To appear in: Concurrent Engineering
Edited by P. Gu and A. Kusiak

Constraint Management in Design Fusion

D. Navin chandra[a], Mark S. Fox[b], Eric S. Gardner[a]

[a]Center for Integrated Manufacturing Decision Systems
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213
USA

[b]Department of Industrial Engineering, University of Toronto
4 Taddle Creek Road, Toronto, Ontario M5S 1A4, Canada

Abstract

We describe the Constraint Management System (CMS) of Design Fusion. Design Fusion supports the preliminary design process by providing criticism and guidance from the many perspectives that span the product life cycle. Central to Design Fusion is the representation and management of design requirements and decisions in the form of a constraint graph. CMS provides for the representation of a variety of constraint types, the propagation and checking of design decisions and the management of sets of constraints within and across design perspectives.

1. Introduction

As today's marketplace becomes more competitive, engineers are being forced to focus on the life-cycle implications of their decisions. An artifact should not only function properly, but it should also be manufacturable, reliable, easy to service, and should be recyclable. In order to design a product that satisfies these criteria, engineers have to concurrently manipulate, solve, and manage a wide variety of constraints. These constraints are in the form of goals, requirements, resource-limitations, and models that arise from a variety of sources spread across the organization. In practice, however, engineering design is often performed in a sequential manner. For example, in the design of a jet engine turbine disk, the aerodynamic shape of the blade might be determined first, and then be modified to satisfy structural constraints. It might be further modified to satisfy manufacturability and maintenance considerations. The problem with sequential design is that it shields engineers from knowing and considering other life cycle constraints. As a result, the design may require many time-consuming iterations to adequately address all the life-cycle 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 considerations such as manufacturing and maintenance are introduced only on an ad-hoc basis during preliminary design gives rise to fundamental design deficiencies. It is the aim 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.

Life-cycle concerns impose required relationships among features of the design that affect functionality, manufacturability, reliability, and serviceability. 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 can express constraints as algebraic and symbolic 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, servicing and disposal. Because there is a single, uniform representation for all constraints there is no differentiation between functional, geometric, manufacturing, and other, so called, life-cycle constraints. Methods used to refine the design by processing constraints are applied uniformly to all the constraints, regardless of their origin. The computer does not distinguish between constraints that are functional and those that have traditionally been considered downstream of the design phase. All constraints, up-stream or down-stream are considered equal. At any stage in the design process, the computer will focus its attention on the constraints that have been violated, regardless of their origins. It is for this reason that our approach achieves concurrency. For example, while a designer is working on the intricacies of the electrical connections in a large power transformer, the designer might be alerted to a problem with maximum allowable size for transportation.

This chapter describes the Constraint Management System (CMS) developed for the Design Fusion system. Design Fusion supports the preliminary design process by providing criticism and guidance from the many perspectives that span the product life cycle. We began by briefly describing Design Fusion followed by a description of the Constraint Management System's (CMS) capabilities with respect to the issues presented above. We start with a look at the constraint representation. We then describe how constraints emanating from multiple agents are managed. Followed by a description of how constraints are used to coordinate the decisions of multiple agents through conflict detection and subsequent resolution.

2. Design Fusion

In this section we will briefly examine the structure of a concurrent design system called Design Fusion. Our Constraint Management System (CMS) is part of Design Fusion. The proceeding discussion is intended to illustrate the context in which a constraint management facility can be used. The CMS is an independent software package that can be part of any concurrent, multi-agent design environment, not just Design Fusion.

The goal of Design Fusion is to create a computer-based design system that will enable a designer to concurrently consider the interactions and trade-offs among different (and even conflicting) requirements, arising from one or more life cycle perspectives. It surrounds the designer with experts and advisors that provide continuous feedback based on incremental analysis of the design as it evolves. These experts and advisors, called *perspectives*, can generate comments on the design (e.g. comments on its manufacturability), information that becomes part of the design (e.g. stresses), and portions of the geometry (e.g. the shape of an airfoil). However, the perspectives are not just a sophisticated toolbox for the designer; rather they are a group of advisors who interact with one another and with the designer.

Design Fusion is based on three underlying concepts:

- Integrating life-cycle concerns through the use of views from multiple *perspectives*, where each perspective represents a different life-cycle concern such as manufacturing, distribution, maintenance, *etc*
- Representing the design space at different levels of abstraction and granularity through the use of *features*, where features are the attributes that characterize a design from the viewpoint of a perspective
- Generating and pruning the design space through the use of *constraints*

Using the concepts of perspectives, features, and constraints, the Design Fusion system generates, prunes, and tests design alternatives. A key element of the Design Fusion architecture is the concept of *degree of fusion*, that is, the degree to which design decisions are simultaneously generated and evaluated by the interacting perspectives. In Design Fusion, all perspectives may generate and test design alternatives at all levels of abstraction and at every stage in the evolution of the design. Thus, Design Fusion is quite distinct from systems that use after-the-fact design critics to evaluate completed designs.

The design space can be viewed as a multi-dimensional space in which each dimension is a different life-cycle objective such as fabrication, testing, serviceability, reliability, *etc*. These dimensions are called *perspectives* because each dimension can be thought of as a different way of looking at the design. As

a design evolves, a designer moves from one viewpoint in the design space to another and moves from one level of abstraction to another both within a perspective and across different perspectives. By continuously viewing, commenting on, and intervening in, the evolution of a design from each of the perspectives, the constraints of the product's life-cycle will be accounted for in the completed design. Design Fusion allows implicit functional requirements such as manufacture, assembly, test, etc. to be integrated into the design at the appropriate time and at the appropriate level of detail.

The Design Fusion architecture [Fox et al. 92] is based on the blackboard model of problem solving [Erman et al. 80, Nii 86a, Nii 86b] illustrated in Figure 2-1. The architecture has four major components: the blackboard, knowledge sources, search manager, and user interface.

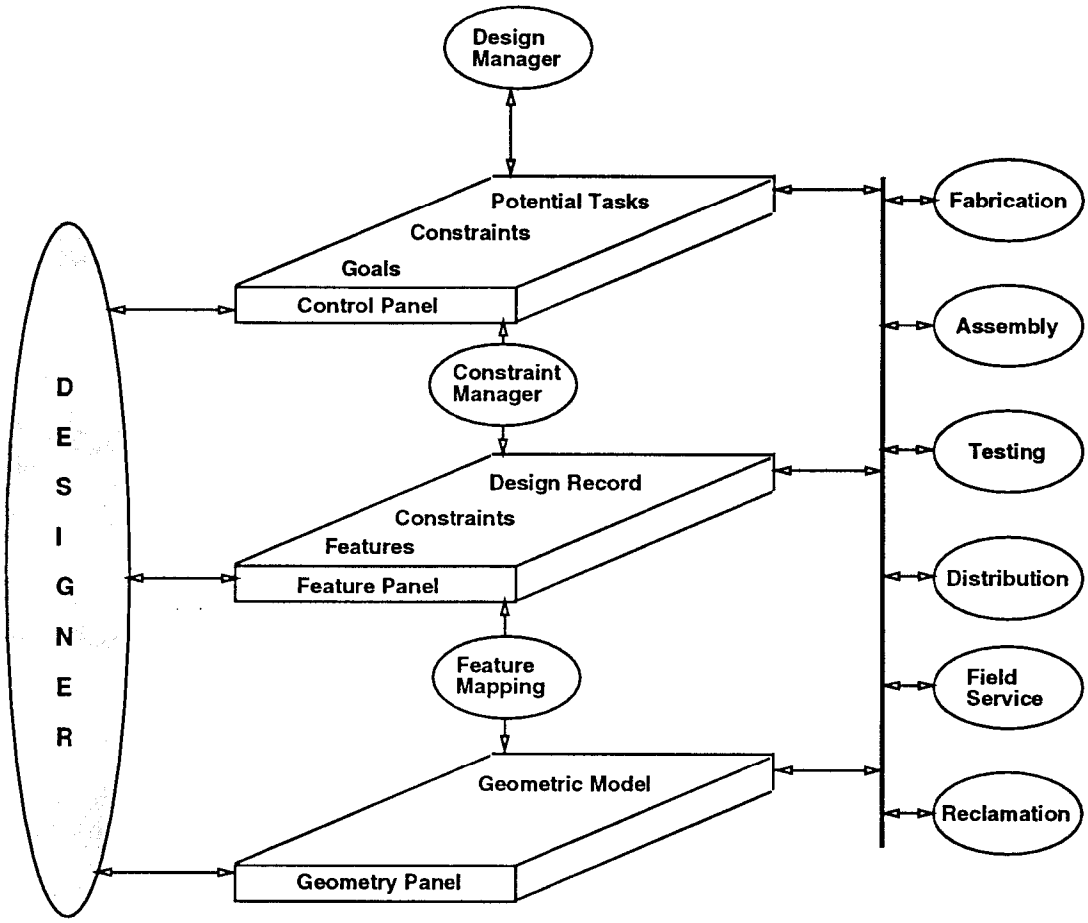

Figure 2-1: Design Fusion system architecture

The blackboard is a composed of a hierarchy of three panels. The *geometry panel* is the lowest level representation of the design. It provides a geometric-based representation using the NOODLES modeller.

The *feature panel* is a symbol level representation of the design. It provides

symbolic representations of features, constraints, specifications and the design record. Features are defined in terms of geometric features and are extracted automatically from the geometry panel. Constraints of various types link features. A design record tracks the design decisions that led to the creation of a constraint or feature. Design records are defined by the perspective which generated the decision, the type of processing that led to the decision, and the information upon which it was based. This information can be used to maintain design consistency when underlying assumptions of the design change or to track constraint violations back to the sources.

The *control panel* contains the information necessary to control the operation of the system. It uses knowledge of design goals and constraints to decide where attention should be focused and which knowledge source to apply next.

There exist two types of knowledge sources: perspectivies and methods. *Perspectives* represent knowledge of different stages in the product life cycle. Each perspective may criticize or generate design decisions. Using perspectives that communicate through a blackboard architecture enables us to partition the design knowledge. Each perspective can define its own internal set of features, constraints, and variables. Inconsistent requirements, names, and definitions are contained within the perspectives because the communication is through the shared representation. *Methods* provide standard analysis capabilities to the system. Three methods are currently being used: feature extraction, constraint management and mathematical programming.

The *search manager* provides a means for dynamically coordinating the application of knowledge sources as required by the problem and the designer. The system cycles through four stages of control: knowledge source identification, knowledge source selection, knowledge source execution, and constraint management. At the beginning of each cycle perspectives indicate their interest in contributing to the design. The search manager must decide which perspective then is to execute. Inconsistencies and conflicts in goals inevitably arise during the design process. These inconsistencies are tolerated by the system but are also tracked. The designer is notified of inconsistencies when appropriate.

The purpose fo the *user interface* is to provide the designer with a complete interactive environment for doing design. It provides the user with the ability to: define specifications and constraints, select from a library of existing designs, and modify designs. The user also has the capability to override the systems decisions at each stage in the search manager's decision cycle.

CMS is a critical component of the Design Fusion system. Design decision-making centers on features, parameters and constraints. It is CMS that maintains the constraint model, propagates decisions and detects inconsistencies as the design process proceeds. Our research has focused on several issues

relating to the use and management of constraints in a concurrent engineering design environment. The foremost issue is that of representation. A uniform representation mechanism for capturing a wide variety of constraint and parameter types is needed. The representation should allow multiple perspectives to dynamically add and retract constraints from an evolving design. While there is a need to handle all the constraints simultaneously, there is also the need to keep track of the origins of constraints, their importance, and information about who created them and for what reason. These constraints can be used for inter-perspective communication via a protocol that allows for the posting, checking, retraction and general manipulation of the constraint based information. When many agents post their constraints, it is important to keep track of relationships among constraints as they are added to the constraint pool. Perspectives should not be allowed to modify parameters owned by other perspectives, unless the appropriate permissions are provided. Mechanisms to handle inter-perspective referencing of parameters are needed. In this context, constraints can also be used to "bridge" related parameters in two different perspectives. For example, two perspectives might use different terms to mean the same thing, a constraint equating the two parameters could address this mismatch. In addition, two perspectives might use the same term to mean different things, in this case contextual information has to be maintained. For example, when aerodynamics says "length", it should not be confused with the manufacturing perspective's "length".

As several perspectives post their own constraints, conflicts may be detected. We need methods by which constraints can be checked. This checking should include chaining, that is, the implications of a design decision should be propagated to other parts of the design. This approach makes it possible to detect violations among the decisions made by the perspectives. Finally, the detected conflicts have to be resolved. We need a conflict resolution protocol that will allow the various perspectives to arrive at a settlement. In distributed problem solving situations the passing of constraints among agents can be viewed as a means of negotiation. A protocol for conflict detection, blame assignment and resolution is needed.

Although constraints are a general mechanism for representing 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 paper that constraints can be expressed as relationships among feature parameters, then we can say that a design is complete when all parameters have been assigned values and when all the constraints have been satisfied simultaneously.

3. Constraint Representation

The constraint representation has three parts: parameters, constraints, and constraint sets.

3.1. Parameters

A constraint defines a relationship among design parameters. Parameters are associated with features of the design. For example, a "hole" feature will have (at least) "radius" and "depth" parameters. Parameters are typed, with the following types available in CMS:

1. **continuous-variable.** Examples: 10.3, 4.3^3
2. **discrete.** Examples: 3, 10
3. **symbolic.** Examples: green, table
4. **logical.** == {.true. .false. .unknown. .dontcare.}
5. **qualitative-quanitity.** {0 + -}

The frame representation of a parameter is as follows:

```
{{ PARAMETER
  full-name              ; full print name
  value                  ; current value of parameter, starts
                         ; as UNKNOWN
  constrained-by         ; constraints that constrain the
                         ; parameter
  context                ; current design context
  abstraction-level      ; current level
  time-stamp
  owner                  ; perspective name
  protection             ; visibility to other perspectives
  relaxation-function    ; function which will return
                         ; relaxations
  related-feature        ; name of originating entity
  generated-by           ; How generated, by input or by a
                         ; constraint
  generation-justification ; reason for creation
 }}
```

3.2. Constraints

Parameters may be linked together by one or more constraints that define a graph. CMS supports the following constraint types:

1. **A Restriction:** A limitation on some parameter. For example: *The largest motor that can be wound on the automatic winding machine is 25 cm. dia.*

2. **Equations:** Equalities and in-equality equations are used for engineering constraints. For example: $F = m.a$, or $F <= m.a$

3. **A Requirement:** An inequality or set membership. For example: *Safety rating should be at least 2.3,* or *Color should be one of {Green Opal Blue}.*

4. **A predicate:** Simple logical relations among logical parameters can be checked. For example: *a* AND *b* OR *c* = *.true.*. Predicate constraints, however, cannot be propagated. This is done with rules.

5. **Inference Rules.** These are symbolic constraints that can be used for propagation. For example *a* AND *b* -> *c*.

6. **Qualitative-constraint.** Qualitative constraints can be imposed on qualitative quantities. For example, in a hydromechanical domain to require that some water flow (Q) does not increase with some signal (X), we write the constraint as follows: [not(Q X +) AND not(Q X -)]

7. **A Goal:** A max or min goal requirement. For example: *Maximize pressure difference across stage-one.* This has not been implemented yet.

8. **A Range:** Intervals capture the notion of ranges. For example: *Design Velocity should be between 100 to 200 m.p.h.* Ranges may be exclusive: *Vibration harmonics should be outside the interval [33, 44].*

9. **Implicit-constraint.** Black box constraints. These are usually in the form of subroutines that take certain inputs and produce outputs. In these cases, the system does not have access to the inner workings of the constraint.

10. **Compatibility Relations:** For Example: *Snap fits are not compatible with high temperature applications.* This aspect is implemented as look-up tables.

The set of constraints form a dynamic constraint graph. It is possible to post constraints to the graph from a variety of different perspectives. For example, one might load the design constraints together with the manufacturing constraints to find possible conflicts. Such conflicts may be found well before values are selected for all the variables in the design. It is for this reason that the constraint graph representation is useful in coordinating the perspectives.

The constraint graph is represented as a tri-partite graph of constraints, parameters, and features. Each constraint points to the parameters it *constrains* and each parameter points to the feature it is part of (Figure 3-1).

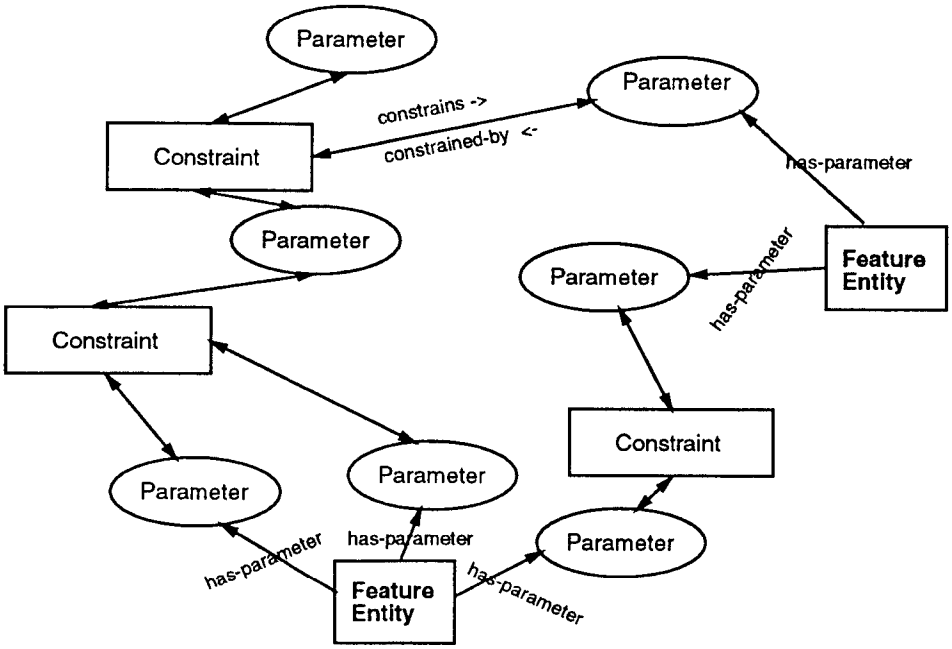

Figure 3-1: Tri-partite graph of features, variables and constraints

A constraint is represented as a frame with attributes that relate it to its surroundings in the constraint graph:

```
{{ CONSTRAINT
  form                      ; the equation/logical relation
  context                   ; the design context
  active?                   ; is it on or off
  time-stamp                ; when was it posted
  satisfaction-level        ; how violated is it (error value)
  abstraction-level         ; name of abstraction level
  in-set                    ; which constraint set is it in
  has-relaxations           ; simpler forms of the constraint
  owner                     ; name of perspective
  constrains                ; list of feature parameters that
                            ; are constrained
  protection                ; read, write protection
  exported                  ; is it visible to outsiders?
  abstraction-of            ; list of constraints that this
                            ; is an abstraction of constraint
  importance                ; on a scale of 1 to 10
  generated-by              ; who created it? Another constraint?
  generation-justification ; reason for creation
}}
```

3.3. Constraint Sets

Each perspective has its own set of features that it is interested in. Consequently, each perspective also has its own set of constraints that relate its parameters. For example, a manufacturing perspective is interested in parameters relating to tolerances, surface finishes and the process plan. These parameters are owned by the perspective. Ownership relations are handled in CMS by partitioning the constraint graph into *constraint sets* (CS). Each constraint set is `owned-by` a perspective or any other client on the network. This arrangement is indicated in Figure 3-2.

Constraint sets are collections of parameters and constraints and are created by perspectives. We also allow constraint sets to contain constraint sets. This information is held in the `parent` and `children` slots of each constraint set. A constraint set can have many parent constraint sets. The highest parent constraint set is "root" and the leaf constraint sets point to the NULL child. Each perspective owns one or more constraint sets. The parameters in a constraint set may be internal to a constraint set or may be *exported*. By exporting a parameter, a particular perspective makes the parameter "visible" to other perspectives.

```
{{ CONSTRAINT_SET
        name                    ; name of the cs
        perspective             ; name of perspective that created this cs
        owned-by                ; name of client who owns the perspective
        interested-parties      ; list of cs that refer to vars in
                                ; this cs including this cs
        parent                  ; list of the cs that contain this cs
        children                ; list of the cs that are contained in this cs
        contains_constraints    ; list of constraints contained in the cs
        owns_constraints        ; list of constraints owned by the cs
        contains_parameters     ; list of parameters contained in the cs
        owns_parameters         ; list of parameters owned by the cs
        contains_constraints    ; list of constraints contained in the cs
        owns_constraints        ; list of constraints owned by the cs
        contains_parameters     ; list of parameters contained in the cs
        owns_parameters         ; list of parameters owned by the cs
        locks_parameters        ; list of parameters locked by the cs
        imports_parameters      ; parameters imported from other cs
        public_export           ; parameters that can be referenced by any cs
        private_export          ; parameters that may be referenced by any cs
                                ; that has the same perspective as the this cs
}}
```

As the Figure 3-2 shows, there are some constraints that refer to parameters in other constraint sets. It is this capability that makes CMS's representation useful in multi-perspective coordination. Any perspective can post its own constraints that relate to parameters in other constraint sets. This is how parameters belonging to different disciplines can be made to relate to one another.

As is shown in the figure, we are required to distinguish between parameters

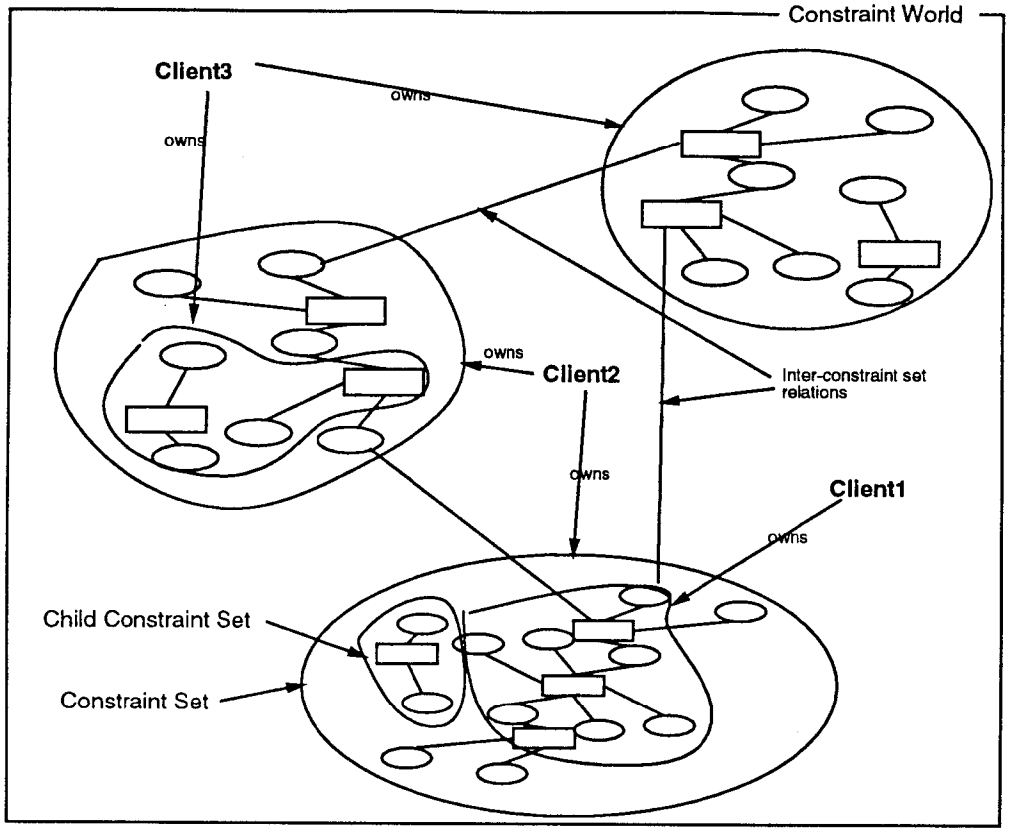

Figure 3-2: Constraint World consisting of inter-connected constraint sets

that are internally and externally referenced. Parameters that are allowed to be referenced in constraints outside the current constraint set are said to be "exported". Parameters may either be exported to other constraint sets belonging to the same perspective or to constraint sets belonging to other perspectives. To capture this information, the exported parameters are distributed in two lists: `public_export` (those exported to all constraint sets) and `private_export` (those exported to constraint sets belonging to the same perspective). Those parameters that exist in the list `owns_parameters` and not in the two exported parameter lists are the current constraint set's internal parameters.

If the parameters that a constraint references are in other constraint sets, one would have to check if the parameters have indeed been "exported" by those constraint sets. If the perspective of the two constraint sets is the same, then the parameter should at least be privately exported (present in `private_export` list). If the perspectives differ, the parameter should have been publically exported.

3.4. Constraint Set Maintenance

As the constraint graph is dynamic in nature, i.e., throughout the design process new constraints are added and modified by developers, logical consistency has to be maintained at all times. Following are consistency maintenance methods provided by CMS. The consistency algorithms were designed and implemented by Kerrin Smith [Smith 91].

Whenever one constraint set is contained by another, the values in the `contains-parameters` and `contains-constraints` slot of a child, are appended to the `contains-parameters` and `contains-constraints` slots in the parent CS. Duplicate values are not appended. The child CS is appended to the `children` slot in the parent CS. The parent CS is appended to the `parents` slot in the child CS. These actions are carried out automatically within the CMS and so the user is relieved of the burden of maintaining this consistency.

To allow for the deletion of constraint sets, each constraint set carries information about other constraint sets that refer to it. This is done with a slot in the `constraint_set` data structure called `interested_parties`. This slot carries the names of constraint sets with constraints that refer to parameters in the current constraint set. The list includes the name of the current constraint set (it is interested in itself). When a perspective attempts to delete a constraint set, it may not be possible because other constraint sets may refer to parameters inside the current constraint set. In other words, although a constraint set is not "interested" in itself, there may be other parties that are still interested in it.

Consistency of the network is maintained through a high degree of redundancy in the data structures. For example a constraint that constrains some parameters X, Y, and Z, maintains this information locally. In addition, parameters X, Y, and Z maintain the information that they are constrained by the constraint. It would be possible to determine the `contains-parameters` slot in a CS by traversing every constraint within the CS or (recursively) within a child CS. However, if this data is needed often (e.g., to determine the criticality of a particular parameter) then retrieving it by a recursive scanning of constraint sets would be time consuming. Hence, this information is maintained in two places. The tradeoff is one of data independence (or orthogonality) vs. quick and efficient traversal of the data.

While there is redundancy of information, the data itself is not redundant. The actual parameters, constraints, etc., are stored in only one place and therefore there is no risk of data inconsistency. The data redundancy is achieved by storing pointers rather than storing values or structures and therefore, we can obtain this useful redundancy with a minimum of additional memory use. The only real price one pays for this redundancy is ensuring that the constraint network remains consistent. This requires, for example, that if a constraint is deleted and with it the last occurrence of a parameter, then all pointers to the parameter must be automatically deleted.

4. Constraint Networks: Manipulation and Propagation

The CMS provides a variety of constraint manipulation and reasoning capabilities, including:

1. **Posting.** Perspectives are able to send constraints to the system's constraint graph anytime during a product's development cycle. Whenever a constraint is received, it is automatically linked into the constraint graph, and hence becomes part of the design's environment.
2. **Checking and Evaluation.** Constraints and algebraic expressions may be evaluated at any time.
3. **Consistency.** When a perspective makes a change to a feature, it may request a consistency check. This provides an assessment of whether the current state of the design satisfies all the constraints. Conflicts between different parts of the design are also shown.
4. **Propagation.** Provides information about the implications of a design decision on other parts of the design. This kind of feedback can be generated even before all the parameters are assigned values.

One of the important functions of the CMS is its ability to propagate design decisions. Whenever a parameter is assigned a value, the CMS propagates the value through the constraint network, checking constraints along the way. Constraint propagation is done by solving constraints one after the other. As there is no notion of directionality in a constraint network, a plan is needed in order to carry out such a propagation. In this section we will examine the propagation planning algorithms in the CMS. The algorithms are based on the approach originally proposed by Serrano [Serrano 87a].

4.1. Serial Decomposition

The simplest type of constraints 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. When a set of constraints is not serially decomposable, the constraints have to be solved simultaneously. This poses serious problems when the constraint set is very large (> 100 non-linear equations). We approach this problem by identifying and isolating subsets of the entire constraint set that necessarily have to be solved simultaneously.

Let us begin by examining an algorithm for planning the solution of a serially decomposable constraint set. The constraints can be ordered using a very simple row and column elimination algorithm:

Step 0. Express the constraint set as an adjacency matrix, with rows corresponding to the constraints and with the columns corresponding to variables. A "X" is placed in each column of a row to indicate which variable appears in which constraint.

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.

4.2. Simultaneous Constraints

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. The algorithm consists of two stages: *Matching* and *Component Finding* [Serrano 87a]. The first stage matches variables to constraints. 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 variables to constraints. It is important to try and find as many matchings as possible. For example, in the three constraints $F1(x,y)$, $F2(x)$ and $F3(x,y,z)$, there are three variables x, y and z which have to be matched. The matching problem is shown in Figure 4-1. 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 using which constraint. In the example, if we decided to solve for x from $F1$ (matched the two) then, we have to match y to $F3$ and there is nothing left to match z to. The Figure 4-1 shows a maximal matching shown in darker lines. The matching shows that x will be solved from $F2$, y from $F2$ and finally z from $F3$. The

matching is converted into a dependency graph (also shown in Figure). In order to calculate for z from $F3$, one needs values for x and y, hence, z is said to depend on x and y. This is indicated by the arcs in the dependency graph. In the same way, y depends on x while x does not depend on anything. A simple dependency-based sorting of the nodes in the graph yields a solution plan. For example, for the graph in the figure, we start by selecting the node that does not depend on any unknown. x is the first such node. After x is solved for, y will be the next such node, and finally z. This yields the solution order.

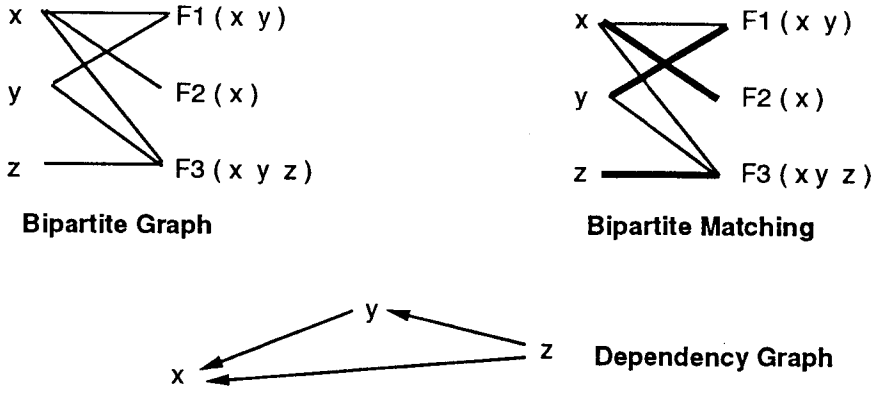

Figure 4-1: Finding a Maximal Match

Simple sorting works only when the constraints are serially decomposable. If there are cycles in the dependency graph, then some simultaneous solution is required. These cycles are identified using a standard graph theoretic algorithm called the Strong Component Algorithm. A strong component of a 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 a 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 that 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. This generates a new version of G, call the new graph G'
5. Repeat the above procedure on G' until no cycles are found when performing a depth-first search from any node

---

The above algorithm is used to develop a propagation plan as shown below[1]:

Serrano's Constraint Solution Planning Algorithm:

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 connected 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 result is returned in the list called ORDER. For example, ORDER might look like this: [x y z (p q r) a b (c d)]. The list is interpreted as follows: solve for

---

[1]The algorithm is based on three standard graph theoretic algorithms: Bipartite-matching, Strong Components and Reverse Topological Sort [Aho, Hopcroft & Ullman 83].

x, then y, then z, then solve for p, q and r simultaneously, after which it will be possible to solve for a, then b and finally one can solve for c and d simultaneously. After the solution plan is generated by the CMS it is used to calculate parameters.

4.2.1. Breaking the Strong Components

Sometimes the strong components are too large to solve simultaneously. We have developed special heuristics to break these components into smaller manageable parts [Krishnan et.al. 90]. 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 n^{th} 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 until the error is within acceptable limits.

4.3. 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. The constraint is a Black-Box; one cannot determine the inputs from the output. 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.

The algorithm used to order mixed implicit and explicit constraint problems is our extension of the basic explicit constraint ordering algorithm by Serrano. 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.

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. Start by developing the bipartite graph. As some of the constraints are uni-directional, some of the links between parameters and constraints will be directed. Call this directed bipartite graph 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.

4.4. Other Solution Ordering Research

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 [Navin Chandra & Rinderle 89a]. 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.

5. The Role of Constraints in Coordinating Perspectives

In this chapter, we have seen how constraint networks are represented and constructed in the CMS. We have also seen how constraint propagation can be used to assess the implications of design decisions. By propagating constraints

across disciplinary boundaries, the system can find downstream violations that are caused by a given design decision. These capabilities serve the basis of a constraint based coordination function.

Design Fusion supports a group-problem solving style of design; each perspective contributes concurrently to an evolving, shared vision of the artifact. Without adequate coordination, the design process can quickly become chaotic, reducing the possibility of convergence. We believe that the development of an effective theory of coordination requires an understanding of the problem solving methods of individual perspectives. In a situation where no single perspective's knowledge dominates the design process, more sophisticated coordination mechanisms are required. We need to understand how perspectives make decisions so that means for *influencing* their decisions can be devised. Different assumptions on how perspectives solve problems may lead to different styles of coordination. We refer to this as the "culture" of the organization.

The "cultural" assumption we make is that a perspective's primary form of communication is in the form of constraints. In this case, constraints are more than simple predicates, but have rich set of additional information such as importance, acceptable relaxations, utility of relaxation, situational relevance, etc. The adequacy of this perspective has been demonstrated in the domain of scheduling [Sycara et al. 91] and concurrent engineering [Navin Chandra & Rinderle 89b, Krishnan et.al. 90].

The rest of this section details the various roles constraints play in coordinating perspectives.

5.1. Communication of Intent

The intended reason for a decision can be expressed as a constraint which may be checked when the decision of one perspective impacts another. If one perspective is considering changing some decision, it can check if the original justifactions for the decision are still satisfied.

For example, the fabrication perspective might find that a bracket, of the size specified in the design, does not exit. The perspective might consider substituting it with a larger bracket from stock. If the original intent is expressed as a constraint, then the fabrication perspective could check if the new bracket violates the intent.

The reason cited for a design decision can be any one of the following (this information is stored in the `generation-justification` slot of the corresponding parameter or constraint):

1. Satisfaction of a requirement or code. For example one might choose a particular color, because it is specified by the customer.
2. Satisfaction of an engineering requirement. A value of a parameter

may be selected in order to satisfy some constraint. The `generation-justification` slot will point to a constraint. For example, a bracket may be re-sized to reduce stresses at a joint. The parameter corresponding to the bracket's size will point to a stress constraint as justification.

3. Satisfaction of a goal. Design decisions are also driven by constraints that can be minimized. This includes aspects such as cost, weight, manufacturability, and environmental compatibility.

The above representations of intent may be easily evaluated whenever a change is considered. If a perspective wants to propose changing a parameter owned by another perspective, then it can experiment with many alternatives before it actually proposes the change. By checking the justification slot, the first perspective can check if its proposed change is compatible with the original intent of the second perspective.

5.2. Constraints as Monitors

Constraints may be set up to detect discrepancies whenever they occur. For example, a transportation perspective may set up constraints that monitor size parameters. The constraint, in this context, looks for sizes that exceed the standard shipping sizes. This concept is implemented in the system through a Rete-Network. Each perspective specifies a list of parameters that it is interested in monitoring. If a change is detected on the blackboard, then associated perspective is invoked.

5.3. Coordinatinating Perspectives

Decisions made by one perspective often impact other perspectives. The sooner the implications of a decision are detected, the fewer the number of design alternatives and revisions will be explored.

Figure 5-1 shows how constraints can inter-relate different perspectives. The figure shows the following coordination functions:

1. **Bridge Constraints:** A variable may have the same name, but different meanings in different disciplines. For example, for the aerodynamics perspective, the length of the turbine blade is the length of the aerofoil only. For the manufacturing perspective, the length includes the height of the root.

2. **Cross references:** A constraint in one perspective can refer to a a parameter in another perspective. For example, the Marketing constraint set has a cost function that uses the manufacturing cost as input.

3. **Protection:** Some parameters are strictly owned by a perspective. Such parameters are locked and are not exported. Other perspectives can change these parameters only in their local

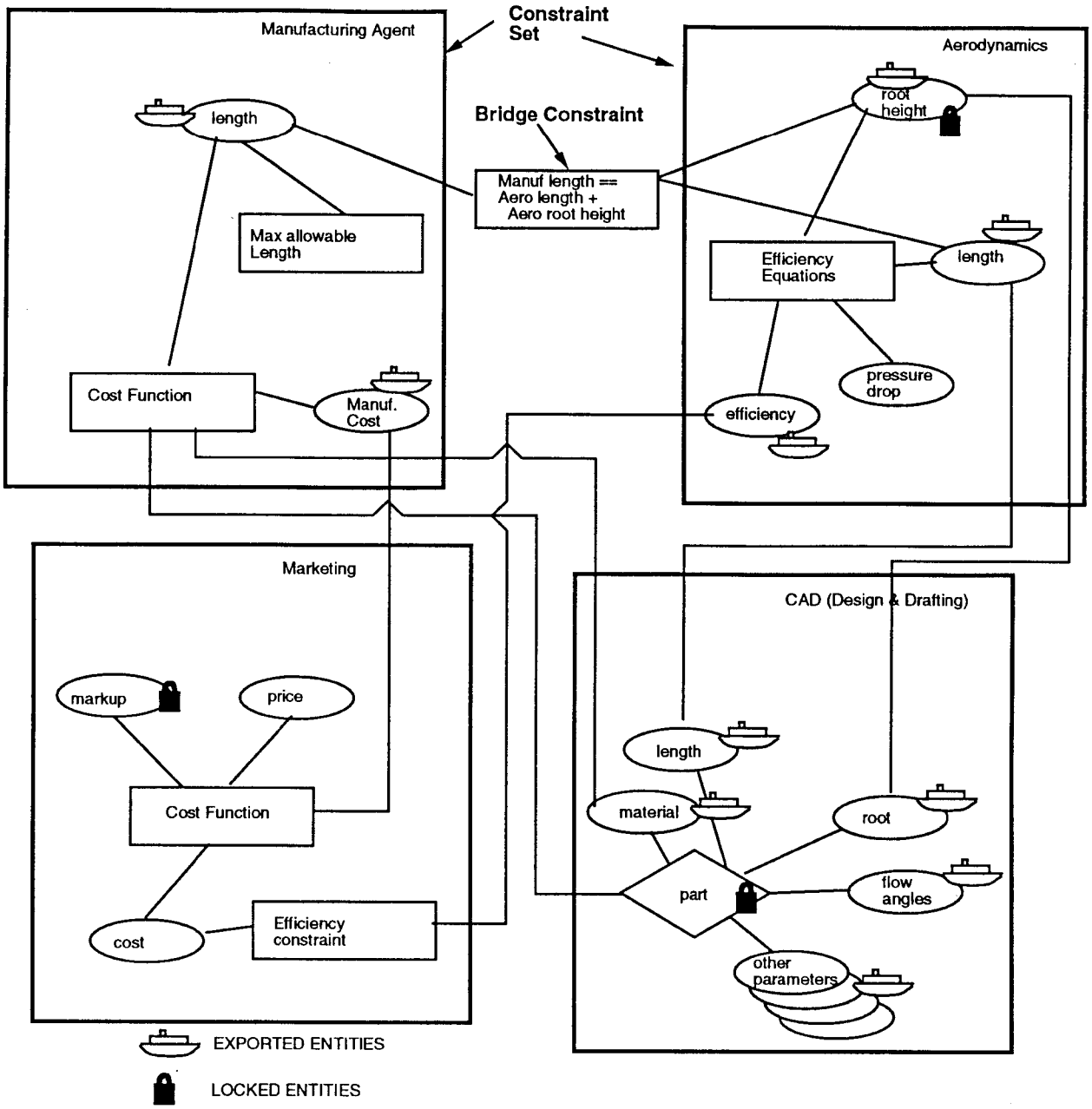

Figure 5-1: Constraints help in Coordinating disparate disciplines

workspaces while operating in a "what-if" mode. Actual changes have to be explicitly requested of the owning perspective.

5.4. Concurrency

True concurrency can be achieved when constraints are evaluated on incomplete designs. A design that is not yet complete may have some parameters that have not been assigned exact values, and there may be some uncertainty about the final design characteristics. Intervals are used to express

upper and lower bounds on parameter values, making it possible to assess some properties of the artifact before exact values are assigned. For example, in a motor design problem one might not know the exact shaft size, but might be able to estimate the general range of values based on prior experience. This information can sometimes be used to guide the designer early in the design process. This is similar to how engineers might sometimes resort to *back of the envelope* calculations, using inexact values and rough calculations.

There are several levels of in-exactness which may be used to represent a parameter value:

1. **Exact Assignment:** A numerical value. For example: X = 55.53
2. **Interval:** A range of possible values. For example: X is between 50.0 and 80.0
3. **Default Value:** One might choose some default even before starting the design. Default values are not taken literally when making calculations. For example: X = 65.0.
4. **Order of Magnitude:** For example: X = $+10^3$
5. **Sign:** Whether it is a positive or negative value. X = +ve.

At any point during a designing process one may have values for parameters at any of the above levels of specificity. The challenge is to be be able to evaluate the constraints on the design and provide the designer valuable feedback about violations and possible violations due to parameters which have yet to be fixed. The constraint management system (CMS) uses interval mathematics to propagate inexact parameter values [Navin Chandra & Rinderle 89b]. 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]. By adopting this approach we are, in essence, treating equalities as inequalities. Instead of equating a variable to a number we assign a range of values (an interval) to the variable. This generalizes the notion of equality assignment and adds flexibility to the representation of parameters, making it possible to capture incompleteness and uncertainty in a design.

6. Recovering from Conflicts

The Design Fusion architecture provides a set of protocols for the posting and refinement of decisions on the blackboard. All decisions are represented as constraints on the evolving design. During the process of design it is common for perspectives to generate conflicting decisions. Conflicts are manifested in the form of constraint violations. Conflicts can be resolved by either relaxing the constraint, replacing it, or by changing the decisions that led up to the conflict. This process may require negotiation among the interested perspectives. The following define a subset of the protocols provided in Design Fusion to manage constraint violations.

6.1. Retraction

The simplest strategy for recovering from a conflict is to retract the violated constraint. Only redundant, wrong, or unimportant constraints should be removed from consideration. In some cases a constraint may be replaced by another constraint.

6.2. Posting

When new features are added to the design, an interested perspective can post constraints that are relevant to the feature. For example, if we add a rib to a molded part, constraints relating to ribs will be posted by the manufacturing feature. These constraints are posted with pointers to the reason for which they were posted. If, at a later time, the rib feature is deleted, the underlying Truth Maintenance Facility (RMS) retracts the corresponding constraints.

6.3. Revelation

It is not possible to put all the relevant constraints about a design in one large constraint network, each perspective uses its own private workspace for storing and managing constraints. The perspective makes public, only those constraints that are relevant to the evolving design. These public constraints are often simplified, abstract forms of more complex constraints that are maintained within the perspective. When a constraint is violated, the perspective that posted the constraint might *reveal* the underlying, more complex constraint. Revelation helps identify the real causes of a problem.

Consider the following example. The manufacturing perspective posts a constraint that limits the length of a part to be fabricated by centrifugal casting. This constraint is simple to interpret. As long as the design lies within the limit, the constraint is satisfied. Assume that the structural perspective wants to make the part larger, thus violating the manufacturing perspective's constraint. On the surface, this problem appears to be a disagreement about the size of the part. If the manufacturing perspective is now asked to reveal his real reasons for posting the constraint it might help focus the conflict resolution process. In this example, lets assume that the real reason for the constraint is manufacturing cost. As the manufacturing facility cannot handle large centrifugal castings, large castings would have to be contracted out. After the constraint is revealed, it becomes clear that there are three possible ways to solve the problem: (a) increase the cost by going to a subcontractor, or (b) change the manufacturing process so that larger parts can be handled in-house, (c) reduce the size of the part.

6.4. Relaxation

Relaxation is the process of substituting a weaker constraint (B) for one that is in conflict (A). Weak is defined as "if A is found to be true, then so will B, but not vice versa". For example, a tolerance constraint may be weakened to accept a greater tolerance. In order to perform such actions, a perspective requires knowledge of how to relax a constraint. Such expertise is embedded within each perspective.

In addition to relaxing numerical constraints we need techniques for relaxing non-numeric relationships. For example, a constraint on colors may be relaxed by attaching utilities to the various color combinations. Having done this, the constraint can be relaxed by just lowering the expected utility [Fox 87, Navin Chandra 91a].

The relaxation command alters the constraint network. For example, the "relax" command will replace a constraint by its relaxation (stored in a slot of the constraint's data structure). If the user wants a special relaxation, then the form of the relaxation is provided. The same idea applies to revelation. If a constraint is explicitly stored as being an abstraction-of some other constraint, then the abstracted constraint is replaced by the constraint is is an abstraction of. In some cases, a constraint is the abstraction of many constraints, in this situation all the abstracted constraints are added to the constraint network.

6.5. Postponement

If a constraint is not very important, but cannot be retracted, it could be disregarded till later. If a perspective postpones a constraint, then the constraints active slot is set to off. The truth maintenance system then identifies decisions based on this constraint and has them retracted.

6.6. Ignore

When an important constraint is violated, it may sometimes be ignored till later. This happens when one is confident of being able to satisfy the constraint at a later time. This kind of behavior has been observed in a protocol study [Navin Chandra 91b].

7. Relationship to Other Research

Research in the general area of "constraints" has been carried out since the early work of Sutherland [Sutherland 63]. The basic idea is to treat constraints, not as procedures, but as relationships among features of a domain. This extension of the classical operations research approach to constrained problem solving has been researched in Artificial Intelligence, and more recently in Design Theory.

Much of the early work [Sussman 80, Borning 79] has concentrated on using

constraints to control a problem solving process, by tracking dependencies and by reducing backtracking [Stefik 80, Fox 83]. These ideas led up the a formalized view of constraints in the AI world. Two areas of research emerged: Constraint Logic Programming (CLP) and Constraint Satisfaction Problems (CSP). The CLP efforts are aimed at including numeric and predicate constraints in a Logic Programming environment such as Prolog.

The CSP work, is more relevant to the CMS effort. Early CSP work [Waltz 75, Haralick & Shapiro 79, Haralick & Shapiro 80, Mackworth & Freuder 85, Nadel 85] developed on the notion of constraint networks and the management of consistencies in such networks. This research, however, has concentrated primarily on unary and binary constraints [Dechter & Pearl 87]. Efforts to extend this to engineering domains have concentrated on: (a) combining numeric and logical constraints [Serrano 87b, Gross 86, Borning 79], (b) abstraction and simplification of complex networks [Navin Chandra & Rinderle 89a], (c) constraint relaxation [Fox 83, Navin chandra 87], (d) Conflict detection and coordination of constraints [Sriram 87, Karnandikar et.al. 91], and (e) Texture Measures of constraint activity [Fox et al. 89].

There are several other systems such as Decision Support Systems and Project Management Systems that take into account, a variety of resource and precedent constraints. These systems, however, have the relevant constraints compiled into their algorithms. There is no notion of treating constraints as both data and procedures as we do in CMS. This is true of our own previous work in project management [Navin chandra & Logcher 85].

Since the time work began on Design Fusion in 1987, viewing design as a constraint-directed process has gained wide acceptance in design community. Serrano and Gossard were one of the first to explore constraint-directed design in detail [Serrano 87a]. Bowen and his colleagues at North Carolina have developed a series of constraint solving languages [Bowen & Bahler 91]. Rinderle and his colleagues have explored an number of issues in constraint propagation and abstraction.

8. Conclusion

The successful design of products requires that goals and constraints from across the product life cycle be satisfied. Consequently we believe that the ability to represent and reason about constraints is central to the design process.

Constraints are more than simple predicates, but span the continuum from quantitative to qualitative relationships. In addition, their importance varies, they may be relaxed, and they must be sensitive to the context in which they are to be applied. Therefore, the representation of constraint knowledge has to equally rich. Secondly, the ability to reason about constraints must be equally rich. As a design evolves from incomplete to complete, from abstract to detail,

the constraint reasoning process must be able to evaluate relevant constraints to the extent possible. Lastly, the process by which constraints are revealed, revised, and retracted is critical to the concurrent design process. Having the appropriate protocols for coordinating the revelation and revision of constraints during the design process will enable more rapid convergence.

The Constraint Management System of Design Fusion provides an approach to representing, reasoning and managing constraints. By means of a frame-based representation, a variety of propagation techniques including interval arithmetic and CSP techniques, and a protocol by which perspectives coordinate the actions, the CMS is able to participate directly in the design process by quickly identifying inconsistencies and proposing which variables to focus on next.

The success of the CMS is not only limited to Design Fusion, but has also been adopted by the DICE[2] program and by a large power equipment manufacturer.

9. Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Agency under contract No. MDA972-88-C-0047 for the DARPA Initiative on Concurrent Engineering (DICE), the National Science Foundation under the Engineering Research Centers Program, Grant CDR-8522616, Asea Brown Boveri, the Natural Sciences and Engineering Research Council of Canada, Digital Equipment Coporation, the Micro Electronics and Computer Research Corp., SPAR Aerospace, Carnegie Group Inc., and Quintus Corp.

10. References

[Aho, Hopcroft & Ullman 83]
Aho, A.V., J.E. Hopcroft, J.D. Ullman.
Data structures and algorithms.
Addison-Wesley, 1983.

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

[Bowen & Bahler 91]
Bowen, J., and Bahler, D.
Supporting Cooperation Between Multiple Perspectives in a Constrainted-Based Approach to Concurrent Engineering.
Journal of Design and Manufacturing :89-105, 1991.

---

[2]DARPA Initiative in Concurrent Engineering.

[Dechter & Pearl 87]
Dechter, R. and Pearl, J.
Network-based Heuristics for Constraint-Satisfaction Problems.
Artificial Intelligence 34(1):1-38, 1987.

[Erman et al. 80] Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D.R.
The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty.
ACM Computing Surveys 12(2):213-253, 1980.

[Fox 83] Fox, M.S.
Constraint Directed Search: A case of Job Shop Scheduling.
PhD thesis, Carnegie-Mellon University, 1983.

[Fox 87] Fox, M.S.
Constraint-Directed Search: A Case Study of Job-Shop Scheduling .
Morgan Kaufmann Publishers, Inc., 1987.

[Fox et al. 89] Fox, M.S., Sadeh, N., and Baykan, C.
Constrained Heuristic Search.
In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 309-316. Morgan Kaufmann Pub. Inc., 1989.

[Fox et al. 92] Fox, M.S., Finger, S., Gardner, E., Navin chandra, D., Safier, S.A., and Shaw, M.
Design Fusion: An Architecture for Concurrent Design.
Knowledge-aided Design.
In Green, M.,
Academic Press Ltd., 1992, pages 157-195.

[Gross 86] Gross, M.D.
Design as Exploring Constraints.
PhD thesis, M.I.T., 1986.

[Haralick & Shapiro 79]
Haralick, R.M., L.G. Shapiro.
The Consistent Labeling Problem: Part I.
Trans. Pattern Anal. Machine Intelligence 1(2):173-184, 1979.

[Haralick & Shapiro 80]
Haralick, R.M., L.G. Shapiro.
The Consistent Labeling Problem: Part II.
Trans. Pattern Anal. Machine Intelligence 2(3):193-203, 1980.