AN AGENT ARCHITECTURE AND ALGORITHM FOR SOLVING DISTRIBUTED CONFIGURATION-DESIGN PROBLEMS

Timothy P. Darr (timd@eecs.umich.edu)
http://ai.eecs.umich.edu/people/timd/timd.html

William P. Birmingham (wpb@eecs.umich.edu)
http://ai.eecs.umich.edu/people/wpb/home.html

Advanced Technologies Lab
The University of Michigan
Ann Arbor, MI 48109

Fax: (313)763-1260

1. INTRODUCTION

Configuration design is a type of design problem in which parts are selected from catalogs and connected to meet the following problem requirements: *functionality*, *specifications*, and *constraints* [7-9, 12]. Functionality defines what the design is supposed to do, specifications define optimality goals, and constraints define the feasibility relationships that must be satisfied for the design to operate correctly. A *design* is a collection of parts evaluated with respect to the problem requirements (functionality, specifications and constraints). A design that meets these requirements is a solution to the problem. Configuration design is a ubiquitous and economically important task, playing a prominent role in complex products such as automobiles, airplanes or computer systems that contain millions of parts, including resistors, light bulbs, screws, microprocessors and engines.

Parts are described by attributes and implement one or more functions. Configuration design is difficult because parts can implement many functions (the *multi-function part* problem) [7, 8], functions can be implemented by many parts, parts may depend on other parts for their correct operation (the *support function* problem) [9], and constraints and specifications defined over part attributes restrict the allowed configurations, introducing horizon effects. There exist several techniques to reduce the number of designs to explore. If the functionality requirement is decomposed into *required* functions that individual parts implement, designs are constructed by selecting parts that implement each required function. Heuristics based on design experience or properties of the design problem can be used to reduce the search space by ruling out certain part selections.

Large-scale configuration design problems that consist of thousands of required functions and millions of parts are too complex for a single agent[1] or small group of agents to manage. Such problems are recursively decomposed into sub-problems, until they are manageable. At this level, parts are selected to implement a small number of required functions, subject to constraints on the selected parts. However, many of these constraints are shared among sub-problems, the agents responsible for solving the sub-problems are often geographically distributed, and the catalogs from which parts are selected often reside outside the organization. Thus, solving large-scale problems requires communication among agents, and algorithms to resolve constraint violations on the shared constraints that link the sub-problems. Computer networks, such as the Internet, facilitate communication among agents making algorithms possible to resolve constraint violations, and provide a way to make the contents of part catalogs outside the design organization available to design agents.

This paper describes the Automated Configuration-Design Service (ACDS) [1-3], a system for solving large-scale configuration design problems using a network of design agents that

[1] In this work, an agent is a human or computer process possessing design knowledge with the capability to communicate with other agents. ACDS agents are computational processes that reside on their own computer host, communicating with other agents by passing messages.

represent part catalogs and design constraints. The ACDS algorithm is guaranteed to terminate, with a solution if one exists. ACDS agents communicate using design protocols and an attribute-space representation that bounds the space of *all* possible designs. ACDS agents concurrently shrink the space by applying local knowledge, properties of the domain, and heuristics, until a solution is found.

The contributions of this work are divided into three areas: (a) distributed design algorithms and networks, (b) distributed design representations, (c) distributed design agents, and (d) distributed configuration design problem representations. Each of these contributions is described briefly below, and in detail in the following chapters.

2. ACDS ALGORITHM AND NETWORK

The ACDS algorithm is executed by a network of computational design agents to solve configuration design problems. There are four basic ACDS agents: catalog agents, that represent part catalogs; constraint agents, that represent feasibility constraints; the system agent, that represents a user interface to the ACDS network; and the search-control agent, that monitors the algorithm to assist the network in recovering from dead-ends and to ensure that a solution will be found, if one exists. In an ACDS network, a communication link exists between a catalog agent and constraint agent if the feasibility constraint represented by the constraint agent restricts the selection of parts from the catalog represented by the catalog agent.

Figure 1 shows an ACDS network for an example elevator configuration design problem. In this example network, there are two catalog agents and two constraint agents. The catalog agents represent sets of motors and machine units. The machine-unit is an assembly that houses the motor that drives the elevator cab up and down the elevator shaft. The constraint agents represent feasibility constraints between the motor horsepower and machine-unit allowed horsepower, and a feasibility constraint specifying the maximum total weight. A solution to the problem is a selection of a motor and machine-unit satisfying the constraints that the horsepower value of the selected motor must lie within the minimum and maximum allowed horsepower values of the selected machine-unit, and that the total weight be no more than 2760 lbs. These constraints are represented by the expressions:

- machine-unit.min_hp ≤ motor.hp,
- motor.hp ≤ machine-unit.max_hp,
- machine-unit.weight + motor.weight ≤ 2760.

Figure 2 shows the set of parts for each catalog.

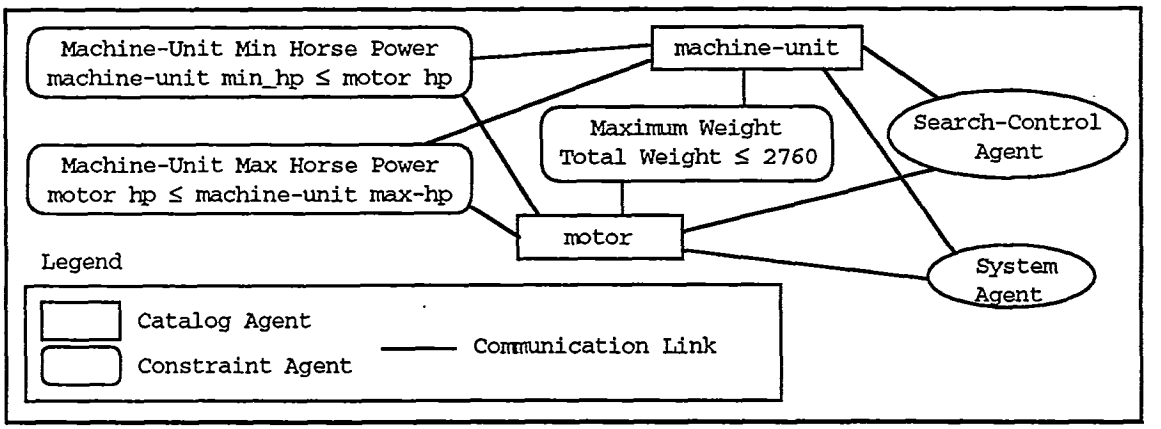

Figure 1: ACDS Network

The ACDS algorithm is based on the concurrent engineering (CE) design paradigm. CE transforms the design process from a serial to a parallel one by concurrently applying all relevant knowledge to the problem. The result of this method is the shortening of the product development

cycle. The ACDS algorithm and network of distributed agents achieves concurrency through *constraint-based decomposition.* In constraint-based decomposition, constraint agents define problem decompositions and use properties of the domain and heuristics to concurrently eliminate designs from consideration that do not satisfy their constraint.

part_name	min_hp	max_hp	weight
model18	10	15	1100
model28	15	20	1700
model38	20	40	2400
model58	40	40	2750

(a) machine-unit catalog

part_name	hp	weight
10HP	10	374
15HP	15	473
20HP	20	534
25HP	25	680
30HP	30	715

(b) motor catalog

Figure 2: Example Configuration Design Catalogs

The ACDS algorithm operates on *spaces* of designs. ACDS agents use operations to shrink this space until a *set* of solutions are found. The ACDS algorithm consists of *concurrent* catalog-agent operations to construct the space from the agent's individual parts, followed by *concurrent* constraint-agent operations to shrink the space, followed by *concurrent* catalog-agent operations to remove parts that lie outside the constraint-agent generated spaces. This cycle is repeated until a solution or set of solutions is found.

3. ACDS DESIGN REPRESENTATION

The solution to a configuration design problem is described as an assignment of values to a set of design attributes. In the example presented in Section 2, the attributes that describe the design are {motor horsepower, total-weight}. Thus, one possible design is {10, 1474} ({model18, 10HP}). ACDS represents the design as an *attribute space*. The *attribute-space representation* is a set of intervals that compactly represent the *space* of all possible designs [1-3]. In this representation, each design attribute is an *interval*. The complete set of intervals is an abstract, *m*-dimensional space, where *m* is the number of attributes that describe the design. As the following example shows, the attribute space representation has several desirable properties when applied to problems whose constraints are monotonic and part attributes can be partially ordered.

Figure 3 shows all possible designs for the example as points, and the attribute-space as a rectangle, given by {motor horsepower = [10 30], total-weight = [1474 3465]}, that encloses the points. Clearly, some of the designs in the attribute space exceed the weight constraint. The attribute-space representation is also used to represent the possible part selections from the catalogs shown in Figure 2. For example, the possible choices for the machine-unit catalog are given by the attribute space {min_hp = [10 40], max_hp = [15 40], weight = [1100 2750]}. The first advantage of the attribute-space representation is that it is a compact representation of a potentially large number of designs. Even though there are only a small number of possible configurations in Figure 3, the attribute space as shown encompasses *any* design within its boundaries.

The second advantage of the attribute-space representation is that it facilitates certain inferences. Assuming that the constraints are monotonic and the part attributes can be ordered, sets of parts that cannot be in any solution (inconsistent parts) can be identified and eliminated. Consider the total-weight constraint. Solving for each term yields the equivalent expressions:

- machine-unit weight $\leq$ 2760 - motor weight
- motor weight $\leq$ 2760 - machine-unit weight

These expressions restrict the allowed assignments to machine-unit weight and motor weight, respectively, given the set of assignments to the other attribute. Using the attribute-space representation for the weight attributes of each catalog, and interval arithmetic to evaluate the expressions, sets of *infeasible* weights can be eliminated. For example, [374 715] is the set of

possible assignments for the motor weight attribute. Inserting this value into the expression for machine-unit weight yields:

- machine-unit weight ≤ 2760 - [374 715]
  ≤ [2045 2386]

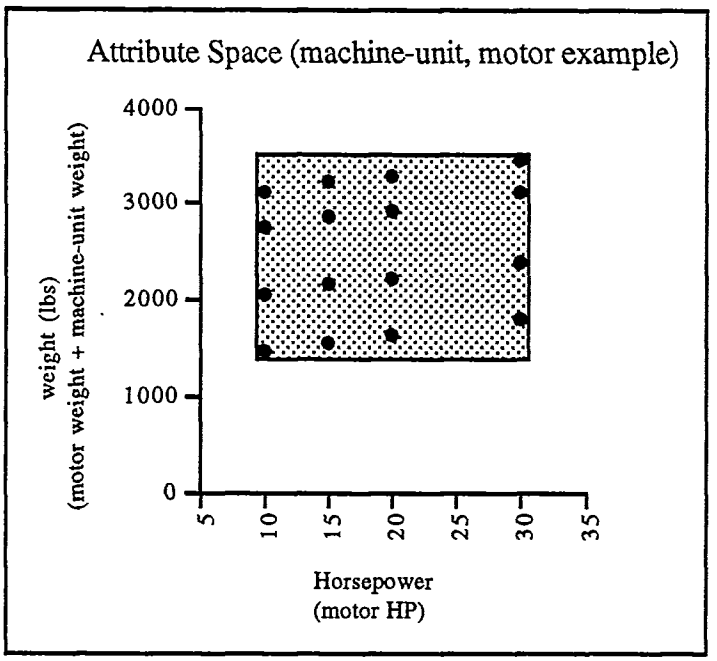

Figure 3: Example Attribute Space

This result indicates that given *any* possible motor selection, the most that the machine-unit can weigh is 2386 lbs. From this, the parts model38 and model58 can be eliminated from consideration without losing any solutions, since if either model38 or model58 is selected, there is no motor light enough to satisfy the total-weight constraint. Note that this inference was possible without knowing information about *specific* parts. In fact, any machine-unit whose weight is in the range 2386 < machine-unit weight ≤ 2750, which could be a large number of parts, is not consistent and should be removed. An attribute space that contains only consistent parts is a *consistent* attribute space [1-3, 10, 11, 16, 17].

Figure 4 shows the same catalogs in Figure 2 with inconsistent parts removed. Any configuration of the remaining parts satisfies the weight constraint, but there are still configurations that violate the constraint between the minimum and maximum allowed horsepower of the selected machine-unit and the horsepower of the selected motor (for example, {model18, 20HP}). The attribute-space representation suggests a heuristic for further eliminating parts to identify a set of solutions. An attribute space that contains only solutions is a *decomposable* space.

part_name	min_hp	max_hp	weight
model18	10	15	1100
model28	15	20	1700

(a) machine-unit catalog

part_name	hp	weight
10HP	10	374
15HP	15	473
20HP	20	534

(b) motor catalog

Figure 4: Catalogs without Inconsistent Parts

To illustrate the heuristic for finding a decomposable space, consider the constraints between the machine-unit minimum and maximum horsepower and the motor horsepower shown below:

- machine-unit.min_hp ≤ motor.hp,
- motor.hp ≤ machine-unit.max_hp,

Replacing the terms in the first expression with their interval values yields the expression:

- machine-unit.min_hp ≤ motor.hp,
- [10 15] ≤ [10 20]

In a decomposable space, *every* value in the interval on the left-hand side of this expression must be less than or equal to *every* value on the right-hand side. By examination, the space is not decomposable, since there exists an assignment that violates the constraint, namely machine-unit.min_hp = 15 and motor.hp = 10. Thus, to satisfy this constraint, an effective heuristic is to remove the machine-unit with value min_hp = 15 and the motor with value hp = 10. Removing infeasible designs using this heuristic moves the attribute-space toward a decomposable space.

4. ACDS DESIGN AGENTS

ACDS agents possess a set of operations to navigate attribute spaces. Catalog agent operations map from the catalog agent's parts, to an attribute space representation; and map from an attribute space to a set of parts that lie within that space. Constraint agent operations transform one attribute space into a smaller attribute space with certain properties. This section outlines the agent properties and operations.

Catalog agents have the capability to:

- identify parts that implement required functions,
- create an attribute-space representation of the parts in its catalog,
- communicate its attribute space to constraint agents to achieve design goals,
- apply an attribute space to its catalog to remove parts.

Catalog agents use operations to map from parts to the attribute-space representation and back. To illustrate, we return to the example presented in Section 2. The machine-unit catalog agent constructs the initial attribute-space representation by mapping from the parts shown in Figure 2 to the space {min_hp = [10 40], max_hp = [15 40], weight = [1100 2750]}, which bounds all possible part selections. To remove inconsistent parts identified by the consistency operation illustrated in Section 3, the machine-unit catalog agent maps from the constraint-agent generated space weight = [1100 2386] to the set of parts {model18, model28} by removing the parts {model38, model58} since their weights exceed 2386 lbs. Once the inconsistent parts are removed, the new machine-unit catalog attribute space is {min_hp = [10 15], max_hp = [15 20], weight = [1100 1700]}.

Constraint agents have capability to:

- construct a consistent attribute space,
- move an attribute space toward a decomposable space.

The constraint agents use operations to create a consistent space, and create a space that moves in the direction of a decomposable space. These operations are implemented using interval techniques described in Section 3. This section illustrates these operations graphically as *space-shrinking* operations.

Figure 5 illustrates the constraint-agent operations for the constraint between the machine-unit minimum horsepower and the motor horsepower. The initial attribute space for the constraint in Figure 5: is {motor hp = [10 30], machine-unit min_hp = [10 40]}.

Viewed as a projection onto the feasible space, there are *slices* along each dimension of the initial attribute space that do not intersect the feasible space. In particular, the slice along the machine-unit min_hp dimension just greater than 30 up to the value 40 does not intersect the feasible region at all, so any machine-unit whose minimum required horsepower lies within this

slice can be removed as they are not consistent. The *consistent* space {motor hp = [10 30], machine-unit min_hp = [10 30]} is shown. The updated space after the catalog-agent operations have been applied is {motor hp = [10 30], machine-unit min_hp = [10 20]}, shown as the space bounded by dotted lines. Even though this space intersects the feasible space along each dimension, it is not enclosed by the feasible space.

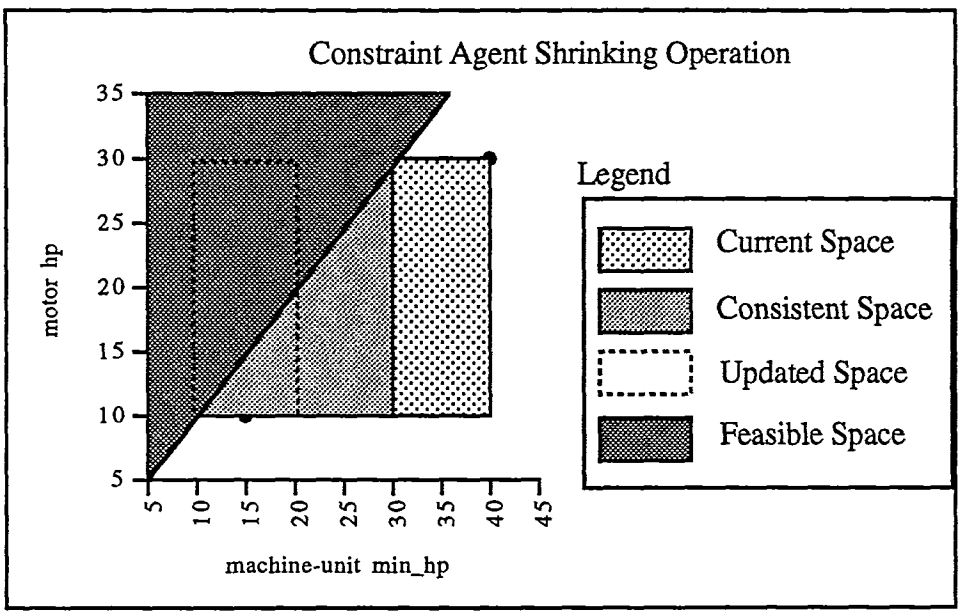

Figure 5: Constraint-Agent Shrinking Operations

To further shrink the space so that it is enclosed within the feasible space, a second constraint-agent operation using the space-shrinking heuristic described above is used. This operation directs the attribute space toward the feasible region by examining the bounds of the current attribute space. In this example, the corner of the current attribute space defined by {motor horsepower = 10, machine-unit min_hp = 20} lies outside the feasible space. To direct the space toward the feasible space, the bounds that define this corner are *tightened* by creating the space {motor hp = (10 30], machine-unit min_hp = [10 20)}, where the interval motor hp = (10 30] means the horsepower must be greater than 10 and less than or equal to 30.

The system agent represents the user interface to ACDS, and has the capability to:

- collect and broadcast design requirements,
- identify when the design process is complete.

The search-control agent monitors the design process, and has the capability to:

- detect dead-end conditions when backtracking is necessary,
- direct the agents to a new attribute space to explore,
- guarantee complete coverage,
- identify when no solution exists.

ACDS agents communicate through design protocols to achieve design goals. These protocols specify the format of messages sent and received, and the sequence of messages used within the protocol. For example, *consistency* and *feasibility* protocols allow catalog and constraint agents to achieve consistent and decomposable spaces by sending messages from the catalog agents to the constraint agents to define the current space. The constraint agents use this space to construct new spaces which are sent back to the catalog agents, who remove parts that lie outside the space.

5. ACDS DESIGN PROBLEM REPRESENTATION

ACDS includes a precise definition of the configuration design problem, which is mapped to a distributed, dynamic, multi-attribute domain, interval constraint-satisfaction problem (CSP) computational model [14, 15]. A CSP is a general problem-solving representation given by a set of variables, a set of domain values for the variables and a set of constraints that restrict the possible assignment of values to variables. The ACDS CSP computational model has the following additional properties:

1. the variables and constraints have preconditions that specify when they are active (dynamic CSP),
2. each variable or constraint is a separate computational process that communicate by sending messages (distributed CSP),
3. the domain elements are described by an attribute tuple (multi-attribute domain CSP),
4. constraints are evaluated over interval-valued variables (interval CSP).
5. Variable represent parts that cover the *same* set of functions.

The ACDS CSP computational model provides a precise framework for applying established CSP properties and heuristics to configuration design problems. For example, ACDS uses *arc-consistency* [4, 5, 16, 17] to reduce the number of combinations by efficiently removing certain configurations that violate constraints, and the *forward checking* heuristic to propagate the effects of design decisions [6]. By mapping required functions to CSP variables and design constraints and specifications to CSP constraints, the ACDS CSP framework encompasses all types of configuration design problems, including those with multi-function parts and support components. This work extends the class of problems to which a CSP model applies by defining the multi-attribute CSP, which is a CSP whose variable domain elements are described by an attribute tuple. In the multi-attribute CSP, parts map directly to CSP domain elements.

An additional property identified by the ACDS CSP computational model is the *boundary-part* property. This property applies to a class of configuration design problems in which the parts can be ordered for each part attribute and in which the constraints that are defined over the part attributes are monotone. This property states the conditions under which a single part can be selected to implement a function without losing any solutions. This property is used as a basis for the conditions for backtrack-free search, namely a boundary part implements each required function.

6. ACDS CONTRIBUTIONS

This section summarizes the contributions of this work and describes the experimental and theoretical results that support these contributions. The contributions of this work fall into three areas: distributed design algorithms and networks, distributed design problem representations, and distributed design agents.

6.1.1 Distributed Design Algorithm Contributions

An effective algorithm for large-scale distributed design is guaranteed to terminate, uses heuristics to manage the problem complexity and is scaleable. This work defines a distributed-design algorithm for solving configuration design problems that is guaranteed to terminate. The contributions are the following:

- ACDS Algorithm
  - Sound and complete,
  - Uses heuristics for particular design domains and classes of designs,
  - Scales to over 200 agents.

This work proves a theorem that states the properties of the search-control agent that must hold for the algorithm to terminate, with a solution if one exists. We show that for the VT elevator

configuration design problem [13], the space-shrinking heuristics reduce the time to solve the problem over binary-search methods. Finally, we demonstrate that the ACDS algorithm solves an elevator design problem when scaled to over 200 agents, which is a limit on the number of computer hosts available at the University of Michigan.

6.1.2 Design Problem Representation Contributions

To incorporate domain-independent heuristics and into design algorithms, it is necessary to have a precise problem definition and a domain-independent computational model. This work precisely defines the distributed configuration design problem and casts it as a dynamic, distributed, interval constraint-satisfaction problem (CSP). The contributions are the following:

- Precise definition of the distributed configuration design problem,
- The attribute-space representation.
  - Facilitates efficient reasoning for a large class of problems,
  - Effective for communicating large design spaces.
- CSP model for configuration design represents
  - Multi-function parts,
  - Support components,
  - Multi-function catalogs.
- CSP techniques reduce the search space using node- and arc-consistency.

This work defines the distributed configuration design problem, and maps it to a CSP computational model. Multi-function part, support component and multi-function catalog design problems are solved using ACDS to demonstrate that the CSP model can be used to solve these problems. Experiments are performed to demonstrate the effectiveness of node- and arc-consistency properties in solving design problems.

The multi-attribute domain CSP introduces unique properties that must be satisfied to extend the class of backtrack-free search problems. This work extends the class of problems to which the basic CSP representation applies to include problems with multi-attribute variable domain elements. The contributions are the following:

- Multi-attribute CSP:
  - Boundary element property reduces the number of domain elements to a single element without losing solutions,
  - Precise characterization of the conditions for backtrack-free search.

This work precisely defines the multi-attribute CSP, and the boundary element property, and proves the properties of the boundary element. The conditions for backtrack-free search are stated and a theorem proven.

6.1.3 Design Agent Contributions

To create flexible design networks for solving a variety of design tasks, design agents must be able to interact with other agents in well-defined ways to achieve joint design goals. This work describes a set of generic design agents at the knowledge level, and a set of design protocols these agents use to achieve specific design goals. The contributions are the following:

- Knowledge-level description of agent capabilities, design message classes, agent operations for distributed design.
- Protocols to achieve specific design goals.

These agents and protocols are specialized for configuration design and incorporated into the ACDS algorithm to demonstrate their effectiveness in solving distributed configuration design problems.

7. REFERENCES

1. Darr, T.P. and W.P. Birmingham, Automated Design for Concurrent Engineering, *IEEE Expert*, 1994, Vol. 9, No. 5, p. 35-42.

2. Darr, T.P. and W.P. Birmingham, An Attribute-Space Representation and Algorithm for Concurrent Engineering, *AI EDAM*, 1996, Vol. 10, No. 1, p. 21-36.

3. Darr, T.P., Ph.D. thesis in preparation, 1996, The University of Michigan,

4. Mackworth, A.K., Consistency in Networks of Relations, *Artificial Intelligence*, February 1977, Vol. 8, No. 1, p. 99-118.

5. Freuder, E.C., Synthesizing Constraint Expressions, *Communications of the ACM*, 1978, Vol. 21, No. 11, p. 958-966.

6. Haralick, R.M. and G.L. Elliott, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence*, 1980, Vol. 14, No. 3, p. 263-313.

7. Mittal, S. and F. Frayman, Towards a generic model of configuration tasks, *Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, 1989, Detroit, MI, Morgan Kaufmann, p. 1395-1401.

8. 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.

9. 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.

10. Hyvönen, E., Constraint reasoning based on interval arithmetic: the tolerance propagation approach, *Artificial Intelligence*, December 1992, Vol. 58, No. 1-3, p. 71-112.

11. Benhamou, F., D. McAllester, and P. Van Hentenryck, CLP(Intervals) Revisited, 1994, Brown University Technical Report CS-94-18.

12. Birmingham, W.P., A.P. Gupta, and D.P. Siewiorek, *Automating the Design of Computer Systems*, 1992, Jones and Bartlett Publishers.

13. 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.

14. 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.

15. Mittal, S. and B. Falkenhainer, Dynamic Constraint Satisfaction Problems, *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, 1990, p. 25-32.

16. Carlson, B., S. Haridi, and S. Janson, AKL(FD) - A concurrent language for FD programming, *Logic Programming: 1994 International Symposium*, 1994, The MIT Press, p. 521-535.

17. Carlson, B., M. Carlsson, and D. Diaz, Entailment of Finite Domain Constraints, *Proceedings of the Eleventh International Conference of Logic Programming*, 1994, MIT Press.