0$ .. end deflection

$g5 = 0.094t + (0.02776/b^3 t) \leq 0.0$ ..buckling

$g6 = -h + 0.125 \leq 0.0$ ..weld width's lower bound.

[Papalambros and Wilde 88] used the variable transformation $a = bt$, where a is the vertical cross-sectional area of the bar. They eliminate b using the transformation and obtain the following set of constraints.

$f = C_1 h^2 l + C_2 la + C_3 a$

The constraints become

$g1 = -hl + 1.5211 \leq 0.0$

$g2 = -at + 16.8 \leq 0.0$

$g3 = -a + ht \leq 0.0$

$g4 = -at^2 + 9.08 \leq 0.0$

$g5 = -1 + 0.02776t + (0.094 t^2/a^3) \leq 0.0$

$g6 = -h + 0.125 \leq 0.0$

Start with the following intervals:-

$h = [0.1 , 1]$

$t = [1 , 3]$

$b = [3 , 10]$

$l = [3 , 30]$

$a = bt = [3 , 30]$

Step 1 Select the only nonobjective variable, t , for consideration. The constraints in which it appears are as follows:-

$g2 =$ $-at + 16.8 \leq 0.0$

$g3 =$ $-a + ht \leq 0.0$

$g4 =$ $-at^2 + 9.08 \leq 0.0$

$g5 =$ $-1+0.02776t + (0.094 t^2/a^3) \leq 0.0$ and the set constraints

$t \geq 1.0,\ t \leq 3.0$

On grouping the constraints into those that bind the variable t from above and those that bind from below, we obtain

Set A(binds t from above)

$g3 =$ $-a + ht \leq 0.0$

$g5 =$ $-1+0.02776t + (0.094 t^2/a^3) \leq 0.0$ and

$t-3.0 \leq 0.0$

Set B(constraints providing lower bounds)

$g2 =$ $-at + 16.8 \leq 0.0$

$g4 =$ $-at^2 + 9.08 \leq 0.0$

$-t+1 \leq 0.0$

Because t is a monotonic nonobjective variable, it has to be bounded by two critical constraints, one from above and one from below. So exactly one constraint from each set must be critical.

By Interval Dominance we can show that the set constraint $t-3.0 \leq 0.0$ dominates the other two constraints in set A. So it is critical and $t = 3.0$;

This makes sure that $-t+1 \geq 0.0$ is not critical. Using t=3.0 and the interval dominance condition we can show that

$g2 =$ $-at + 16.8 \leq 0.0$ dominates

$g4 =$ $-at^2 + 9.08 \leq 0.0$ over the specified interval of a.

g4 can be deleted and g2 is critical resulting in a=(16.8/3)=5.6;

h is bounded from below by two constraints. The set constraint $h \geq 0.1$ and

the constraint $g6 =$ $-h + 0.125 \leq 0.0$;

Obviously g6 dominates the set constraint and so h= 0.125;

Thus the solution in this region of the design space is $h = 0.125$, $a = 5.6$, $t = 3.0$, $b = 1.9$.

1.15 Conclusions

In this chapter the utility of interval methods to reason with constraints was presented. Intervals help abstract, represent and manipulate regional information. They also help characterize regional properties. They can easily describe incomplete designs and can accomodate any level of specificity in the values of the variables. Furthermore, their assuredness property, makes them useful candidates for a methodology to be used in global optimization.

Chapter 2

Planning Constraint Solution Strategies

In concurrent design problems we often have large numbers of complex constraints which have to be satisfied to complete a design task. As it is impossible to guarantee the simultaneous solution of a large set of design constraints, we have investigated algorithms for planning and simplifying such constraint problems.

Satisfying a large number of constraints does not imply that all the constraints be solved simultaneously. Often, some parts of the design tend to be more coupled than others. This chapter presents algorithms for finding the coupled constraints and for developing a solution strategy which minimizes simultenaety.

The simplest type of constraint sets are those which do not need any simultaneous solution of constraints. Such constraint sets are said to be *Serially Decomposable*. The constraints can be solved serially, yielding the value of one new variable for each constraint evaluation. We present algorithms to detect serial decomposability and for ordering the solution sequence of such constraint sets. When a constraint set is not serially decomposable, the constraints have to be solved for simultaneously. Instead of trying to solve the entire constraint set simultanecusly, we would like to isolate and identify subsets of the entire constraint set which necessarily have to be solved simultaneously. This chapter presents algorithms for achieving this.

One of the assumptions we make in ordering algebraic constraints is that they are invertible. That is, for any function $F(X)$, one can find the value of any variable x_i in X if the values of all the other variables are known. Not all constraints are explicit and not all constraints are invertible. For example, a Finite Element package, unlike an algebraic relation, takes inputs and produces outputs. One cannot determine the inputs from the outputs. Such constraints have to be handled in a special way. We present an algorithm for ordering constraint sets which contain both reversible and irreversible constraints.

2.16 A Design Example

Before embarking on discussions about graph theory and algorithms, let us examine the major ideas of this chapter with the aid of a simple, but illustrative example. We will be using this example throughout this chapter.

Consider the design of a friction-type disc clutch shown in Figure 2-7. This example is taken from [Hindhede, Et.al. 83]. The problem is to find the size of the clutch plate (D_{out}), and the inner diameter of the lining (*Din*), to safely transmit 32 HP at 3000 rpm.

There are several design equations relating the known and unknown clutch parameters. We treat these

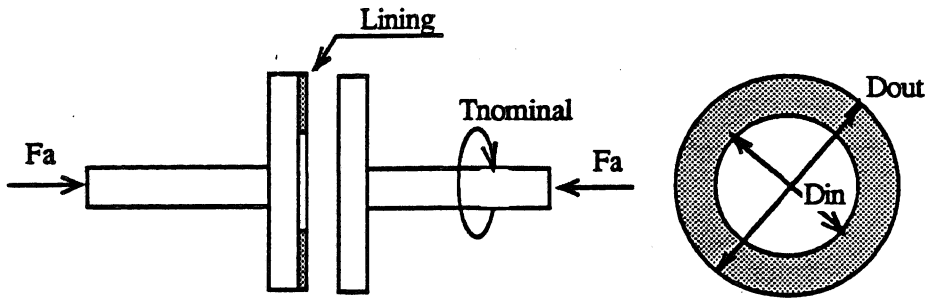

Figure 2-7: Clutch Example

equations as constraints. In other words, the equations are used both as procedures for calculating unknowns and also as relations among parameters. These relations have to be satisfied in the final design. For the clutch example, we will be using the following seven constraints:

$$T_{design} = \frac{F_a \mu D_e}{3} \tag{5}$$

where,

T_{design} = Design Torque (N.m)
F_a = Actuating Force (N)
D_e = Effective Diameter (mm)
μ = Coefficient of friction

The effective friction radius is the average radius

$$D_e = \frac{D_{out} + D_{in}}{2} \tag{6}$$

The actuating force depends on the allowable contact pressure ($P_{allowable}$) of the materials chosen for the clutch plates.

$$F_a = 0.25\,\pi\,(D_{out}^2 - D_{in}^2)P_{allowable} \tag{7}$$

The nominal torque ($T_{nominal}$) is increased to the design torque (T_{design}) using a service factor (K_s). The service factor depends on the type of load, drive and starting mode [Gagne 53]. We will assume $K_s = 2.5$.

$$T_{design} = T_{nominal}\ K_s \tag{8}$$

The nominal torque is given in terms of power to be transmitted (*Power*) and the speed (ω)

$$Power = \omega T_{nominal} \tag{9}$$

Based on past experience, the ratio of inner and outer diameters (D_{ratio}) is fixed at 1.5

$$D_{ratio} = \frac{D_{out}}{D_{in}} \tag{10}$$

Finally, we would like to check the centrifugal hoop stress S_{hoop} on the clutch.

$$S_{hoop} = \frac{\rho D^2_{out}\,\omega^2}{4} \tag{11}$$

The design task is to find values for the unknown variables such that the above constraints are all satisfied simultaneously.

2.16.1 Ordering the constraints

Instead of trying to solve all the above equations simultaneously, one can try to identify a reasonable strategy to solve the equations. The algorithm presented in this paper produces the following strategy:

Step 1: Calculate for $T_{nominal}$ from Equation (9) by expressing the equation in terms of $T_{nominal}$ and substituting for power and angular speed.

Step 2: Calculate for T_{design} from Equation (8) by substituting the just determined value of $T_{nominal}$.

Step 3: Calculate for D_{out}, D_{in}, D_e and F_a simultaneously from equations (5), (6), (7) and (10)

Step 4: Calculate for S_{hoop} from Equation (11)

The above strategy shows three aspects of solution planning: (1) Constraints may be treated as procedures, where any variable in the constraint can be determined if the values of all the other variables are known. (2) Some variables can be determined only after other variables are determined. This produces a chain (or ordering) of constraint evaluations, and (3) Variables which depend on one another have to be solved for simultaneously. The actual numerical method used to solve the constraints is outside the scope of this report. It is our aim, to find a viable solution strategy which identifies and isolates only those constraints which have to be solved simultaneously. This is done to avoid treating the entire constraint set simultaneously.

2.17 Planning Algorithm for Serially Decomposable Constraint Sets

Some constraint sets do not need any simultaneous solution of constraints. We call such constraint sets *Serially Decomposable*. This is because the constraints can be solved serially, that is, there is no simultenaety among the constraints. For example, consider a reformulation of the clutch problem:

$$D_e^3 = \frac{6\,T_{design}}{\pi\,\mu} \tag{12}$$

$$Power = \omega\,T_{nominal} \tag{13}$$

$$S_{hoop} = \frac{9\,\rho\,D_e^2\,\omega^2}{25} \tag{14}$$

$$D_{out} = 1.2\,D_e \tag{15}$$

$$D_{in} = 0.8\,D_e \tag{16}$$

$$T_{design} = T_{nominal}\;K_s \tag{17}$$

$$F_a = \frac{3\ T_{design}}{\mu D_e} \tag{18}$$

This constraint set is equivalent to the original constraint set, but has been manually reformulated to be serially decomposed. The notion can best be explained with using an adjacency matrix representation. Constraints form the rows of the matrix, and the variables correspond to the columns. In every row (constraint), *X*s are marked in the columns which correspond to the variables involved in the constraint. The adjacency matrix is as shown in Figure 2-8 (a). Careful re-ordering of the rows and columns yields the matrix in part (b) of the figure. This matrix shows the order in which the constraints can be solved. As $T_{nominal}$ is the only unknown in Equation (13), it is determined first. Once $T_{nominal}$ is known, T_{design} becomes the only unknown in Equation (17). In this way all the variables can be solved for one after the other, that is, serially.

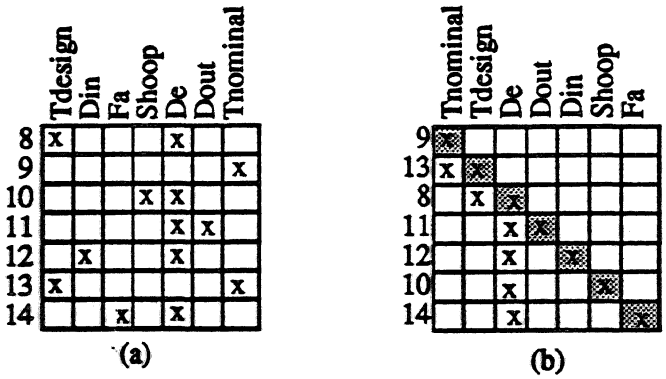

Figure 2-8: A Serially Decomposable Constraint Set

An important property of a serially decomposed set is that the rows and columns can be reordered to yield an adjacency matrix which is populated (with X's) only on and/or under the diagonal. This is what allows us to solve the constraints in a serial order. Every time a variable is determined, at least one constraint becomes solvable (i.e. only one unknown left). If a constraint set is not serially decomposable, it implies that some of the equations would have to be solved simultaneously. In such cases, it is our aim to find an ordering which tries to minimize the total number of equations which have to solved simultaneously.

2.17.1 An Algorithm for Ordering Serially Decomposed Constraint Sets

A serially decomposable constraint set can be ordered using a very simple row and column elimination algorithm.

Step 0. Express the constraint set as an adjacency matrix.
Initialize a stack called ORDER

Step 1. If there are no rows or columns in the matrix, return: ORDER

Step 2. Find all the rows with only one X in it.
If there are no such rows, return "The Matrix is not Serially Decomposable"

Step 3. For all the rows with only one "X" in it:
a. read off the corresponding column (variable name) and push it on the stack ORDER
b. remove the row from the matrix
c. remove the column with the "X" in it.

Step 4. Go to Step 1.

A problem with using the above algorithm, is that it fails ungracefully when the constraint set is not serially decomposed. The algorithm has no way of telling whether a given constraint set is serially decomposable or not, it starts generating an ordering and fails only when it reaches an impasse. It would be better if we could determine, up front, whether a constraint set is orderable or not. The next section presents an algorithm aimed at quickly determining whether a constriant set is serially orderable or not. The algorithm works without actually trying to order the constraints, and thus runs very fast.

2.18 Special Treatment of Serially Decomposable Constraint Sets

The constraint sets encountered in design practice are usually extremely complex. The number of constraints and the number of variables is large. An interesting property of the constraint sets is that they are usually sparse. Each of the individual equations have a small number of variables. A direct simultaneous solution or optimization of the set of constraints is computationally very complex. It is undesirable as it does not take advantage of the sparseness of the constraint set. The sparse set of constraints is broken down in to different groups which can be solved separately and progressively one after another.

A serially decomposable set can be ordered and solved directly without taking recourse to simultaneous solution methods. A constraint set, which is not serially decomposable on the whole, may have parts which are serially decomposable. This property can be used effectively to partition the constraint set into smaller and more managable subsets.

Detection of serially decomposable sets is valuable. At present, the constraint sets are reordered using various algorithms, to partition the system of equations. It would be good to detect before resorting to reordering, whether the constraint set is serially decomposable or not.

An analytical method has been developed to test the serial decomposability of a constraint set. This method is based on an adjacency matrix representation of the constraint set and a set of boolean properties of the set.

Adjacency matrix representation

A constraint set can be represented as a bi-partite graph. The nodes of the graph are the equations and the variables. The edges are between the equations and the variables present in the equations.

The adjacency matrix of the constraint set is matrix formed by presenting the equations along the rows and variables along the columns. An element of a row is 1 if the variable corresponding to the column is

present in the equation, else it is 0. For example, consider the system of constraint equations of the design example in section 2 :

$$T_{design} = \frac{F_a \mu D_e}{3} \quad (1)$$

$$D_e = \frac{D_{out} + D_{in}}{2} \quad (2)$$

$$F_a = 0.25\,\pi\,(D_{out}^2 - D_{in}^2)\,P_{allowable} \quad (3)$$

$$T_{design} = 2.5\,T_{nominal} \quad (4)$$

$$Power = \omega\,T_{nominal} \quad (5)$$

$$D_{ratio} = \frac{D_{out}}{D_{in}} \quad (6)$$

$$S_{hoop} = \frac{\rho D_{out}^2 \omega^2}{4} \quad (7)$$

The adjacency matrix of this system can be written as,

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} (1) \\ (2) \\ (3) \\ (4) \\ (5) \\ (6) \\ (7) \end{matrix}$$

where, the columns are :($T_{nominal}$, T_{design}, D_{out}, D_{in}, F_a, D_e, S_{hoop})

It can be observed that the adjacency matrix is a boolean matrix : its elements can take only the values 1 or 0.

Boolean Determinant of Adjacency Matrix

Definition of a boolean determinant of an adjacency matrix is very similar to that of the determinant of a real matrix. The multiplication operation is replaced by the logical AND operation. The additions are replaced by an operator "nary-XOR", which is an extension of the binary exclusive OR.

Consider a square boolean matrix A of size n.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & & \cdots & \\ \cdots & & \cdots & \\ \cdots & & \cdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

where elements a_{ij} are boolean.

If A is 2 x 2 then,

$\det(A) = (a_{11}$ AND $a_{22})$ XOR $(a_{12}$ AND $a_{21})$.

If A is n x n then

$\det(A) = (a_{11}$ AND minor $(a_{11}))$ XOR $(a_{12}$ AND minor $(a_{12}))$ XOR ... XOR $(a_{1n}$ AND minor$(a_{1n}))$.

= nary-XOR ($(a_{11}$ AND minor $(a_{11}))$,$(a_{12}$ AND minor $(a_{12}))$, ... $(a_{1n}$ AND minor$(a_{1n}))$).

The XOR expansion is defined as n-ary XOR; i.e. before evaluating any of the boolean expressions, the whole expression should be expanded. After the complete expansion, the AND operations should be evaluated. Then the n-ary XOR should be applied to the whole expression. If there is *exactly* one '1' in the expression, the result will be 1, else the expression will evaluate to 0.

The determinant can be expanded by any row or column. Consequently, the value of the determinant does not change on rearrangement of rows and columns.

Criterion for Serial Decomposability

The determinant of the adjacency matrix indicates the presence of loops.

If A is an adjacency matrix, then

if det(A) = 1, the constraints can be serially decomposed,

= 0, the constraints can not be serially decomposed.

Consider the first example. The adjacency matrix on rearrangement :

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \quad \begin{matrix} (5) \\ (4) \\ (1) \\ (2) \\ (3) \\ (6) \\ (7) \end{matrix}$$

where the columns are :$(T_{nominal}, T_{design}, F_a, D_e, D_{out}, D_{in}, S_{hoop})$

Expanding the determinant of A by first row,

det(A) = nary-XOR (1 AND minor(a_{11}))

= nary-XOR ((1 AND minor(a_{22})))

= nary-XOR ((1 AND minor(a_{33})), (1 AND minor(a_{34})))

= nary-XOR ((1 AND minor(a_{44})), (1 AND minor(a_{53})))

= nary-XOR ((1 AND minor(a_{55})), (1 AND minor(a_{56})),

(1 AND minor(a_{45})), (1 AND minor(a_{46})))

= nary-XOR ((1), (1), (1), (1))
= 0

Matrix A is not serially decomposable.

Consider the reformulated system of equations (8) to (14).

The adjacency matrix of the constraint set is given by,

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{matrix} (9) \\ (13) \\ (8) \\ (11) \\ (12) \\ (10) \\ (14) \end{matrix}$$

where the columns are : ($T_{nominal}$, T_{design}, F_a, D_e, D_{out}, D_{in}, S_{hoop})

det(B) = nary-XOR (1 AND minor(a_{11}))

= nary-XOR (1 AND minor(a_{22}))

= nary-XOR (1 AND minor(a_{33}))

= nary-XOR (1 AND minor(a_{44}))

= nary-XOR (1 AND minor(a_{55}))

= nary-XOR (1, 0)

= 1

B is serially decomposable.

These propositions have been developed in a line similar to the theory of system of linear equations. The proofs are not included here.

Evaluation of the determinant qualitatively corresponds to determination of a path through the '1' elements of the matrix. A serial decomposation can be viewed as a determination of a *spanning tree* of the matrix, such that the parts of the path along the rows (equations) are not longer than one edge per row.

A serially decomposable matrix has a path through the all the 1 elements, which is a tree. In a matrix, which is not serially decomposable, such a path does not exist. It contains a path which is a graph with loops rather than a tree. The number of these loops and their inter-relations are an important consideration in a solution of the constraint management problem.

The present work does not include non-square constraint sets and directed constraints. It can be extended to cover these cases. The theoretical approach seems to have a potential to give rise to an analytical formulation for optimal partitioning of a constraint set.

2.19 Ordering a Non-Decomposable Constraint Sets

When a constraint set is not serially decomposable, the constraints have to be solved simultaneously. Instead of trying to solve the entire constraint set simultaneously, we would like to isolate and identify subsets of the entire constraint set which necessarily have to be solved simultaneously. This section presents an ordering algorithm which helps identify such subsets.

Let us return to the original clutch example. The original problem, as we noted earlier, is not serially decomposable. This can be seen with the aid of the adjacency matrix representation. Figure 2-9 (a) shows the matrix for the given equations. By carefully reordering the rows and columns, one can find a solution plan which is better than trying to solve all the variables and constraints simultaneously. Figure 2-9 (b) shows the ordered matrix: after calculating $T_{nominal}$, it is possible to calculate T_{design} from equation 8. The next four variables have to solved as a block (simultaneously).

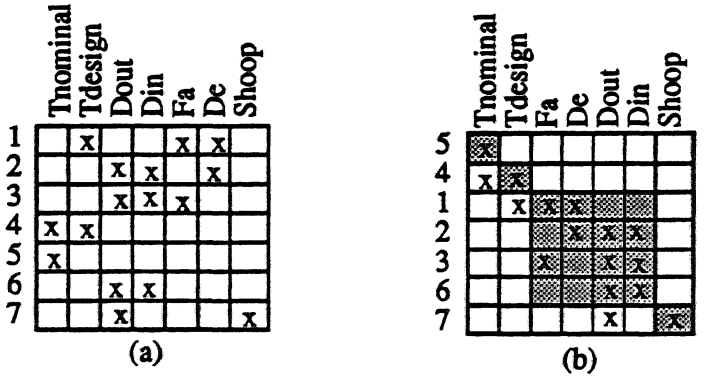

Figure 2-9: Adjacency Matrix Representation of the Ordering Task

2.19.1 Intuitive Explanation

The algorithm consists of two stages: *Matching* and *Component Finding*. The first stage matches variables to equations. This is done because we know that each equation can be used to solve for only one variable. Every variable will be calculated from one constraint. It is for this reason that we start by matching constraints to variables. It is important to try and find as many matchings as possible. For example, in the two equations $F1(x,y)$ and $F2(x)$, there are two variables x and y which have to be matched. The matching problem is shown in Figure 2-10 (a). The variables are listed on the left and the constraints are listed on the right. The lines indicate which variable is involved in which constraint. This

representation is called a bipartite (two parts) graph. We have to now decide which variable is going to be solved for from which constraint. In the example, if we decided to solve for x from $F1$ (matched the two) then, there is nothing to match y to. Figure 2-10 (b) shows a better matching, the matchings are shown in darker lines. The matching shows that y will be solved from $F1$ and x will be solved from $F2$. Matching tells us what would be calculated from where. It does not tell us the order of the calculation. That is determined in the second stage of the algorithm.

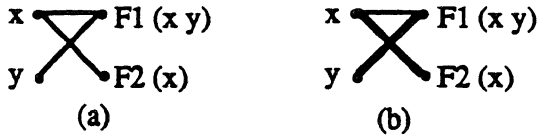

Figure 2-10: The importance of finding a Maximal Match

A matching should be maximal, that is, the maximum number of possible matchings should be found. This is achieved using a standard bipartite matching algorithm (Aho, Hopcroft & Ullman 84). The algorithm proceeds in stages. The algorithm starts by picking a matching at random. This matching is then improved by adding new matches between variables and nodes. The technique, known as "augmenting paths" is iteratively applied untill no new matches can be found. The algorithm is described in [Aho, Hopcroft, & Ullman '83].

Let us return to the clutch example to find a maximal matching. The constraints are as follows:

$$F1(T_d, F_a, D_e)$$
$$F2(D_{out}, D_{in}, D_e)$$
$$F3(D_{out}, D_{in}, F_a)$$
$$F4(T_{nominal}, T_{design})$$
$$F5(T_{nominal})$$
$$F6(D_{out}, D_{in})$$
$$F7(D_{out}, S_{hoop})$$

Figure 2-11 (a) shows a bipartite graph representation of the problem. The maximal matching (which need not be unique) is shown in Figure 2-11 (b).

Finding a maximal match tells us which variable is determined from which constraint, but it does not tell us in what order to solve the constraints. The next step of the algorithm is *ordering*. The order in which we solve for variables is based on the variable-constraint matching. For the constraints $F1(x, y)$ and $F2(x)$, x is first determined from the constraint it is matched to, $F2$. The value is then substituted in $F1$ and y is determined. The order is determined by the fact that y depended on knowing x. These dependencies can be represented as a directed graph among variables. For example, y depends on x, and x does not depend on anything.

The directed graph (digraph) is prepared after matching. In the case of the clutch, when D_e is matched to $F2(D_{out}, D_{in}, D_e)$ then it follows that D_e can be calculated only after D_{out} and D_{in} are known. In other words, Re depends on D_{out} and D_{in}. The matching shown in Figure 2-11 (b) can be converted into a digraph as shown in Figure 2-12, where, an arrow indicates that the tail depends on the head.

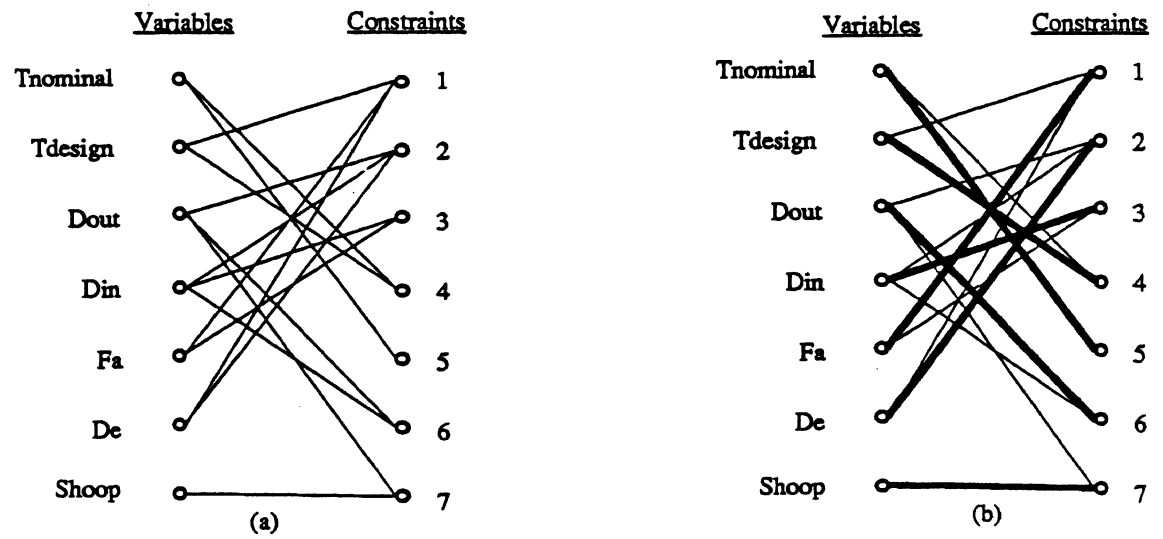

Figure 2-11: Bipartite Matching

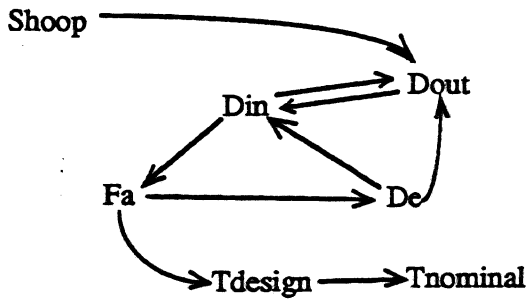

Figure 2-12: Dependency graph among variables

In the digraph, $T_{nominal}$ does not depend on any unknown quantity and hence can be immediately determined from the equation it was matched to. Once $T_{nominal}$ is known, T_{design} can be calculated. The rest of the variables, however, cannot be chained as the first two. The variables D_{out}, D_{in}, D_e and F_e are in a cycle of dependencies. That is they depend on one another and have to be solved simultaneously. The variable S_{hoop} is not in the cycle. Cycles are identified using a standard graph theoretic algorithm called the Strong Component Algorithm.

A strong component of a digraph is a maximal set of nodes in which there is a path from any node (variable) in the set to any other node in the set. A depth-first search based technique is used to determine strong components efficiently [Aho, Hopcroft & Ullman '84]. The strong component algorithm consists of the following steps:

---

1. Perform a depth-first search of the digraph (G) starting at any node N. Make a note of all the nodes visited in the list L1. The depth first search procedure is as shown below:

   DFS(G, Current-Node)
   1. Add Current-Node to globally defined list: VISITED
   2. Get the dependents (D) of the Current-Node which are not in VISITED
   3. IF there are no such dependents return NULL
      ELSE each dependent (d) do DFS(G, d)

2. Construct a new directed graph G_r by reversing the direction of every arc in G.
3. Perform a depth-first search on G_r starting the search node N. Make a note of all the nodes visited in the list L2
4. The intersection of L1 and L2 will be a cycle. Collapse the cycle into one big node, call the graph G'.
5. Repeat the above procedure on G' untill no cycles are found when performing a depth-first search from any node.

The digraph for the clutch problem is shown in Figure 2-12. Step 1 of the algorithm involves a depth first search on the graph. Let the root node be D_{out}. Let the depth first search path be: (D_{out}, D_{in}, F_a, T_{design}, $T_{nominal}$). In Step 2 the graph is inverted and searched again in Step 3 from the same start node. This time, let the search tree be (D_{out}, D_{in}, D_e, F_a). The nodes common to the two trees are (D_{out}, D_{in}, F_e). These three nodes may be collapsed as shown in Figure 2-13. Reapplying the algorithm, D_e is also collapsed into the strong component. Further applications of the procedure stops when there are no nodes in the digraph from which components can be found.

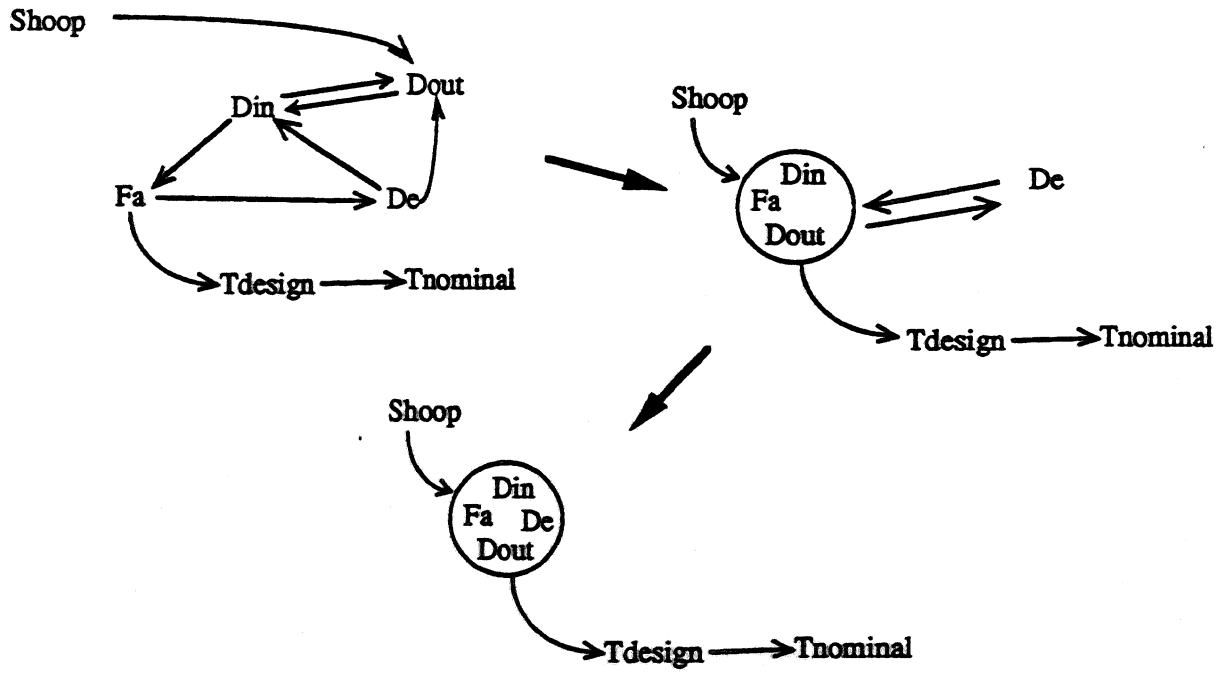

Figure 2-13: Finding strong components in a bi-partite graph.

Note, in the example above, we used a single path for the depth first search. In practice, the search tree would contain all the possible paths emanating from the start node. The nodes which can be visited from D_{out} include all the nodes in the graph except S_{hoop}. Using the entire tree helps in finding components faster. In the clutch example, if the complete tree emanating from D_{out} were used, the the strong component would have been found in just one iteration of the algorithm. We deliberately did not do this to illustrate how nodes can be iteratively collapsed to find components.

2.19.2 The Complete Planning Algorithm

In summary, the steps are as follows.

Step 1. As the evaluation of a constraint yields the value of only one variable at a time, we have to first decide which variables will be calculated from which constraint. As we would like to evaluate as many variables as possible, the matching of variables to constraints is done using a bi-partite graph matching technique.

Step 2. A directed graph of dependencies among variables is generated. For example, if one is going to calculate for variable a from the constraint $f(a, b, c)$, then a is said to depend on b and c.

Step 3. Cycles in the above di-graph indicate simultaneity among variables. Using an algorithm to find strongly components in the di-graph, the smallest cycles are found and isolated.

Step 4. After all cycles are isolated, the rest of the di-graph becomes a tree. A reverse topological sort yields the steps which can be taken to find the values of the variables. The algorithm (RTS) is as follows:

RTS(Tree)

Step A. Initialize a stack called ORDER

Step B. If there are no nodes in the Tree, return ORDER

Step C. Find all nodes that have no children (depend on no other variable)
If there are no such nodes, then the input graph is not a tree.

Step D. For each node found in Step C , do the following:
i. push the node onto the stack ORDER
ii. remove the node from the digraph

Step E. Go to Step B.

The algorithm is based on three standard graph theoretic algorithms: Bipartite-matching, Strong Components and Reverse Topological Sort. These algorithms are all described in introductory graph theory texts. Please see [Aho, Hopcroft & Ullman '83].

2.20 Breaking the Strong Components

Strong components can sometimes be broken or simplified by picking the value of one of the variables in the strong component. The process is analogous to untying knots in a string. Untying a large knot might either reveal smaller knots or might eliminate the knot altogether. The idea behind breaking a strong component is to perform a single-degree-of-freedom search on one variable instead of solving all the variables simultaneously. Consider, for example, a coupled constraint set with n variables and n constraints. Assume that all simultenaety is eliminated if one variable x is guessed. After guessing x the values of all the remaining $n-1$ unknowns can be easily determined from $n-1$ constraints. The remaining constraint can be used to calculate a new value for x. The new value is compared to the guessed value. If there is some error, a new value for x is guessed and the process is repeated. Iterations are carried out untill the error is within acceptable limits.

In the the clutch example, the strong component can be broken by picking either D_{out} or D_{in}. The effect of picking a variable can be shown in an adjacency matrix. Figure 2-14 (a) shows the strong component we just found. If we pick the value of D_{out}, then the right-most column can eliminated, and the rest of the constraints become serially decomposed (Figure 2-14 (b)). D_{in}, F_a and D_e are solved for serially from equations (6), (3) and (1) respectively. The last constraint (Equation (2)) is redundant and is used to calculate a new value for D_{out}. If the error (difference between the new and the old values of the outer diameter) is not acceptable, D_{out} is re-guessed and the solution process is repeated. Iterations are carried out untill a suitable solution is found.

In the above example, we saw how it is possible to simplify a constraint set by picking the value of one variable. We went from a problem of simultaneously solving four equations to performing several iterations on a problem with a closed form solution. It is our hypothesis that this general idea is extensible to much larger problems. We present algorithms which help identify the best variables to pick in order to simplify a given constraint problem. We also present experiments which have shown that in many cases it is possible to completely eliminate simultenaety by picking the value of just one variable. In fact, if one picks the value of either D_{out} or D_{in}, simultenaety is eliminated.

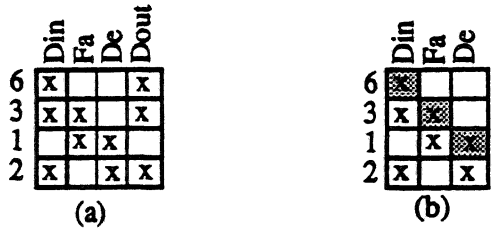

Figure 2-14: Picking the Right Variable: D_{out} is guessed

Not all variables in a strong component break the component when picked. For example, if one picks either F_a or D_e, then the component is only reduced to a smaller one. We have identified two heuristics to pick promising variables.

Innermost-Loop Heuristic

In Figure 2-13, we saw how a strong component was found in stages: by finding and collapsing cycles of nodes and super-nodes into bigger super-nodes. According to the figure, the cycle containing D_{out}, D_{in} and F_a was found first, and was later incorporated in the final strong component. Intuitively speaking, it appears that the first three variables form the "inner" component of the larger strong component. This heuristic is based on the hunch, that these "inner" variables might be the best variables to pick.

We found this heuristic to be rather weak. Inner loops are not unique, and even though the inner loops often tend to contain the best variable, they also contain non-optimal variables. This leads to a lot of wasted effort. Another reason for the inefficacy of this heuristic is that inner loops are oftentimes inside other loops only because of the order in which the nodes where considered. Change the order, and the order of collapsing of nodes into components will change. For example, in collapsing the digraph in Figure 2-12, one could have found a different cycle first. This depends on where the search tree is started, for example one could have found the cycle: F_a, D_e and D_{in}. This forms the innermost component. D_{out}

is added in the the second stage. The heuristic will wrongly consider F_a and D_e as possible candidates.

Most-Dependents Heuristic

We have found that the best variables to pick are often the ones which are most "coupled." Simply put, we sort the nodes (variables) in the strong component by the total number of dependent variables in the strong component. In the clutch digraph, applying this heuristic places D_{in} and D_{out} as the best choices (Figure 2-15). The numbers in the figure indicate the number of dependents at each node.

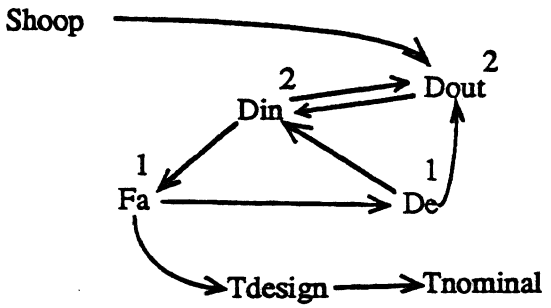

Figure 2-15: Counting the number of dependents at each node.

The number of dependents of a variables is a heuristic measure of how critical it is to know the value of a variable, before other parts of the design can be determined. It is this property that, we believe, makes the heuristic work. We have conducted experiments to verify this hypothesis.

2.20.1 Experiments with the Most-Dependent Heuristic

We are currently conductive extensive experiments to assess the efficacy of heuristic approaches to selecting variables which can break a given strong component. The experiments are being run on hundereds of randomly generated constraint sets.

Preliminary results show, that if one uses the Most-Dependents heuristic it takes (on the average) two tries to find a variable which breaks the strong component. If no heuristic is used it takes about five or six tries before the appropriate variable is found. These experiments yielded results only for small components. Larger components were rarely eliminated by choosing a single variable. Detailed experimental results will be reported in a later version of this document.

2.21 Handling Uni-Directional Constraints

One of the assumptions made in the above ordering algorithm is that all constraints are invertible. That is, for any function $F(X)$, one can find the value of any variable x_i in X if the values of all the other variables are known. Not all constraints are explicit and not all constraints are invertible. For example, a Finite Element Method (FEM) based tool takes some inputs and produces outputs. One cannot determine the inputs from the output: The constraint is a Black-Box. Algebraic constraints can also be implicit. For example, it may not be possible to calculate for all the variables in a very complex transcendental function. For such constraints only a subset of the involved variables can be solved for, thereby making the rest of the variables serve merely as inputs.

Generalizing this idea, for an irreversible constraint, there may be many possible input/output cases. For example, assume there is a constraint (let us call it *IRR*1) with the following variables: (*Power*, T_{design}, D_{out}, *MaxStress*). Where, *MaxStress* is the maximum allowable S_{hoop}. Assume that the constraint is irreversible and can be executed only in the following ways:

Case1: If the *Power* and T_{design} are known, then the constraint can give D_{out} and *MaxStress*.

Case2: If the T_{design} and D_{out} are known, then the constraint can give *MaxStress*.

If this constraint is introduced, the variables in the clutch example can be solved without iteration. It should be noted, however, that the problem is now over constrained. Unless, by chance, some of the constraints turn out to be redundant, over constrained problems are difficult to solve and require special treatment. Our purpose for introducing two new constraints is only to illustrate how explicit and implicit constraints can be ordered.

The algorithm used to order mixed implicit and explicit constraint problems is an extension of the basic explicit constraint ordering algorithm. As soon as uni-directional constraints are introduced, the bi-partite graph becomes a partially directed bi-partite graph. Algorithms for such graphs are rare and inefficient. Our solution consists of using two graphs, one for inputs and the other for outputs. The basic algorithm is modified to use the second graph when it matches variables to constraints and to use the first graph when it needs dependency information.

2.21.1 Intuitive Explanation

As before, the constraints can be represented as a bipartite graph. This time, however, the graph has to be directed in order to indicate the direction in which variables may be solved. Continuing the clutch example, all of the bi-directional constraints and one uni-directional constraint (*IRR*1) can be represented as shown in Figure 2-16. All lines with no arrows are bi-directional. The two cases of the constraint *IRR*1 are treated as separate constraints. The arrows indicate which variables are inputs and which ones are outputs.

The standard matching algorithms do not apply to directed bipartite graphs. We solve the problem using a simple trick: as the matching step of the algorithm is supposed to pair variables to constraints, with the idea that the variable can be calculated from the constraint, we should only consider those arcs which are either bidirectional or point to output variables. Matching is carried out using the original algorithm, however, the graph input to the matching algorithm is modified to depict which variables can match with constraint. This is done by removing all uni-directional arcs which point from variables to constraints. These arcs are removed to ensure that we don't match a variable to a constraint when it cannot be solved for from that constraint. In the clutch example, the graph in Figure 2-16 is converted into the bipartite graph shown in Figure 2-17 (a). Matching is carried out as usual. The maximal match is shown in part (b) of the Figure.

The next step is the generation of the dependency digraph. For reversible constraints, when a variable was matched to a constraint, then it was said to be dependent on all the other variables in the constraint. In the

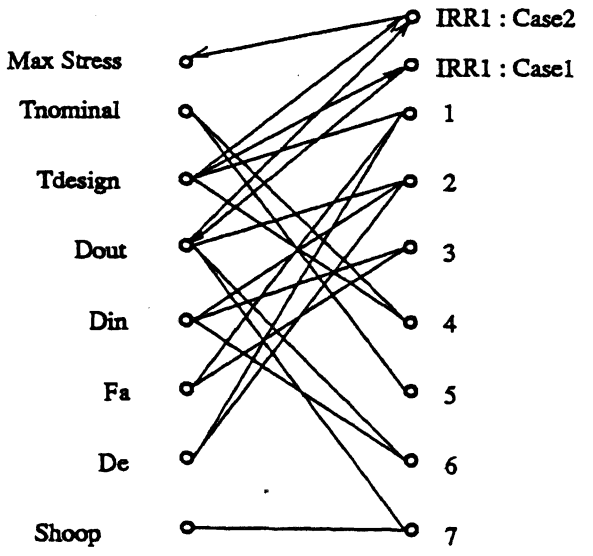

Figure 2-16: A Directed Bipartite Graph representation

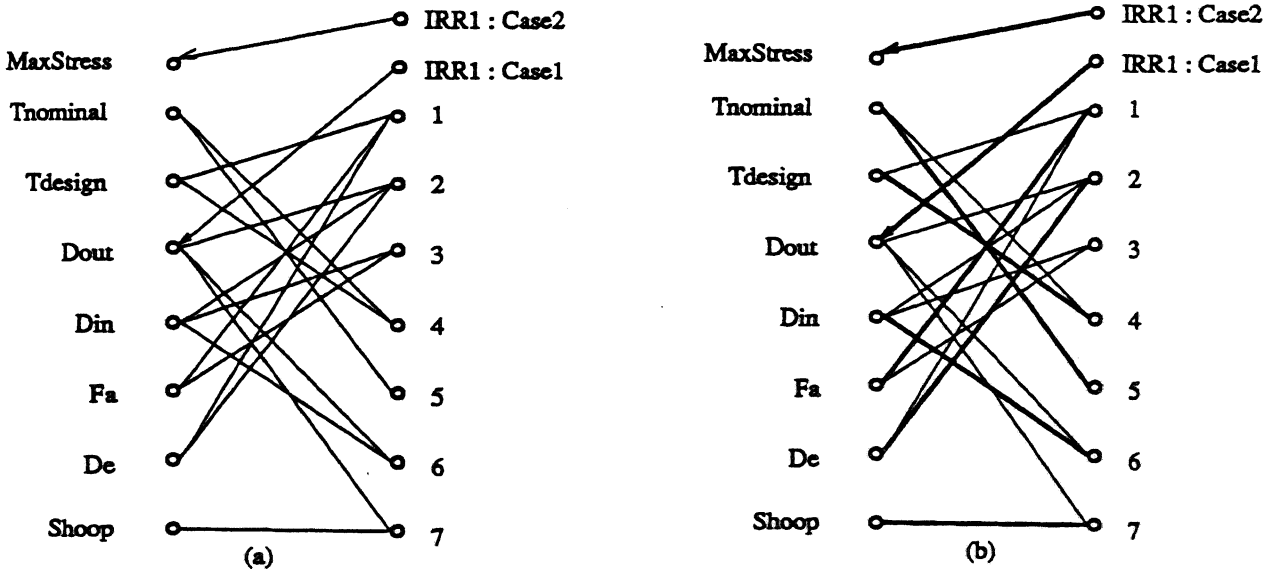

Figure 2-17: A Directed Bipartite Graph representation

case of an irreversible constraint, the matched variable is only dependent on the inputs to the constraint. In other words, one need consider only those uni-directional arcs which point from variables to constraints. In the clutch example, the matchings are as follows:

MaxStress	$IRR1 : Case2\,(T_{design}, D_{out})$
$T_{nominal}$	$F5(T_{nominal})$
T_{design}	$F4(T_{nominal}, T_{design})$
D_{out}	$IRR1 \cdot Case1\,(T_{design})$
D_{in}	$F6(D_{out}, D_{in},)$
F_a	$F1(Td, F_a, D_e)$
D_e	$F2(D_{out}, D_{in}, D_e)$
S_{hoop}	$F7(D_{out}, D_{in}, F_a, S_{hoop})$

Following the original algorithm, the matchings then have to be then converted into a directed graph. For example, *MaxStress* depends on T_{design} and D_{out}. Note that the variables listed (in the matchings above) for *IRR*1 only include the input variables. This the crux of the algorithm: We use the directed bipartite graph in two different ways. During matching, only arrows which go from right to left (Figure 2-16) are considered. This is done only for uni-directional constraints. When the dependency graph (the digraph) is prepared, we consider only those arrows which go from left to right. The digraph for the above set of matchings is shown in Figure 2-18. The numbers on the figure indicate the order in which the variable may be solved. There are no strong components in the graph.

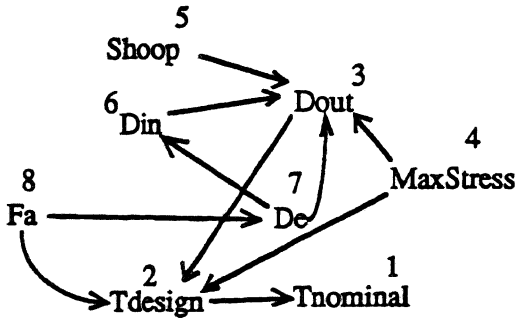

Figure 2-18: Dependency Graph with Unidirectional Constraints

2.21.2 Ordering Algorithm for a Mixed, Explicit and Implicit constraint Sets

The combined algorithm is as shown below. The actual graph theoretic algorithms being used need no modification. We just change the inputs to these algorithms.

Step 1. Split all multi-cased implicit constraints into separate constraints and develop a directed bipartite graph (Call it *B*).

Step 2. Develop a new graph (B_{match}) by removing all uni-directional arcs pointing from variables to constraints.

Step 3. Develop a new graph ($B_{dependents}$) by removing all uni-directional arcs pointing from constraints to variables .

Step 4. Find a maximal match on B_{match}.

Step 5. Develop a directed graph of dependencies using the matching found in Step 4, but using $B_{dependents}$ to find dependents.

Step 6. Find Strong components in the above digraph as usual.

Step 4. After all strong components are found, the digraph becomes a tree. A reverse topological sort yields the steps which can be taken to find the values of the variables.

2.22 Related Work

The notion of using bipartite matching and the strong components algorithm together was originally suggested by Wang (Wang 73). The algorithms were originally used to solve Gaussian matrices for solving sets of equations using Newton-Raphson like methods. Serrano applied a similar algorithm for finding strong components in sets of constraints (Serrano 87). The aim of this work was to concentrate solution on components and to avoid having to solve the entire constraint set simultaneously. Both these efforts are aimed at bi-directional constraints. We have extended the algorithms to uni-directional constraints. We have also developed the notion of breaking strong components using heuristic approaches.

Recently, Eppinger & Whitney have described a coordination problem in complex design projects [Eppinger & Whitney '89]. A design project is viewed as being composed of several tasks, each of which needs some input data and produces (as output) some data for other tasks. The dependencies among the tasks can be expressed in an adjacency matrix. The paper presents a heuristic approach to ordering the tasks. A comparison study of our approach to ordering uni-directional constraints and the proposed heuristic approach is in order.

References

[Borning 79] Borning, A.
ThingLab- A Constraint Oriented Simulation Laboratory.
Technical Report, Xerox Palo Alto Research Center, 1979.

[Gagne 53] Gagne Jr., A.F.
Torque Capacity and Design of Cone and Disk Clutches .
Product Engineering , December, 1953.

[Gosling 83] Gosling, J.
Algebraic Constraints.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1983.

[Hindhede, Et.al. 83]
Hindhede, U., J.R. Zimmerman, R.B. Hopkins, R.J. Erisman, W. C. Hull, J.D. Lang.
Machine Design Fundamentals: A practical approach.
John Wiley & Sons, 1983.

[Mackworth 77] Mackworth, A. K.
Consistency in Network Relations.
Artificial Intelligence 8:99-118, 1977.

[Moore 66] Ramon Moore.
Interval Analysis.
Prentice-Hall Inc., Englewood Cliffs,NJ, 1966.

[Moore 79] Ramon Moore.
Methods and Applications of Interval Analysis.
SIAM, Philadelphia, 1979.

[Papalambros and Wilde 88]
Panos.J.Papalmbros and D.J.Wilde.
Principles of Optimal Design.
Cambridge University Press, NewYork, 1988.

[Popplestone 80] Popplestone, R. J., Ambler, A. P. and Bellos, I.
An Interpreter for a Language for Describing Assemblies.
Artificial Intelligence 14(1):79-107, 1980.

[Ragsdell and Philips 76]
Ragsdell K.M and D.T. Philips.
Optimal design of a class of welded structures using Geometric Programming.
Transactions of ASME J. of Engin. for Industry. 98(3):1021-25, 1976.

[Ratschek and Rokne 88]
Ratschek H. and Rokne J.
New Computer Methods for Global Optimization.
Ellis Horwood Limited, Chichester, England, 1988.

[Serrano 87] Serrano, D.
Constraint Management in Conceptual Design.
PhD thesis, Dept. of Mechanical Engineering, M.I.T. , 1987.

[Sussman 80] Sussman, G. J. and Steele Jr, G. L.
CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions.
Artificial Intelligence 14:1-39, 1980.

[Sutherland 83] Sutherland, I.E.
Sketchpad - A Man-Machine Graphical Communication System.
Technical Report TechReport #296, MIT Lincoln Lab. Cambridge, Massachusetts, 1983.

[Ward 89] Ward, A.C.
A Theory of Quantitative Inference Applied to a Mechanical Design Compiler.
PhD thesis, M.I.T., 1989.

[Navinchandra & Rinderle '89]
Navinchandra D., J. Rinderle, Interval approaches for Concurrent Evaluation of Design Constraints, In proceedings of the Symposium on Concurrent Product and Process Design, held at the American Society of Mechanical Engineers Winter Annual Meeting, San Francisco, December, 1989

[Eppinger & Whitney '89]
Eppinger, S.D. , D. Whitney, Coordinating Tasks in Complex design projects, In Proceedings of the MIT-JSME (Japan Society of Mechanical Engineers) joint workshop on Concurrent Engineering. Boston, Nov, 1989.

[Aho, Hopcroft & Ullman '83]
Aho, A.V., J.E. Hopcroft, J.D. Ullma, Data structures and Algorithms, Addison-Wesley series in computer science and information processing. Addison-Wesley, Reading, MA 1983

[Serrano 87]
Serrano, D., Constraint Management in Conceptual Design, PhD dissertation, Dept. of Mechanical Engineering, MIT, 1987

[Sriram et.al '89]
Sriram, D., G. Stephanopoulos, R.D. Logcher, D. Gossard, N. Groleacu, D. Serrano, D. Navinchandra, "Knowledge-Based System Applications in Engineering Design: Research at MIT", AI Magazine, Fall 1989

[Wang 73]
Wang, R.T.R., Bandwidth Minimization, Reducibility Decomposition, and Triangularization of Sparse Matrices, PhD dissertation, Computer and Info. Science Research Center, Ohio State University, 1973

APPENDIX A: Implementation Details

The solution planning code is all written in generic Common Lisp. To avoid clashes, the code resides in a package called loop.

There is a more embellished version of the program which produces graphics. To run these capabilities, one would need the Knowledge Craft package and PostScript display capabilities.

Getting the code

The files reside on the machine gold.kalendar.ri.cmu.edu under the directory /usr/dchandra/desfus/order . The files may retrieved using the ftp package. The files to be retrieved are:

loop.lisp
imsl.lisp
lisp-utils.lisp
match.lisp
components.lisp
unido.lisp

All the files load into the ***loop*** package and will have to be run from that package.

There are some example files in the same directory. The clutch equations are in clutch.eqns. The other constraint files are:

clutch.eqns	The clutch design equations
motor.eqns	Electrical characteristics of a D.C. motor
aero.eqns	Aerodynamic equations for the design of a fan blade
turb.eqns	Turbine blade design equations

Compiling and loading

The files should be compiled while in the loop package and after the file **loop.lisp** has been loaded. The following procedure may be followed

```
CLisp> (compile-file "loop.lisp")      ; compile the loop macro

CLisp> (load "loop.lisp")              ; load the loop macro

CLisp> (in-package 'loop)             ; change the package

CLisp> (compile-file "imsl.lisp")      ; start compiling

; compile all the files
CLisp> (compile-file "lisp-utils.lisp") .....
```

After compilation, all the files may be loaded as usual. One may choose to either stay in the ***loop*** package or move to some other package.

Examples

Example 1

To order the equations:

$x = y + y^2$
$y = xy$
$b = c^3$

$$a = \frac{10b}{p}$$
$$p = 3r$$
$$z = b - 2a$$
$$p = 5$$

The run is as follows:

```
CLisp> (load "loop") ... ; load all the files

CLisp> (setq equations
             '((X = Y + Z ** 2) (Y = X * Z) (B = C ** 3)
               (A = (B * 10) / P)
               (Z = B - 2 * A) (P = 5) (P = 3 * R)))

CLisp> (loop::order eqns2 :verbose t) ;verbose switch is on

Step 1: Solve for P from constraint:
       (P = 5)
Step 2: Solve for R from constraint:
       (P = 3 * R)

Under Constrained by 1 degrees of freedom ; some stats
Collapsing ((X Y))                        ; trace information

Step 1: Solve for A from constraint
 NIL
Step 2: Solve for B from constraint
 (A = B * 10 / P)
Step 3: Solve for Z from constraint
 (Z = B - 2 * A)
Step 4: Solve the following variables simultaneously:
        (X Y)
        from the constraints:
        ((Y = X * Z) (X = Y + Z ** 2))
Step 5: Solve for C from constraint
 (B = C ** 3)

(((P (P = 5)) (R (P = 3 * R))) (A B Z (X Y) C)) ;returned list
```

The function used is ***order*** which is called from the ***loop*** package. The ordering is shown in two parts. The first part indicates the part that is directly decomposed. The second part is where components are found. In this case, the problem is under constrained. This means that there is an extra variables during bipartite matching. The system decides that variable *A* be determined from constraint *NIL*. This means that the value of *A* has to be guessed, as it is an extra degree of freedom. The rest of the setps are self explanatory.

The function returns the results as a list. The list also two parts. The first part is a list of lists. Each sublist contains two elements. The first is the name of the variable and the second is the equation from which it should be calculated. The second list in the result corresponds to the second part of the output. The list shows the order in which the variables have to be solved in. Variables in parenthesis have to be solved simultaneously. In the ouput above, we can solve for *A*, *B* and *Z*, solve *X* and *Y* simultaneously and finally solve for *C*.

Example 2

Let us now introduce some uni-directional constraints. Each case of the unidirectional constraint (unid) is expressed separately. For example, if a unid takes *A* and *B* as input and produces *X* as output, then the unid is represented as: (unid-name (A B) (X)) . If we guess A and introduce the above unid, then the equations become serially decomposed. The input would look like this:

```
Clisp> (loop::order equations :unids '( (unid1 (A B) (X))
                                        (Guess NIL (A))))

Equations serially decomposable

(((P (P = 5)) (R (P = 3 * R)) (A Guess) (B (A = B * 10 / P))
  (Z (Z = B - 2 * A)) (C (B = C ** 3))
  (X unid1) (Y (Y = X * Z))
  (Y (X = Y + Z ** 2)))
 NIL)
```

The second element of the returned list is empty because none of the equations have to be solved simultaneously. The first part of the list, chronologically lists the variables and the equations they have to be solved from.

Usage: Calling the Order Function

order *equations* &key :verbose :unids [*Function*]

The *equations* can be in either infix or postfix format. The only real requirement is to keep symbols separate. The parser does not tokenize the input. For example (A = B * 3) should not be written as (A= B * 3) or as (A = 3B).

The *unids* are expressed in the form: (*Unid-name input-list output-list*)

Each case of a unid should be listed separately, and under a unique name. If the user would like to pick a value, then a unid may be used to express this. For example, if one wanted to pre-pick the value of variable *A* then the unid: (Pre-picking-A nil (A)) will do the job. This has to be done manually.

:verbose
When set to *t*, the program will pretty print out the order in a stepped form. The default is *nil*

:unids
A list of unids. The default is *nil*