

# **An Attribute-Space Representation and Algorithm for Concurrent Engineering**

Timothy P. Darr and William P. Birmingham

CSE-TR-221-94

October 1994

Computer Science and Engineering Division  
Room 3402 EECS Building

The University of Michigan  
Department of Electrical Engineering and Computer Science  
Ann Arbor, Michigan 48109-2122  
USA

Copyright October 1994

# An Attribute-Space Representation and Algorithm for Concurrent Engineering

Timothy P. Darr (timd@eecs.umich.edu)  
William P. Birmingham (wpb@eecs.umich.edu)  
Department of Electrical Engineering and Computer Science  
The University of Michigan  
1101 Beal Avenue  
Ann Arbor, MI 48109  
(313) 936-1590  
Fax: (313)763-4617

## **ABSTRACT**

This paper presents a novel formulation of the configuration-design problem that achieves the benefits of the concurrent engineering (CE) design paradigm. In CE, all design concerns (manufacturability, testability, etc.) are applied to an evolving design throughout the design cycle. CE identifies conflicts early on, which avoids costly redesign and can lead to better products. Our formulation is based on a distributed, dynamic, interval constraint-satisfaction problem (DDICSP) model. Persistent *catalog agents* map onto DDICSP variables and *constraint agents* map onto DDICSP constraints. These agents use a set of operations and heuristics to navigate through the space of possible designs to rapidly eliminate sets of designs until a solution is found. Experimental results show that an architecture where each catalog agent resides on a separate computer has performance advantages over non-distributed approaches.

## **1 Introduction**

Design of large-scale artifacts involves consideration of hundreds or thousands of often competing concerns such as manufacturability, testability, cost, etc. The *concurrent engineering* (CE) approach to product development identifies potential problems in the design while still easy to correct, avoids costly redesign, and reduces the time needed to bring high-quality products to market. Many CE approaches focus on facilitating *communication* through standard data-interchange formats, communication protocols, electronic colocation, etc. Our claim is that even if communication barriers were removed, *coordination problems* prevent realization of CE goals.

This paper presents a more effective formulation of the design problem and design process. We base our design process and representation on a distributed, dynamic, interval constraint-satisfaction problem (DDICSP) formulation of the design problem, where *catalog agents* map onto DDICSP variables and *constraint agents* map onto DDICSP constraints. The catalog agent's *multi-valued* domains are attribute intervals that represent the set of values for all parts in the catalog; essentially creating a space of designs possible with the components in each catalog. Constraints are defined over the domains, and provide an effective way to decompose the problem into *attribute spaces* in such a way that all constraints (i.e., downstream concerns) are treated equally and can be applied in parallel throughout the design process. This paper develops a coordination algorithm and set of operations that allow the catalog and constraint agents to navigate through these spaces. The algorithm uses a set of heuristics to rapidly eliminate sets of designs until a solution is found.

## **2 The Automated Configuration-Design Service (ACDS)**

This section describes the Automated-Configuration Design Service (ACDS) [1, 2], which is the implementation of our approach. ACDS applies to any design activity where *parts* are described by attributes. This includes configuration design [3-10], where a designer

selects and connects components from catalogs to provide user-defined functionality subject to performance and feasibility constraints. A large number of products are developed from catalogs of components, which makes configuration design an economically important type of design.

ACDS is a collection of loosely-coupled, autonomous agents that organize communication among themselves based on design constraints. These agents represent part catalogs and design constraints, and consist of *catalog agents*, *system agents*, *bid agents* and *constraint agents*<sup>1</sup>. ACDS agents are distributed functionally and geographically, and communicate by passing messages. Thus, agents can reside anywhere in the country and still be part of ACDS. To use ACDS, a designer provides a high-level specification of the desired design and uses this to configure the ACDS network. This specification includes the functions to be performed, their interconnections and performance specifications (bounds on cost, failure rate, *etc.*).

## 2.1 Problem Definition

We now define the problem formally. Given:

- A set of distributed electronic catalog agents that represent part catalogs.  
 $CA = \{ca_1, \dots, ca_n\}$   
 $p_{ji}$  is a part in the catalog represented by  $ca_i$ .
- A set of preferentially independent, design attributes that describe the parts [11].  
 $A = \{a_1, a_2, \dots, a_n\}$ .  
For all  $p_{ji}$  and  $a_k$ ,  $p_{ji}.a_k$  is the value of attribute  $a_k$  for part  $p_{ji}$ .
- A linear utility function  $u(p_{ji})$  [11].  
For all  $p_{ji}$ ,  $p_{ji}$ -utility =  $u(p_{ji})$  is a value representing the utility of  $p_{ji}$ .
- A set of distributed constraint agents that represent feasibility constraints.  
 $C = \{C_1, C_2, \dots, C_p\}$ .

Find:

- A set of parts  $S = \{p_{ji}, i = 1, \dots, m\}$  such that:
  - $S$  satisfies all feasibility constraints.

## 2.2 The Attribute-Space Representation

We assume that a design and the parts that compose it can be represented by a set of attributes. This assumption is consistent with those of configuration and parametric design, and with various other design methodologies. The *attribute-space representation* of the design space, given in Definitions 1 and 2, is a set of intervals<sup>2</sup> specifying the set of values, over all attributes, for all designs in the design space [2, 12, 13]. The attribute-space representation provides a compact, abstract representation of a large number of designs, and facilitates efficient reasoning about sets of design possibilities.

**Definition 1:** Let  $A = \{a_1, a_2, \dots, a_n\}$  be the set of attributes that characterize a complete design and  $D_i = (D_i.a_1, D_i.a_2, \dots, D_i.a_n)$  be an  $n$ -tuple that defines a single design, where  $D_i.a_k$  is the value of attribute  $a_k$  for design  $D_i$  ( $D_i.a_k$  may be NULL if  $a_k$

<sup>1</sup> An agent is a computational process with expertise about a limited portion of a design problem, and are capable of achieving specific goals. Agents have the capability to direct other agents to perform operations within the context of our design representation and algorithm.

<sup>2</sup>  $x = [a b]$  is the interval  $a \leq x \leq b$ ;  $x = [a b)$  is the interval  $a \leq x < b$ ;  $x = (a b]$  is the interval  $a < x \leq b$ ; and  $x = (a b)$  is the interval  $a < x < b$ .

is undefined for  $D_i$ ). The set of all possible designs, or *design space*, is  $DS = \{D_1, D_2, \dots, D_m\}$ .

**Definition 2:** Let  $A$  be the set of attributes and  $DS$  be the design space from Definition 1. The *attribute-space representation* of  $DS$  is  $[A] = \{[a_1], [a_2], \dots, [a_n]\}$ , where  $[a_k]^3 = [\min(D_i.a_k) \max(D_i.a_k)]$ , for all  $D_i$  and  $a_k$ .

There are possibly an infinite number of designs in the design space if each design is not composed of discrete elements (e. g., parametric vs. configuration design).

### 2.3 Constraint-Based Decomposition

Not all designs in the design space are physically possible or desired by the designer. A constraint is a relation over a subset of the attribute space that defines *physically* feasible designs. Viewed individually, a constraint is a projection onto a *subset* of the attribute space that defines the constraint's feasible region. A constraint is satisfied when no design lies outside its feasible region.

To decompose the problem, we assign an *agent* to each constraint. If the bound of the attribute space exceeds the bound of the feasible region of the constraint, the agent removes designs outside the feasible region to *shrink* the attribute space. It is easy to detect which designs to remove by examining the bounds of the attribute space. Agents concurrently and independently shrink the space until it lies entirely within the feasible region, yielding a space containing only feasible designs. The design problem is solved when there are no designs outside the feasible region of any constraint.

**Definition 3:** Let  $[A]$  be the attribute-space representation defined in Definition 2, and  $C_j$  be a constraint defined over a subset design attributes  $A$  ( $A_j \subseteq A$ ). The *constraint-based decomposition* is the subset of the attribute space ( $[A_j] \subseteq [A]$ ) defining the feasible region for  $C_j$ .

### 2.4 Dynamic, Distributed Interval Constraint-Satisfaction Problem Formulation

We now present the distributed, dynamic interval constraint-satisfaction problem (DDICSP) computational model that forms the foundation of the ACDS approach [2]. The DDICSP is an amalgamation of the constraint satisfaction problem (CSP) [14, 15], the interval constraint-satisfaction problem (ICSP) [13, 16], the distributed constraint-satisfaction problem (DCSP) [17, 18], and the dynamic constraint-satisfaction problem [19-21] (DCSP).

The ICSP assumes that domain values are intervals and uses standard interval arithmetic to evaluate constraints. Interval arithmetic is well-defined for the addition, subtraction, multiplication and division (when the denominator is non-zero) operators [22].

We define the DDICSP as follows:

- $CA = \{ca_1, ca_2, \dots, ca_m\}$  is a set of part catalogs (variables in CSP terminology). A *catalog agent* is assigned to each  $ca_i$ .
- $[ca_i] = \{[ca_i.a_1], [ca_i.a_2], \dots, [ca_i.a_n]\}$ , is the multi-valued attribute-space domain of  $ca_i$ , where  $[ca_i.a_k]$  is the *set* of values of the attribute  $a_k$  for all parts in  $ca_i$ . If  $a_k$  is

<sup>3</sup> We distinguish interval-valued variables from scalar-valued variables by using bold brackets [].

not defined for  $ca_i$ , then  $[ca_i.a_k] = \text{NULL}$ . The global attribute space AS from Definition 2 is  $[A] = \bigcup_i [ca_i]$ .

- $C = \{C_1, C_2, \dots, C_p\}$  is a set of independent, monotonic, non-directional constraints ( $C_j = f(ca_i.a_k \in A_j)$ ) that restrict the assignment of values to the domain of each  $ca_i$ . (Non-directionality means that given the values of  $n-1$  variables, the value of the  $n$ th variable can be inferred). Each  $C_j$  has a predicate that indicates when it is active and defines a problem decomposition  $[A_j] \subseteq [A]$ , as in Definition 3. We assign a constraint agent to each  $C_j$ .
- The DDICSP is a graph  $G(CA, C)$ .

A solution to the DDICSP is:

- an assignment  $ca_i = \{\alpha_1, \dots, \alpha_k, \dots, \alpha_n\}$ ,  $\alpha_k \in [a_{ik}]$ ,  $\forall [a_{ik}] \in [A_i]$ ,  $\forall ca_i \in CA$ , that satisfy the set of constraints  $C$ .

Figure 2 shows an example DDICSP network for an elevator-configuration problem. The design consists of a motor, counterweight (cwt), cab and cable. The attribute-space domain appears next to each catalog. Figure 3 shows the part sets for the example. The utility function for this example is a weighted sum of *attribute-value scores*, where the weight of the dollar attribute-value score is 0.9 and the weight of the fpmh attribute-value score is 0.1 ( $u(p_{ji}) = 0.9 * v(\text{dollar}, p_{ji}) + 0.1 * v(\text{fpmh}, p_{ji})$ ). If the user prefers low cost, low failure rate designs, we *normalize* the actual attribute value of each part to get the attribute-value scores.

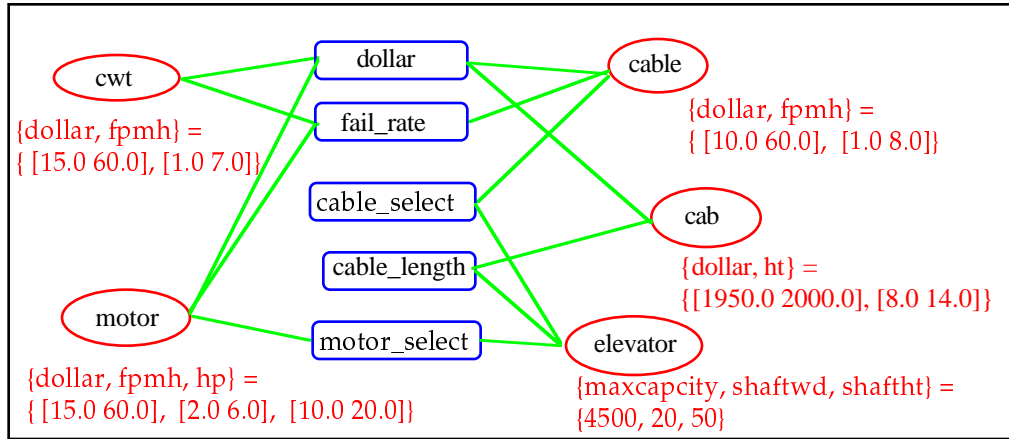


Figure 2: DDICSP Network

In traditional or discrete CSPs, the catalog-agent domain is a set of parts. The space of all possible designs is given by the cross product of the domains of each variable:  $\text{dom}(ca_1) \times \text{dom}(ca_2) \times \dots \times \text{dom}(ca_n)$ . This corresponds to the rows of Figure 3. In the ACDS formulation, the catalog-agent domain is the set of intervals that represent the *columns* in the part-catalog tables of Figure 3. The space of all possible designs in the ACDS representation is given by the set of all catalog-agent domains:  $\bigcup_i [ca_i]$ .

Table 1 shows the five constraints<sup>4</sup>: motor\_select, cable\_select, cable\_length, dollar and fail\_rate found in the example. The fail\_rate and dollar constraints are static (their

<sup>4</sup> Not shown in this figure are four additional constraints that specify the requirement that the parts

precondition is always true), and provide an upper bound on the total dollar cost and failure rate of the final design. The motor\_select constraint, also static, provides bounds on the motor horsepower, dependent on the user-specified maximum capacity of the elevator. The cable\_length constraint restricts the length of the cable based on the height of the cab and specifications of the elevator shaft. The cable\_select constraint is dynamic, and specifies that a particular cable is not to be chosen if the elevator capacity is greater than 4000.0 pounds.

Motor					Cable				
Part	dollar	fpmh	hp	utility	Part	dollar	length	fpmh	utility
m1	60.0	2.0	10.0	0.10	cb11	60.0	70.0	1.0	0.10
m2	35.0	2.0	10.0	0.60	cb12	30.0	70.0	3.0	0.61
m3	25.0	3.0	10.0	0.78	cb13	25.0	65.0	3.5	0.69
m4	20.0	6.0	20.0	0.80	cb14	20.0	60.0	4.0	0.78
m5	20.0	4.0	20.0	0.85	cb15	10.0	60.0	8.0	0.90
m6	15.0	6.0	10.0	0.90	cb16	10.0	60.0	4.0	0.96

Counterweight					Cab			
Part	dollar	fpmh	utility		Part	dollar	ht	utility
cwt1	60.0	1.0	0.10		cab1	2000.0	14.0	0.00
cwt2	40.0	7.0	0.40		cab2	1990.0	12.0	0.18
cwt3	30.0	5.0	0.63		cab3	1980.0	11.0	0.36
cwt4	25.0	2.0	0.78		cab4	1970.0	10.0	0.54
cwt5	20.0	2.5	0.88		cab5	1960.0	9.0	0.72
cwt6	15.0	4.0	0.90		cab6	1950.0	8.0	0.90

Figure 3: Initial Part Sets

Table 1: Example Problem Constraints

Constraint Name	Precondition	Constraint
dollar	true	total cost $\leq$ \$2035.00
fail_rate	true	total failure rate $\leq$ 10 fpmh

**Definition 6:** The DDICSP network is *completely node- and arc-consistent* iff, for all  $C_j$  such that  $C_j$  is active and unary,  $C_j$  is node-consistent, and for all  $C_j$  such that  $C_j$  is active and non-unary,  $C_j$  is arc-consistent. The attribute space that corresponds to this network is called the *completely node- and arc-consistent attribute space*  $[A]^{ac}$ .

Figure 4 shows the initial attribute spaces for the part sets in Figure 3. The elevator agent defines the elevator specifications:  $\{\text{maxcapacity} = 4500.0, \text{shaftht} = 50.0, \text{shaftwd} = 12.0\}$  and contains only one part, so we omit it from Figure 3 and all subsequent figures. The top row in Figure 4 shows the decomposition: dollar, fail\_rate, motor\_select and cable\_length<sup>5</sup>.

dollar	fail_rate	motor_select	cable_length
dollar	fpmh	hp	length ht



*attribute spaces*  $(A)^6$ , which move the network toward complete-network decomposability by tightening the bounds on the catalog-agent attribute spaces. The catalog agents map from these spaces to the discrete space by removing parts. This process continues until a solution is found.

To achieve network consistency, the catalog agents form the attribute-space representation of its parts. The constraint agents create *node- and arc-consistent attribute spaces*  $[A_j]^{ac}$ ,  $([A]^{ac} = \bigcup_j [A_j]^{ac})$  using the catalog-agent-generated attribute spaces and Definitions 4 and 5. The catalog agents remove parts that lie outside  $[A]^{ac}$  to make themselves node- and arc-consistent (note, this is an inverse operation to arc-consistency or node-consistency). Once there are no more parts to throw out, the network is completely node- and arc-consistent.

The *bid* phase evaluates the current attribute space with respect to the constraints. If a constraint is not decomposable, then the constraint agent creates a *tightened attribute space*,  $(A_j)$   $((A) = \bigcup_j (A_j))$ . The catalog agents individually identify parts that, if removed, would yield an attribute space that lies within  $(A_j)$ . A *catalog-agent proposal*  $ca_i^{PR}$  consists of a set of tuples  $(p_{ji}, \{(A_k)\}, ca_i.utility)$  such that if  $p_{ji}$  were removed, the attribute space would lie within  $\{(A_k)\}$ .  $ca_i.utility$  is the interval of utility values that would result if  $p_{ji}$  were removed. The catalog-agents determine the parts to remove without regard to the effect that this action might have on other agents. Removal of individual parts in this way is equivalent to removing designs that violate the constraints, since removed parts are those that appear in possibly infeasible designs.

The *reduce violations* phase shrinks the attribute space, thus moving the network toward complete-network decomposability. The bid agent accepts a set of catalog-agent proposals, one for each non-decomposed  $C_j$  (details of the selection algorithm are described later). The catalog agents remove the parts in the accepted catalog-agent proposals, which creates a new attribute space. The network performs a *feasibility check* on the attribute space to remove parts that have become infeasible as a result of the accepted proposals. If the attribute space is empty, then the bid agent accepts a new set of catalog-agent proposals. We define a *cycle* as the bid and reduce violation phases.

The process continues until complete network decomposability is achieved. At that point, the catalog agents select their highest utility part as a solution. Table 2 summarizes the ACDS notation.

### 3.1 Network-Consistency Operations

Consistency operations that create node-  $([A]^{nc})$  and arc-consistent  $([A]^{ac})$  attribute spaces are the following:

**create\_node\_consistent\_space:** This operation creates a node-consistent attribute space  $[A_j]^{nc}$  for decomposition  $[A_j]$ , as defined in Section 2.5  $([A]^{nc} = \bigcup_j [A_j]^{nc})$ .

**create\_arc\_consistent\_space:** This operation creates an arc-consistent attribute space  $[A_j]^{ac}$  for decomposition  $[A_j]$ , as defined in Section 2.5  $([A]^{ac} = \bigcup_j [A_j]^{ac})$ .

---

<sup>6</sup> The notation  $(A)$  means that the bounds of the space  $[A]$  are tightened.

For our example, each constraint agent applies the `create_node_consistent_space` and `create_arc_consistent_space` operations to create the node- and arc-consistent attribute space. Each agent solves for each of the attributes in its space  $A_j$  yielding a range of allowable values for that attribute. We illustrate this for the `cable_length` constraint agent:

`cable_length`:

$$\begin{aligned} \text{cable.length} &\geq \sqrt{(\text{elevator.shaftht} - \text{cab.ht})^2 + (\text{elevator.shaftwd} / 2)^2} + 20 \\ &\geq \sqrt{([50 \ 50] - [8 \ 14])^2 + ([20 \ 20] / 2)^2} + 20 \\ &\geq [57.36 \ 63.17] \\ \text{cab.ht} &\geq \text{elevator.shaftht} - \sqrt{(\text{cable.length} - 20)^2 - (\text{elevator.shaftwd} / 2)^2} \\ &\geq [1.01 \ 11.27] \end{aligned}$$

Table 2: Summary of ACDS Notation

Notation	Meaning
CA	Set of catalog agents
$ca_i$	Catalog agent $i$
AS	Attribute space
$[A]^{ac}$	Arc-consistent attribute space
(A)	Tightened attribute space
$AS_i^T$	Tightened attribute space for $ca_i$
$ca_i^{PR}$	Catalog agent proposal
C	Set of constraints to be met
$C_j$	Constraint agent $j$
$[A_j]$	Decomposition for $C_j$
$[A_j]^{ac}$	Node- or arc consistent space for $C_j$
( $A_j$ )	Tightened attribute space for $C_j$
$P_i$	Part set for $ca_i$
$p_{mi}$	Part $m$ in $ca_i$
$p_{mi}.a_k$	Value of attribute $a_k$ for part $p_{mi}$
$p_{mi}.utility$	Utility value for $p_{mi}$
$ca_i.utility$	Range of utility values for $ca_i$
$a_{ik}$	Range of values for $a_k$ in $ca_i$

These calculations define the bounds on the arc-consistent attribute space for the `fail_rate` and `cable_length` decomposition. The `cable_select`, `motor_select` and `dollar` constraint agents perform similar calculations to yield their node- and arc-consistent attribute spaces. Figure 5 shows the entire arc-consistent attribute space.

The space-transformation operation transforms one attribute space into another attribute space as follows:

**create\_space:** This operation creates an attribute space  $[A] = \bigcup_i [ca_i]$  that lies within an attribute space  $[A]^* = \bigcup_i [ca_i]^*$ . Creating a space that lies within  $[A]^*$  is equivalent to removing parts outside  $[A]^*$ .

$$\begin{aligned} [ca_i.a_k] &= [\min(p_{mi}.a_k) \max(p_{mi}.a_k)], \\ &\text{for all } p_{mi} \text{ such that } p_{mi}.a_k \in [ca_i.a_k]^*, \text{ for all } [ca_i.a_k]^* \in [A]^* \end{aligned}$$

	dollar dollar	fail_rate fpmh	motor_select hp	cable_length length ht		cable_select
motor	[0.0 60.0]	[0.0 8.0]	(9.0 18.0)			
cable	[0.0 55.0]	[0.0 7.0]		[57.36 ∞)		not cbl3
cwt	[0.0 60.0]	[0.0 7.0]				
cab	[0.0 1995.0]				[1.01 ∞)	

Figure 5: Arc-Consistent Attribute Space

Figure 4 shows the initial attribute spaces for the part sets in Figure 3. These spaces are based on an attribute space that lies within  $[A]^* = \{ca_i.a_k = [-\infty \infty], \text{ for all } ca_i \in CA, \text{ for all } a_k \in A\}$ . The catalog agents apply the create\_space operation to create a space that lies within the space of Figure 5. This is done by removing m4, m5, cbl1, cbl3, cbl5 and cab1. Figure 6 shows the new space.

	dollar dollar	fail_rate fpmh	motor_select hp	cable_length length ht	
motor	[15.0 60.0]	[2.0 6.0]	[10.0 10.0]		
cable	[10.0 30.0]	[3.0 6.0]		[60.0 70.0]	
cwt	[15.0 60.0]	[1.0 7.0]			
cab	[1950.0 1990.0]				[8.0 12.0]

Figure 6: Updated Attribute Space

Since parts were removed, the consistency operations are applied again, which leads to the removal of cwt2 by the counterweight catalog agent. The new attribute space lies entirely within  $[A]^{ac}$ , so no more parts are removed.

### 3.2 Bid Operations

Returning to our example, each constraint agent determines if it is completely decomposable. This is shown below for the cable\_length constraint agent.

cable\_length:

$$\text{cable.length} \geq \sqrt{(\text{elevator.shaftht} - \text{cab.ht})^2 + (\text{elevator.shaftwd} / 2)^2} + 20$$

$$[60 \ 70] \geq \sqrt{([50 \ 50] - [8 \ 12])^2 + ([20 \ 20] / 2)^2} + 20$$

$$[60 \ 70] \geq [59.29 \ 63.17]$$

This constraint is not decomposable. The lower bound of the left-hand side interval for the cable\_length constraint is less than the upper bound of the right-hand side interval, which violates the constraint. This means that there exists at least one design that violates the constraint, namely, the design that contains the cable whose length is 60 ft and the cab whose height is 12 ft. By eliminating this design, the cable\_length constraint will move closer to decomposability.

### 3.3 Reduce Violations Operations

Tightening operations create a tightened attribute space (A), and catalog-agent proposals  $[ca_i]^{PT}$  to move the network toward complete-network decomposability. Given  $[A]$  and C, the constraint agents generate a *tightened attribute space*, (A) by determining for each variable  $ca_i.a_k \in A_j$  of the constraint represented by  $C_j$  if the lower or upper bound of  $[ca_i.a_k]$  should be tightened to move the constraint toward decomposability. For each constraint  $C_j$  the constraint agent solves and evaluates each  $ca_i.a_k \in A_j$ . Let the result of this be the *calculated*

interval  $[ca_i.a_k]^{calc}$ , which specifies the set of values for each  $ca_i.a_k$  given the values of all the other variables. The result of this calculation is one of the following expressions, depending on the form of the constraint:

$$ca_i.a_k = f^{-1}(ca_m.a_n \in A_j, m \neq i, n \neq k) = [ca_i.a_k]^{calc} \quad (1)$$

$$ca_i.a_k \neq f^{-1}(ca_m.a_n \in A_j, m \neq i, n \neq k) = [ca_i.a_k]^{calc} \quad (2)$$

$$ca_i.a_k < f^{-1}(ca_m.a_n \in A_j, m \neq i, n \neq k) = [ca_i.a_k]^{calc} \quad (3)$$

$$ca_i.a_k \leq f^{-1}(ca_m.a_n \in A_j, m \neq i, n \neq k) = [ca_i.a_k]^{calc} \quad (4)$$

$$ca_i.a_k > f^{-1}(ca_m.a_n \in A_j, m \neq i, n \neq k) = [ca_i.a_k]^{calc} \quad (5)$$

$$ca_i.a_k \geq f^{-1}(ca_m.a_n \in A_j, m \neq i, n \neq k) = [ca_i.a_k]^{calc} \quad (6)$$

The original interval  $[ca_i.a_k]$  is compared to the calculated interval  $[ca_i.a_k]^{calc}$  to determine which bound of  $[ca_i.a_k]$  to tighten. For  $C_j$  to be decomposable, *every* value in  $[ca_i.a_k]$  must satisfy the expression for *every* value in  $[ca_i.a_k]^{calc}$ . For example, suppose we have the equation  $ca_i.a_k \leq [ca_i.a_k]^{calc}$  and  $\max([ca_i.a_k]) > \min([ca_i.a_k]^{calc})$ . In this case, there exists at least one value in  $[ca_i.a_k]$  and one value in  $[ca_i.a_k]^{calc}$  that do not satisfy the constraint. By tightening the upper bound on  $[ca_i.a_k]$ , which would remove an offending value, we will move closer to decomposability.

Table 3 shows interval-tightening rules<sup>7</sup>. The column headings indicate the operator in the expression for  $ca_i.a_k$  (from equations (1)-(6)), and the rows indicate the condition on the bounds of  $[ca_i.a_k]$  and  $[ca_i.a_k]^{calc}$ . The entry in position (i, j) is tightened attribute space element for  $ca_i.a_k$  if the condition in row i is true and the expression operator in column j applies.

**Definition 9:** The *tightened-attribute space* for  $[A_j]$  is the set of all tightened intervals  $ca_i.a_k \in [A_j]$  defined Table 3. The *tightened-attribute space* is given by  $(A) = \bigcup_j (A_j)$ .

**tighten\_attribute\_space:** This operation creates a tightened attribute space  $(A_j)$  that moves the constraint  $C_j$  toward decomposability, as defined above.

Table 3: Attribute-Space Tightening

	<	≤	>	≥
$\min([ca_i.a_k]) < \max([ca_i.a_k]^{calc})$			$(ca_i.a_k)$	$(ca_i.a_k)$
$\min([ca_i.a_k]) \leq \max([ca_i.a_k]^{calc})$			$(ca_i.a_k)$	$(ca_i.a_k)$
$\max([ca_i.a_k]) > \min([ca_i.a_k]^{calc})$	$[ca_i.a_k)$	$[ca_i.a_k)$		
$\max([ca_i.a_k]) \geq \min([ca_i.a_k]^{calc})$	$[ca_i.a_k)$	$[ca_i.a_k)$		

The catalog agents use the tightened attribute space to create catalog-agent proposals,  $[ca_i]^{pr}$ , one for each non-decomposed constraint. The catalog agent proposes a part or set of parts, that if removed, would lie within the tightened attribute space. These are the parts that lie on the bound of the previous attribute space. The decision about what parts to remove is made by the bid agent, who uses an algorithm described below to accept and reject proposals

<sup>7</sup> Equality and inequality constraints can be replaced by a pair of inequality constraints.

from specific catalog agents. A catalog agent proposal consists of a set of parts to remove  $\{p_{ji}\}$ , the set of tightened-attribute spaces the proposal applies to  $\{[A_k]\}$ , and the utility interval of the catalog if the part are removed ( $ca_i.utility^{pr}$ ).

**create\_proposal:** This operation creates catalog-agent proposals by identifying parts that, if removed, would create an attribute space that lies within (A).

$[ca_i]^{pr} = \{(\{p_{ji}\}, \{[A_k]\}, ca_i.utility^{pr}) \text{ such that } p_{ji}.a_k \notin [ca_i.a_k] \in (A_k), ca_i.utility^{pr} \text{ is the utility interval of the catalog if } \{p_{ji}\} \text{ is removed}\}$

Returning again to our example, the constraint agents apply the `tighten_attribute_space` operation to determine which bounds to tighten. We illustrate this for `cable_length` constraint agent.

`cable_length:`

$$cable.length \geq \sqrt{(elevator.shaftht - cab.ht)^2 + (elevator.shaftwd / 2)^2} + 20$$

$$[60 \ 70] \geq [59.29 \ 63.17]$$

since  $\min([ca_i.a_k]) < \max([ca_i.a_k]^{calc})$ , *tighten lower bound of [60 70]*

$$cab.ht \geq elevator.shaftht - \sqrt{(cable.length - 20)^2 - (elevator.shaftwd / 2)^2}$$

$$[8 \ 12] \geq 50 - \sqrt{([60 \ 70] - 20)^2 - ([20 \ 20] / 2)^2}$$

$$[8 \ 12] \geq [1.01 \ 11.27]$$

since  $\min([ca_i.a_k]) < \max([ca_i.a_k]^{calc})$ , *tighten lower bound of [8 12]*

Figure 7 shows the tightened attribute space and Figure 8 shows the catalog-agent proposals. The `motor_select` constraint is decomposable, and does not appear in either table. It is easily verified that the proposed attribute space lies entirely within the tightened attribute space.

dollar	fail_rate	cable_length	
dollar	fpmh	length      ht	

1.  $C = \{C_1, \dots, C_n \mid C_j (1 \leq j \leq n) \text{ is not decomposed}\}$
2.  $P = \{ \}$
3. while ( $C$  not NULL)
4.  $pr = \{(p_{ji}, \{(A_k)\}, ca_i, utility)\}$
5. sort  $pr$  by  $|(A_k)|$  breaking ties by  $\max(ca_i, utility)$ <sup>8</sup>
6. for all  $p_{ji} \in pr$
7.      $P = P \cup pr$
8.      $C = C - \{C_j \text{ such that } p_{ji} \text{ applies to } (A_j)\}$
9.     if  $C$  is NULL
10.         accept the proposals in  $P$ .

Figure 9: accept\_proposal algorithm

The bid agent uses the heuristic: *accept the proposal that apply to the most  $(A_k)$* . To break ties, the bid agent selects the proposal with the highest utility interval value. Once the bid agent accepts a proposal, the bid agent removes from consideration all  $(A_k)$  that the accepted proposal applies to. The bid agent accepts proposals in this manner for each constraint. Chronological backtracking is used if a dead-end is reached.

Figure 10 shows the catalog-agent proposals. The bid agent uses the accept\_proposal operation, to accept cbl6, since it applies to the greatest number of tightened spaces (fpmh and cable\_length). The bid agent next removes from consideration the proposals m6, cwt3 and cab6. The bid agent then accepts cbl2 since its utility interval contains the highest value of the remaining proposals (those that apply to the dollar constraint).

The bid agent accepts the proposals {cbl2, cbl6} and rejects all others, so the cable catalog agent removes cbl6 and cbl2 from its catalog. This results in an empty catalog for the cable catalog agent. There is no solution, so the bid agent *backtracks* to accept another set of proposals. It goes back to the last choice point, which was to accept a proposal for the dollar space, and accepts the next best proposal (cwt1). The new set of accepted proposals is {cwt1, cbl6}. The catalog agents remove these parts from their respective catalogs, followed by a feasibility check (node- and arc-consistency) on the new attribute space. Several additional cycles are required to achieve complete-network decomposability.

Proposal	Constraints		
	dollar	fpmh	cable_length
remove m1	[0.60 0.90]		
remove m6		[0.10 0.78]	
remove cbl2	[0.78 0.96]		
remove cbl6 (cbl4)		[0.61 0.78]	[0.61 0.61]
remove cwt1	[0.63 0.90]		
remove cwt3		[0.10 0.90]	
remove cab2	[0.36 0.90]		
remove cab6			[0.18 0.72]

Figure 10: Proposal Acceptance

Figure 16 shows the complete ACDS algorithm. The following theorem establishes an important property of the ACDS algorithm.

**Theorem 1:** If there is a solution, the ACDS algorithm will find it.

<sup>8</sup> The max() function returns the upper bound of an interval.

**Proof:** When a dead-end is reached, ACDS uses chronological backtracking to accept a new set of proposals to shrink the space. Chronological backtracking is complete (*i.e.* it will find a solution if one exists), therefore, ACDS is also complete since it maps directly onto chronological backtracking.

### 3.4 Discussion

The complexity of the problem being solved is  $O(|parts|^{functions})$  (Balkany *et. al.* [30] gives a more exact complexity calculation for a class of configuration problems), where  $|parts|$  is the average number of parts in all catalogs and  $functions$  is the number of functions to implement. For even relatively small problems, the combinatorics many render the problem intractable. Some researchers have thus concluded that automated design is impractical or impossible [31]. Our algorithm manages the complexity in the following ways.

The attribute-space representation in the nominal case drastically reduces  $|parts|$  by abstracting individual components into a more compact representation. This representation exploits the regularity present in many part catalogs, where parts are good with respect to one attribute but poor with respect to other attributes.

#### Initialization

1) Network established by identifying catalog and constraint agents.

#### Achieve Network Consistency

- 2) Catalog agents form attribute spaces  $[A_j]$ .
- 3) Constraint agents apply `create_node_consistent_space` and `create_arc_consistent_space` to create node- and arc-consistent attribute spaces  $[A_j]^{nc}$  and  $[A_j]^{ac}$ .
- 4) Catalog agents apply `create_space` to create attribute space that lies within  $[A_j]^{ac}$ .
- 5) Repeat steps 2) through 4) until no more parts are removed. If the attribute space is empty, then terminate with failure, since no feasible solution exists.

#### Bid

6) Constraint agents evaluate attribute space and apply the `tighten_attribute_space` operation to create tightened attribute spaces  $(A_j)$ .

#### Reduce Violations

- 7) Catalog agents apply the `create_proposal` operation.
- 8) `accept_proposal` operation used to accept a set of catalog-agent proposals.
- 9) Catalog agents remove parts corresponding to accepted space-reduction proposal, creating new attribute space.
- 10) Feasibility check is performed on the new attribute space (steps 2) through 5)).
- 11) If the attribute space becomes empty, bid agent accepts another set of catalog-agent proposals (if none left, terminate with failure).
- 12) Else goto 6.

#### Solution Generation

- 13) Each catalog agent selects the part with the highest utility as a solution.
- 14) Terminate with success

Figure 11: ACDS Algorithm

ACDS uses *interval subsumption* to reduce the size of the search space. Consider the part set shown at the left of Figure 12. Suppose that this catalog agent receives information back from the constraint agents that the upper bound of attribute A1 must be tightened and the lower bound of attribute A2 must be tightened (increased in value). The catalog agent must propose a new catalog that makes the smallest possible change to its attribute space. Figure 12 shows the possible proposals: the agent can remove p3, generating the set labeled 1 in the

figure; or, it can remove p2 and p3, generating the set labeled 2 in the figure<sup>9</sup> Set 1 is the proposed set because it *subsumes* set 2.

**Definition 10:** A proposal p1 *subsumes* a proposal p2 if p1 minimally tightens the attribute space and p2 does not.

Set 1 is minimal because it reduces the upper bound of A1 by 3 and increases the lower bound of A2 by 2, while set 2 reduces the upper bound of A1 by 5 and increases the lower bound of A2 by 5. Interval subsumption saves computation because set 2 is not generated. It is considered implicitly when the proposal corresponding to set 1 is generated.

Part	A1	A2
p1	1	6
p2	3	3
p3	6	1

Set	A1	A2
1. p1, p2	[1 3]	[3 6]
2. p1	[1 1]	[6 6]

Figure 12: Interval Subsumption

The problem combinatorics are a result of the operation of accepting catalog-agent proposals, but the proposal acceptance heuristic is effective in reducing its effects. In addition, the generation of all possible accepted proposals can be done internally. Instead of generating all accepted proposals before notifying the catalog agents, the bid agent generates and accepts a single proposal. The bid agent generates the remainder of the possible accepted proposals while the network is exploring the first one. The bid agent hides the effects of the problem combinatorics through concurrency. In a large number of cases, these factors make the problem tractable.

## 4 Related Work

One of the first set-based design system is the Mechanical Design Compiler (MDC) [32]. The MDC uses a labeled-interval calculus (LIC) [12] to represent sets of designs and to make inferences about and eliminate sets of designs. ACDS uses constraints to concurrently identify sets of designs to eliminate to bring the current space of designs within the feasible space. In addition to the complexity reduction ACDS achieves by reasoning over sets of designs, its approach further reduces problem complexity through the constraint-based decomposition and algorithm. ACDS is also able to solve much larger designs (the MDC reported results for only a few small designs).

In configuration design systems, Cossack [10], VT [8], AIR-CYL [4], M1 [7], and GOPS [3] use a variant of the traditional *point-by-point* design process where a complete design is generated and modified to resolve constraint violations, or a partial design is generated and extended to resolve constraint violations. As a result, the complexity arguments in favor of the ACDS representation and approach hold in relation to each of these systems. ACDS handles the inherent combinatorics of the design process by a representation and algorithm that reduces the search space. These systems do not achieve the advantages of concurrency that ACDS does, where all constraints are applied to eliminate infeasible designs.

Several agent-based concurrent-engineering systems have been developed to help aid a team of designers throughout the product life cycle. Among these are PACT (a testbed for building large-scale CE systems) [33], DesignWorld [34], Galileo [35] and First-Link (cable

<sup>9</sup> Proposals are not generated in this way in reality.



harness configuration) [36] all based on the framework developed in Pan [37]. These systems also employ the point-by-point approach to create a design and assume that agents should primarily fill the role of assistants to human designers. The ACDS general model of design and the design process allows its electronic agents to take a more active role. In addition, a shared utility function provided by the designer provides the gradient necessary to search the design space in the most promising paths.

Distributed AI approaches are characterized by the exchange of local, partial information among a set of independent agents. Among these systems are multistage negotiation [38, 39], distributed, constrained heuristic search [40], TEAM (parametric design of steam condensers) [41, 42], DFI (steel-connection design) [43]. Agents in these systems are usually responsible for using their unique domain knowledge to both assign values to variables and evaluate constraints. By treating constraints as active problem-solving agents, ACDS defines all necessary communication paths *a priori*. In the constraint-based decomposition, agents minimize or eliminate the amount of information communicated. The evaluation of a constraint represents all possible conflicts that could arise among agents, and each constraint agent is given the problem-solving capability and knowledge to resolve those conflicts. ACDS represents partial or uncertain information explicitly in the form of intervals, and provides an explicit repository for all communication and conflict resolution in the form of constraints.

Finally, Bradley and Agogino [44, 45] present a system for single-part/single-function electronic-part selection. This system builds a model of designer preferences, which is used to select a part subject to a set of constraints. This is an iterative process, where a set of candidate solutions is identified, allowing the user to select or reject the set, or refine the preferences until an optimal selection is made. This approach is limited to selection of *single* components, whereas ACDS is able to solve the same problem over tens or hundreds of components. The ACDS shared utility function captures the designer preferences.

## 5 Experimental Results

ACDS has generated designs for single-board computer systems, milling machine configurations and the VT elevator configuration problem. This section presents some experimental results on the VT elevator configuration problem (for more information on the VT problem, go to <http://camis.stanford.edu/protege/> on the Mosaic server and click the Sisyphus-2 icon)<sup>10</sup>. A sequence of experiments were performed where catalog agents were incrementally added and assigned to a separate machine, along with their respective constraint agents, until each of the VT functions were implemented. In each problem instance, a solution was found without backtracking. Figure 13 shows the solution-space size and the agent population as catalog agents were added. In Figure 13(a), the number of solutions is plotted against the number of catalog agents, showing the exponential nature of the problem. Figure 13(b) shows the total number of agents as the number of catalog agents increases.

---

<sup>10</sup> Some VT constraints were not invertible and were omitted.

I

architecture, we are able to rapidly eliminate sets of provably infeasible designs. A novel algorithm is used to efficiently and concurrently identify and eliminate designs that lie outside the feasible region defined by the design constraints. A shared utility function ensures the uniformity of decisions among all agents. In addition to rapidly identifying sets of feasible designs and incorporating downstream concerns throughout the design process, ACDS manages the exponential complexity inherent to the class of design problems that it solves.

## 7 References

1. Darr, T.P. and W.P. Birmingham, Automated Design for Concurrent Engineering, *IEEE Expert*, 1994, To appear.
2. Darr, T.P. and W.P. Birmingham, Concurrent Engineering: An Automated Design-Space Exploration Approach, *Concurrent Engineering: Methodology and Applications*, P. Gu and A. Kusiak, Editors, 1993, Elsevier Science Publishers, New York, NY, p. 91-115.
3. Haworth, M.S., W.P. Birmingham, and D.E. Haworth, Optimal Part Selection, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1993, Vol. 12, No. 10, p. 1611-1617.
4. Brown, D.C. and B. Chandrasekaran, Knowledge and Control for a Mechanical Design Expert System, *IEEE Computer*, 1986, Vol. 19, No. 7, p. 92-100.
5. Balkany, A., W.P. Birmingham, and I.D. Tommelein, An Analysis of Several Design Tools, *AI EDAM*, 1993, Vol. 7, No. 1, p. 1-17.
6. Tommelein, I.D., *et al.*, SightPlan Experiments: Alternate Strategies for Site Layout Design, *ASCE Journal of Computing in Civil Engineering*, 1991, Vol. 5, No. 1, p. 42-63.
7. Gupta, A.P., W.P. Birmingham, and D.P. Siewiorek, Automating the Design of Computer Systems, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, April 1993, Vol. 12, No. 4, p. 473-487.
8. Marcus, S., J. Stout, and J. McDermott, VT: An Expert Elevator Designer That Uses Knowledge-based Backtracking, *AI Magazine*, 1988, Vol. 9, No. 1, p. 95-112.
9. Mittal, S., C.L. Dym, and M. Morjaria, PRIDE: An Expert System for the Design of Paper Handling Systems, *IEEE Computer*, 1986, Vol. 19, No. 7, p. 102-114.
10. Mittal, S. and F. Frayman, COSSACK: A Constraints-Based Expert System for Configuration Tasks, *Proceedings of the 2nd International Conference on Applications of AI to Engineering*, 1987, Boston, MA.
11. Fishburn, P.C., *Utility Theory and Decision Making*, Publications in Operations Research, Vol. 18, 1970, New York, John Wiley & Sons, Inc., p. 42-46.
12. Ward, A.C., T. Lozano-Perez, and W. Seering, Extending the Constraint Propagation of Intervals, *AI EDAM*, 1990, Vol. 4, No. 1, p. 47-54.
13. Hyvonen, E., Constraint reasoning based on interval arithmetic: the tolerance propagation approach, *Artificial Intelligence*, December 1992, Vol. 58, No. 1-3, p. 71-112.
14. Mackworth, A.K., Consistency in Networks of Relations, *Artificial Intelligence*, February 1977, Vol. 8, No. 1, p. 99-118.
15. Kumar, V., Algorithms for Constraint-Satisfaction Problems: A Survey, *AI Magazine*, Spring 1992, Vol. 13, No. 1, p. 32-44.

16. Davis, E., Constraint Propagation with Interval Labels, *Artificial Intelligence*, July 1987, Vol. 32, No. 3, p. 281-331.
17. Yokoo, M., *et al.*, Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving, *Twelfth International Conference on Distributed Computing Systems*, 1992, Yokohama, Japan, IEEE Computer Society, Los Alamitos, CA, p. 614-621.
18. Collin, Z., R. Dechter, and S. Katz, On the Feasibility of Distributed Constraint Satisfaction, *12th International Conference on Artificial Intelligence (IJCAI-91)*, 1991, Sydney, Australia, Morgan Kaufmann Publishers, San Mateo, CA, p. 318-324.
19. Mittal, S. and B. Falkenhainer, Dynamic Constraint Satisfaction Problems, *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, 1990, p. 25-32.
20. Bowen, J. and D. Bahler, Conditional Existence of Variables in Generalized Constraint Networks, *Ninth National Conference on Artificial Intelligence (AAAI-91)*, 1991, p. 215-220.
21. Bessiere, C., Arc-Consistency in Dynamic Constraint Satisfaction Problems, *Ninth National Conference on Artificial Intelligence (AAAI-91)*, 1991, AAAI Press/The MIT Press, p. 221-226.
22. Navinchandra, D. and J.R. Rinderle, Interval Approaches for Concurrent Evaluation of Design Constraints, *Concurrent Product and Process Design*, 1990, DE-vol. 21, 101-108.
23. Mackworth, A.K. and E.C. Freuder, The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence*, January 1985, Vol. 25, No. 1, p. 65-74.
24. Mohr, R. and T.C. Henderson, Arc and Path Consistency Revisited, *Artificial Intelligence*, March 1986, Vol. 28, No. 2, p. 225-233.
25. Deville, Y. and P. Van Hentenryck, An Efficient Arc Consistency Algorithm for a Class of CSP Problems, *12th International Joint Conference on Artificial Intelligence*, 1991, Sydney, Australia, Morgan Kaufmann Publishers, San Mateo, CA, p. 325-330.
26. Van Hentenryck, P., Y. Deville, and C.-M. Teng, A generic arc-consistency algorithm and its specializations, *Artificial Intelligence*, 1992, Vol. 57, No. 2-3, p. 291-321.
27. Faltings, B., Arc-consistency for continuous variables, *Artificial Intelligence*, February 1994, Vol. 65, No. 2, p. 363-376.
28. van Beek, P., On the Minimality and Decomposability of Constraint Networks, *Tenth National Conference on Artificial Intelligence (AAAI-92)*, 1992, p. 447-452.
29. Dechter, R., I. Meiri, and J. Pearl, Temporal constraint networks, *Artificial Intelligence*, May 1991, Vol. 49, No. 1-3, p. 61-95.
30. Balkany, A., W.P. Birmingham, and J.T. Runkel, Solving Sisyphus by Design, *International Journal of Human-Computer Studies*, 1994, Vol. 40, p. 221-241.
31. Bowen, J., P. O'Grady, and L. Smith, A constraint programming language for Life-Cycle Engineering, *Artificial Intelligence in Engineering*, October 1990, Vol. 5, No. 4, p. 206-220.
32. Ward, A.C. and W.P. Seering, Quantitative inference in a mechanical design compiler, *ASME Journal of Mechanical Design*, 1993, Vol. 115, No. 1, p. 29-35.
33. Cutkosky, M.R., *et al.*, PACT: An Experiment in Integrating Concurrent Engineering Systems, *IEEE Computer*, January 1993, Vol. 26, No. 1, p. 28-37.

34. Huyhn, P.N., M.R. Genesereth, and R. Letsinger, Automated Concurrent Engineering in Designworld, *IEEE Computer*, January 1993, Vol. 26, No. 1, p. 74-76.
35. Bowen, J. and D. Bahler, Supporting Multiple Perspectives: A constraint-based approach to concurrent engineering, *Artificial Intelligence in Design '92*, 1992, Kluwer Academic Publishers, p. 85-96.
36. Park, H., *et al.*, An agent-based approach to concurrent cable harness design, *AI EDAM*, 1994, Vol. 8, No. 1, p. 45-61.
37. Pan, J.Y.C. and J.M. Tenenbaum, An Intelligent Agent Framework for Enterprise Integration, *IEEE Transactions on Systems, Man, and Cybernetics*, 1991, Vol. 21, No. 6, p. 1391-1408.
38. Conry, S.E., *et al.*, Multistage Negotiation for Distributed Constraint Satisfaction, *IEEE Transactions on Systems, Man, and Cybernetics*, 1991, Vol. 21, No. 6, p. 1462-1477.
39. Lesser, V.R., A Retrospective of FA/C Distributed Problem Solving, *IEEE Transactions on Systems, Man, and Cybernetics*, 1991, Vol. 21, No. 6, p. 1347-1362.
40. Sycara, K., *et al.*, Distributed Constrained Heuristic Search, *IEEE Transactions on Systems, Man, and Cybernetics*, 1991, Vol. 21, No. 6, p. 1446-1461.
41. Lander, S.E. and V.R. Lesser, Negotiated Search: Organizing Cooperative Search Among Heterogeneous Expert Agents, *Proceedings of the Fifth International Symposium on Artificial Intelligence, Applications in Manufacturing and Robotics*, 1992, Cancun, Mexico, p. 351-358.
42. Lander, S.E. and V.R. Lesser, Understanding the Role of Negotiation in Distributed Search Among Heterogeneous Agents, *Proceedings of the 13th International Conference on Artificial Intelligence*, 1993, Cambery, France, p. 438-444.
43. Werkman, K.J., Multiple Agent Cooperative Design Evaluation Using Negotiation, *Artificial Intelligence in Design '94*, 1992, Kluwer Academic Publishers, p. 161-180.
44. Bradley, S.R. and A.M. Agogino, An Intelligent Real Time Design Methodology for Catalog Selection, *Design Theory and Methodology - ASME 1991*, 1991, p. 201-208.
45. Bradley, S.R. and A.M. Agogino, Computer-Assisted Catalog Selection With Multiple Objectives, *Design Theory and Methodology - ASME 1993*, 1993, p. 139-147.
46. Runkel, J.T. and W.P. Birmingham, Solving VT By Reuse, *Proceedings of the 8th Banff Knowledge-Acquisition for Knowledge-Based Systems Workshop*, 1994, Banff, Alberta, Canada, SRDG Publications, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, T2N 1N4, p. 42.1-42.28.