Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2000), *14*, 39–51. Printed in the USA.
Copyright © 2000 Cambridge University Press 0890-0604/00 $12.50

Part-selection triptych: A representation, problem properties and problem definition, and problem-solving method

TIMOTHY P. DARR[1] AND WILLIAM P. BIRMINGHAM[2]
Artificial Intelligence Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, 1101 Beal Avenue, Ann Arbor, MI 48109

(RECEIVED September 1, 1998; ACCEPTED September 8, 1999)

Abstract

In *part-selection* problems, *parts* are selected from *catalogs* and connected to meet the following problem requirements: *functionality*, *specifications*, and *constraints*. This paper formally defines the part-selection problem, enumerates a set of design properties that are useful during a search for a design solution, and provides an algorithm for solving part-selection problems based on a novel set of operators for manipulating portions of the design space.

Keywords: Part selection; Constraint-satisfaction; Design; Optimization; Configuration

1. INTRODUCTION

Many products today are designed using "components off the shelf" (COTS). These products can range from sophisticated computer systems, to aircraft subsystems, to software systems, to integrated circuits (e.g., "intellectual property" modules), and even to buildings. With the proliferation of "electronic" catalogs, we expect that increasingly more products will be designed with COTS.

In general, designing with COTS is an example of a commonly occurring, fundamental class of engineering-design problems called the *part-selection problem*. In these problems, parts are selected from catalogs and connected to meet the following problem requirements: *functionality, specifications, and constraints*. Functionality defines what the artifact is supposed to do; specifications define optimality conditions; and, constraints define the feasibility relationships that must be satisfied for the artifact to operate correctly. An artifact that satisfies these requirements is a solution to the design problem.

Reprint requests to: Timothy P. Darr.
[1]Now at Trilogy Development Group, 6034 West Courtyard Drive, Austin, TX, 78730.
[2]All editorial decisions regarding this paper were made by the Editor Emeritus, Clive Dym.

In contrast with configuration (Darr & Dym, 1997), the part-selection problem does not include part arrangement. As such, part selection is a subset of the more general configuration problem. Even though, as we show in this paper, part selection is a very difficult modeling and computational problem. The results given in this paper apply directly to configuration problems, since configuration requires part selection.

In this paper, we aim to do the following things:

- Provide a new and comprehensive formal representation for part-selection problems that extends previous (related) problem representations, yet is compact with well-defined semantics.
- Describe several important properties about part-selection problems and solutions. These properties, combined with our representation, help to uncover structure in the problem that can be exploited to create heuristics. An example of this is the "boundary" part, defined later in this paper, which eliminates search during problem solving. Further, the representation provides a basis to rigorously compare various problem-solving approaches to the part-selection problem.
- Provide a new solution method that effectively exploits our novel "attribute-space" representation. This solution method suggests a family of efficient problem solvers.

The problem representation we give is motivated by many things, one of which is combinatorics: part-selection problems are characterized by brutal combinatorics. We can roughly estimate the number of possible solutions as the following:

$$|\text{parts}|^{|\text{functions}|}, \tag{1}$$

where |functions| is the number of functions needed to meet the functionality requirements in the design and |parts| is the average number of parts that can fulfill a function. We assume here that a part can fulfill only one function, since this makes the calculation easier and has little affect on the asymptotic results we are using here for illustration. Note, however, that later in the paper we describe how to handle parts that can perform multiple functions. We make the simplifying assumption that the time to find a solution to the part-selection problem is at worst case proportional to the size of the design space. Thus, we can bound computation time as $\mathcal{O}(|\text{parts}|^{|\text{functions}|})$.

Needless to say, we are very concerned with mitigating this combinatorial problem. Thus, our representation is biased to help with the average case. Given that it is impossible to reduce the number of functions (this is generally accepted as a problem input[3]), we choose to try to reduce the base (i.e., the number of parts). We do this by creating a representation that is based on "attribute spaces," where we form an abstraction over individual parts, aggregate their values first as catalogs, and then as an interval with only the upper and lower bounds for the values of attributes represented.

For example, imagine all the parts in a part-selection problem are described by two attributes, cost and mass. We would first represent each part by two intervals: one for cost and one for mass. We would then create two intervals for each catalog: one for cost and one for mass. Thus, we can compactly represent each catalog, regardless of the number of parts it contains, as two intervals.

In the very best case, this can drastically reduce the problem combinatorics: we have now effectively reduced |parts| to two (2) for each attribute: the upper and lower bounds on an interval. There are some strict limitations on this approach, which we describe formally and informally in the paper. Thus, for a simple design with a single attribute (but many functions), we reduce problem size, and hence computational effort, to

$$2^{|\text{functions}|}. \tag{2}$$

Some might argue that there is little comfort in Eq. (2). We would agree, except to say that two is much better than |parts|. This represents, in an informal way, an approximation on the lower bound of the worst case. Clearly, part selection remains a tough problem.

[3]One cannot generally reduce the number of functions. This would result, for example, in the user of the artifact having to make due with fewer functions; this is not acceptable, in our view.

As discussed in this paper, the interval representation provides some important additional benefits, and we have crafted a design process that exploits intervals to gain computation advantage. More importantly, intervals have exposed a very useful heuristic called "boundary parts." If boundary parts are present in all catalogs, it is possible to solve the part-selection problem without search (Darr, 1997; Darr et al., 1998). The design process we discuss is based on manipulating these intervals. We start with a representation of the entire design space, and then reduce it through a series of (nondeterministic) operators.

The problem representation given in this paper, while attempting to combat combinatorics, is also expressive. Our representation is based on the *attribute*. This notion is then systematically extended to parts and catalogs. Further, we also describe a *multifaceted* representation of constraints. We believe, as we have described elsewhere (Darr et al., 1998), that driving the part-selection process *via* constraints has computational advantages.

In the remainder of the paper, we provide our part-selection triptych as follows: we begin with basic definitions leading to the attribute-space representation, which we use to compactly represent parts and catalogs. We then give a formal definition of important properties of the part-selection problem and a formal problem definition. This is followed by a nondeterministic method for solving these problems based on "shrinking" the attribute space. We then summarize the paper.

2. BASIC DEFINITIONS

Part selection has a distinguishing characteristic that the problem is solved by selecting predefined parts that can be connected only in certain ways (Mittal & Frayman, 1989; Rahmer & Voss, 1998) to perform some high-level functionality (Colton & Ouellette, 1993). In other words, new parts cannot be created at will, constraints restrict the ways that parts can be connected, and parts implement low-level functions whose combination results in the desired high-level functionality (Mittal & Frayman, 1989; Kota & Lee, 1993; Pimmler & Eppinger, 1994). Functions have been variously defined as "what a device is for" (de Kleer & Brown, 1984) or "a description of behavior abstracted by humans through recognition of the behavior in order to utilize it" (Umeda et al., 1990). Here, a function is a property of a part that, alone or in combination with other parts, achieves some user-defined functionality. We assume that the mapping from the functions that a part implements to the higher level functionality is given as part of the definition of a part. The design is the collection of parts that implements the user-defined functionality, among other things.

Functions help to manage the complexity of the design through functional decompositions, whereby the user-defined functionality is decomposed into functions implemented by individual parts (Colton & Ouellette, 1993; Gupta et al., 1993; Kota & Lee, 1993; Pimmler & Eppinger, 1994). For

part name	hp	max current	sheave diameter	efficiency	pfm
model18-10	10	150	24	0.81	machine-unit, motor
model18-15	15	250	24	0.81	machine-unit, motor
model28-15	15	250	30	0.76	machine-unit, motor
model28-20	20	260	30	0.76	machine-unit, motor
model38-20	20	260	30	0.76	machine-unit, motor
model38-25	25	340	30	0.76	machine-unit, motor
model38-30	30	440	30	0.76	machine-unit, motor
model38-40	40	530	30	0.76	machine-unit, motor
model58-40	40	530	32	0.842	machine-unit, motor
model18	-	-	24	0.81	machine-unit
model28	-	-	30	0.76	machine-unit
model38	-	-	30	0.76	machine-unit
model58	-	-	32	0.842	machine-unit
motor10HP	10	150	-	-	motor
motor15HP	15	250	-	-	motor
motor20HP	20	260	-	-	motor
motor25HP	25	340	-	-	motor
motor30HP	30	440	-	-	motor
motor40HP	40	530	-	-	motor

Fig. 1. Motor-housing catalog.

example, a high-level function in the automotive industry is "power generation and transmission," which can be decomposed into the functions "engine" and "transmission." The "engine" function can be further decomposed into the functions "combustion," "fuel delivery," etc. Eventually, the decomposition halts when the functions can be implemented by sets of parts. In current methods, the part is the irreducible, fundamental entity that makes up a design. The design is constructed by selecting and connecting parts, and then verifying that the design satisfies all constraints.

Yet, the problem definition in this paper uses the *attribute* as the fundamental entity to describe parts, functions, and designs. In fact, a function is nothing more than an attribute with a particular name: the denotation *via* a name for a function. While functions are prominent in part-selection problems, they are not the only things considered. Many other aspects—attributes—of a design are important, such as cost, size, etc.

2.1. Attributes, parts, catalogs, designs

In this section, we develop the definitions (semantics) for parts, catalogs, and designs based on the *attribute*. A set of attributes represents a part; a set of parts form a catalog, which can also be considered a set of attributes; parts and catalogs are partitions in the space of attributes. Thus, we have a simple, uniform representation for parts and catalogs, with a straightforward, yet expressive semantics.

An *attribute* is a two-tuple $\langle a_k, d_k \rangle$, where a_k is the attribute's name and d_k is its domain. The domain is the set of values $\alpha \in d_k$ that can be assigned to the attribute. The domain d_k of a part attribute a_k is restricted to scalar values. In this definition, there are two types of domains: numeric-valued domains consisting of integer or real numbers; and domains that consist of tokens. An attribute domain is ordered if for each $\alpha, \beta \in d_k$, either $\alpha < \beta$, or $\alpha > \beta$, or $\alpha = \beta$. In general, numeric-valued domains are ordered, and symbolic domains are not. $A = \{a_i : i = 1, N\}$ is the set of all attributes.

Parts are sets of attributes, part $= \{\langle a_1, d_1 \rangle, \ldots, \langle a_m, d_m \rangle, \langle f_1, fm_1 \rangle, \ldots, \langle f_n, fm_n \rangle\}$.

A special subset of part attributes $\{\langle f_1, fm_1 \rangle, \ldots, \langle f_n, fm_n \rangle\}$ is the *part-function multiplicity* (part$^{\text{pfm}}$), which is the set of functions implemented by the part; the part implements fm_k instances of f_k (Mittal & Frayman 1989; Colton & Ouellette, 1993; Haworth et al., 1993). As an example, consider Figure 1, which is a catalog of parts. Each part in this catalog implements a set of functions; part *model18-10* has part$^{\text{pfm}} = \{\langle$machine-unit, 1$\rangle \langle$motor, 1$\rangle\}$.

A *catalog* is a collection of parts, catalog $= \{\text{part}_i : i = 1, \ldots, D\}$ generally organized by vendor product line, although this need not be the case. The set of all catalogs is catalogs $= \{\text{catalog}_i : i = 1, \ldots, M\}$.

The *catalog attributes*, catalog$^{\text{attr}} = \{\langle a_k, \cup_i d_{i,k} \rangle : \langle a_k, d_{i,k} \rangle \in \text{part}_i \in \text{catalog}\}$, is the set of attributes that define the parts in the catalog.[4] The domain of each catalog attribute a_k is the union of the domains $d_{i,k}$ for all parts in the catalog. The *catalog-function multiplicity*, catalog$^{\text{pfm}} = \{\text{part}_i^{\text{pfm}} : i = 1, \ldots, D\}$, is the set of part-function multiplicities of all parts in the catalog. The *problem-function multiplicity* is the set of all part-function multiplicities, $\{\text{catalog}_i^{\text{pfm}} : i = 1, \ldots, M\}$. As we discussed in the Introduction, there is a computational advantage to aggregating information about parts as attributes of a catalog as a whole.

Figure 1 shows a portion of a motor-housing catalog from the VT elevator problem (Yost & Rothenfluh, 1996). This

[4]The catalog attributes are not necessarily disjoint. The same attribute a_k can appear in more than one catalog. For example, the power attribute may appear in many different catalogs, since it is an attribute of many different parts.

catalog contains three types of parts: parts that implement the motor and machine-unit functions, parts that implement the machine-unit function, and parts that implement the motor function. The part model18-10 is defined by the set {⟨part-name, model18-10⟩, ⟨hp, 10⟩, ⟨max current, 150⟩, ⟨sheave diameter, 24⟩, ⟨efficiency, 0.81⟩, ⟨machine-unit, 1⟩, ⟨motor, 1⟩}. The set of catalog attributes is motor-housing^{attr} = {⟨hp, {10, 15, 20, 25, 30, 40}⟩, ⟨max current, {150, 250, 260, 340, 440, 530}⟩, ⟨sheave diameter, {24, 30, 32}⟩, ⟨efficiency, {0.81, 0.76, 0.842}⟩} (note that not every part in the catalog has a value for each of these attributes). The catalog-function multiplicity is the set motor-housing^{pfm} = {{⟨machine-unit, 1⟩, ⟨motor, 1⟩}, {⟨machine-unit, 1⟩}, {⟨motor, 1⟩}}.

A design is the result of a design process. In part selection, the design is represented as a collection of parts that is evaluated with respect to constraints and preferences (defined in Section 5). Formally, a design is represented as an assignment of attribute values from their domains: design = $\{\langle a_k, \alpha\rangle : \alpha \in d_k, k = 1, \ldots, N\}$.[5] The *design space* is the set of all possible assignments of values to attributes in the design, or the cross product of all attribute domains: $DS = \times_{k=1,\ldots,N}\langle a_k, d_k\rangle$.[6] In other words, the design space is the set of all possible designs, (design$_k \in DS$, for all k).

In the design process we describe later, the notion of design space is important. Our design process, essentially, starts with the entire design space, and through a set of operations refines that space into a design that meets the user's requirements.

The attribute as a fundamental representational entity is consistent with definitions that use the part as the basic element to describe a design (Mittal & Frayman, 1987, 1989; Marcus et al., 1988; Gupta et al., 1993), or optimization techniques that use state and decision variables (Lyon & Mistree, 1985; Bradley & Agogino, 1993; Mistree et al., 1994). Defining the problem with attributes makes it more convenient to use constraints, rather than parts, to direct the search for a solution. Design properties derived from the constraints are used to eliminate parts that ordinarily might be chosen to implement some function, without regard to whether that part violates a constraint or set of constraints. Furthermore, attributes allow us greater representational uniformity, as we can use them to represent not only parts, but ports (Mittal & Frayman, 1989), state variables, and many other things.

2.2. Constraints

In part selection, constraints can be used for two purposes: *evaluation* and *propagation* (Freuder, 1978; Sussman & Steele, 1980; Davis, 1987). Constraints define a network that describes the relationships among all the parts in the design. A *constraint* is a relation c_j defined over a subset of the attributes $A_j = \{a_k : a_k \in A\}$ that restrict the attribute domains: $c_j \subseteq DS$. The set of attributes, A_j, is the constraint's *arguments*.

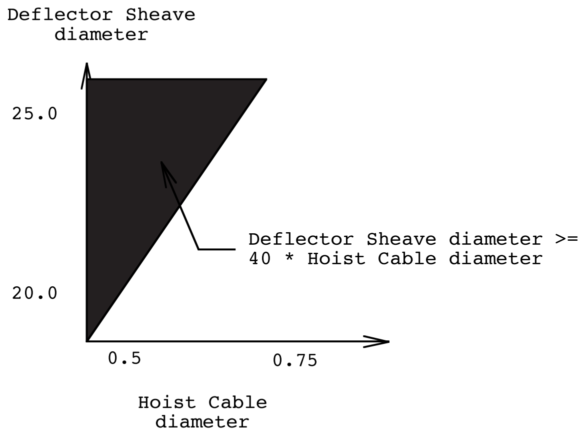

Fig. 2. Example constraint.

Both evaluation and propagation can be used to "drive" the design process. For example, our design process is based on satisfying constraints through a process of making a constraint network consistent and decomposable, as described in Section 4. This is done using constraints to both "propagate" and "evaluate" various assignments to attributes.

Figure 2 shows a constraint from the VT elevator-design problem that restricts the diameters of an elevator deflector sheave and hoist cable (Deflector Sheave diameter $\geq 40 *$ Hoist Cable diameter). Designs that do not satisfy this constraint are not physically possible, violating either a law of physics or some other inviolable relationship. The set of designs that satisfy the constraint are shown in the shaded area (Figure 2).

In our problem definition, constraints are defined over attribute spaces (see Section 3 for a definition), and are not defined over catalogs or parts. Constraints consist of the following elements:

- a *constraint-evaluation function* that determines when the constraint is satisfied,
- a *constraint-propagation function* that infers the value of one attribute given the values of other attributes, and
- a *precondition-evaluation function* that defines the conditions under which the constraint is to be evaluated.

The definition of constraints presented here includes explicit definitions of constraint evaluation and propagation functions.

2.2.1. *Constraint evaluation*

In evaluation, constraints determine if the current assignment of values to design attributes is physically allowed, or *feasible*. The constraint-evaluation function $g_j(A_j)$ maps from an attribute assignment $\{\langle a_k, \alpha\rangle : \alpha \in d_k, \text{ for all } a_k \in A_j\}$ to the Boolean values T or F. If $g_j(A_j) = T$, then the assignment *satisfies* the constraint (the assignment is *feasible*); if

[5]The names of parts can be included in these attributes.
[6]The cross product is defined as the set of assignments $\{\{\langle a_1, \alpha_1\rangle, \ldots, \langle a_N, \alpha_N\rangle\} : \alpha_i \in d_i$, for all a_i in the design}.

$g_j(A_j) = F$, then the assignment *violates* the constraint (the assignment is *infeasible)*.

2.2.2. *Constraint propagation*

In propagation, constraints infer the values of assigned or unassigned design attributes, given the values of assigned design attributes, potentially restricting future assignments. Propagation is an important way of using constraints in design, since it detects the infeasibility of a partial assignment (i.e., some attributes may not have final values) before all design attributes are assigned values. This reduces *thrashing*, where a partial assignment is pursued, even though it will eventually be found to be infeasible (Mackworth, 1977).

The constraint-propagation function $h_{j,i}(A_j)$ restricts the domain of a_i, given an assignment of domain values to all other attributes in A_j except a_i.[7] This function maps from an assignment of domain values to all but one of the constraint's arguments $\{\langle a_k, \alpha\rangle: \alpha \in d_k$, for all $a_k \in A_j, k \neq i\}$ to a subset of domain values, with properties defined in Section 4, for the remaining constraint argument: $h_{j,i}(A_j) \subseteq d_i$. There is a constraint-propagation function for each constraint argument a_i.

2.2.3. *Constraint preconditions*

A difficulty that arises in part-selection problems is that the set of constraints necessary to evaluate the design changes as parts are selected (Mittal & Frayman, 1987, 1989; Mittal & Falkenhainer, 1990; Bowen & Bahler, 1991). Thus, a precondition provides an explicit specification of when the constraint is to be evaluated. Our preconditions are similar to dynamic-constraint preconditions (Mittal & Falkenhainer, 1990).

A *precondition* is a relation c_j^{pre} defined over a subset of attributes $A_j^{pre} \subseteq A_j$ that define when a constraint c_j is active ($c_j^{pre} \subseteq DS$). The set of attributes, A_j^{pre}, are the precondition's arguments. The precondition-evaluation function $g_j^{pre}(A_j^{pre})$ maps from an attribute assignment to the Boolean values T or F. If $g_j^{pre}(A_j^{pre}) = T$, then the assignment satisfies the precondition and c_j is *active*; if $g_j^{pre}(A_j^{pre}) = F$, then the assignment violates the precondition and c_j is *inactive*.

Note that we can get a syntactically simpler constraint form with equivalent expressiveness by simply conjoining the constraint precondition with the constraint propagation function. We argue, however, that separating these elements of a constraint, as shown here, makes it easier to understand how a constraint is used.

2.2.4. *State variables*

In many part-selection problems, constraints are defined over attributes that are not catalog attributes, but which are derived from catalog attributes. For example, a designer is often interested in the total weight of a design, which is the sum of the weights of individual parts. These attributes are called *state variables*. The value of a state variable is computed using a function $f(SV)$.

Formally, a state variable, $\langle sv_k, d_k\rangle \in A$, is an attribute whose value is determined by some function $f_k(SV_k)$, where SV_k is the subset of attributes (catalog attributes and other state variables) that determines the value.

An example state variable in the VT domain is the releveling torque of the motor, defined by the expression: releveling torque = torque factor $*$ machine-unit sheave diameter, where torque factor is another state variable and machine-unit sheave diameter is a *part attribute*.

2.2.5. *Constraint properties*

Monotonic constraints (Hyvonen, 1992) are a class of constraints over numbers that are important for defining a useful set of design properties (Section 4). A monotonic constraint totally orders the possible assignments to an attribute from its domain. These constraints allow the use of interval arithmetic for constraint evaluation and propagation functions, and help to identify heuristics. In particular, if a constraint is monotonic, then we can use *endpoint analysis* (Darr, 1997; Darr et al., 1998) to evaluate only the upper and lower bounds of an interval, and deduce whether it is feasible or not, rather than evaluating all the points comprising the interval. This can be a tremendous advantage, particularly in finding infeasible designs, since a catalog many contain thousands of parts and we need only consider (in best case) two parts: those parts that define the upper ($\alpha_{\max}$) and lower (α_{min}) bounds of the interval.

Constraint c_j is *monotone decreasing* in a_k if for each assignment $\langle a_k, \beta\rangle \cup \{\langle a_i, \alpha\rangle$: for all $a_i \in A_j$ $(i \neq k)\}$ such that $g_j(A_j) = T$, then $g_j(A_j) = T$ for the assignment $\langle a_k, \gamma\rangle \cup \{\langle a_i, \alpha\rangle$: for all $a_i \in A_j$ $(i \neq k)\}$ for all $\gamma < \beta$. Constraint c_j is *monotone increasing* in a_k if for each assignment $\langle a_k, \beta\rangle \cup \{\langle a_i, \alpha\rangle$: for all $a_i \in A_j$ $(i \neq k)\}$ such that $g_j(A_j) = T$, then $g_j(A_j) = T$ for the assignment $\langle a_k, \gamma\rangle \cup \{\langle a_i, \alpha\rangle$: for all $a_i \in A_j$ $(i \neq k)\}$ for all $\gamma > \beta$. Constraint c_j is *monotonic* if for each $\langle a_k, d_k\rangle \in A_j$, the constraint is monotone increasing in a_k or monotone decreasing in a_k.

In other words, a constraint is monotone increasing if the domains of its arguments can be totally ordered (i.e., $\gamma > \beta$) and the constraint evaluation function is T for all domain elements greater than some limit (e.g., β). For monotone decreasing constraints, the idea is the same, only we are concerned with domain elements below some limit.

3. THE ATTRIBUTE-SPACE REPRESENTATION

Ordered attributes are represented as an *attribute space*,[8] which is an interval that bounds the values of all domain

[7] In the propagation function $h_{j,i}(A_j)$, the first subscript (j) denotes the constraint, and the second subscript (i) denotes the attribute.

[8] The attribute-space representation of an attribute is denoted using **bold** font.

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

Fig. 3. Example part catalogs for a *machine unit* and a *motor*.

elements (Davis, 1987; Navinchandra & Rinderle, 1990; Hyvonen, 1992; Finch & Ward, 1995; Van Hentenryck & Michel, 1995). The attribute space is an abstraction that allows us to compactly represent the potentially very large design space. In addition, it focuses on attributes, rather than parts, which is consistent with typical constraint definitions.

The attribute-space representation of an ordered domain attribute a_k is given by the interval: $\boldsymbol{d_k} = [\alpha_{\min}\alpha_{\max}]$, where $\alpha_{\min} \leq \alpha \leq \alpha_{\max}$, for all α, $\alpha_{\min}$, $\alpha_{\max} \in d_k$. If $\alpha_{\min} \leq \alpha \leq \alpha_{\max}$, then we say that $\alpha \in \boldsymbol{d_k}$. The attribute-space representation of the design space is the set of intervals $\boldsymbol{AS} = \{\langle a_k, \boldsymbol{d_k}\rangle : k = 1, \ldots, N\}$ within which all designs in the design space lie. The attribute space is said to contain all designs in the design space: $\boldsymbol{AS}$ contains design$_i$ iff for all $\langle a_k, \alpha\rangle \in$ design$_i$, $\alpha \in \boldsymbol{d_k}$, $\langle a_k, \boldsymbol{d_k}\rangle \in \boldsymbol{AS}$.

Figure 3 shows two part catalogs. In Figure 4 the attribute-space representation for all part combinations, which are enumerated in Figure 5, is given. Figure 3 shows how compactly the entire design space can be represented; the shaded box covers all designs. This box can be represented simply by the intervals for weight [1474 3115] and hp [10 30].

The traditional representation used in part-selection problems is a *point representation*, where constraints are used to evaluate a single design. When used, propagation functions restrict the assignment of values to unassigned design attributes, relative to the current assigned design attributes. The design is then extended by selecting and assigning additional design attributes, evaluating the design, propagating values, and so on. The advantage of the attribute-space representation is that it allows propagation functions to be applied to all design attributes, whether assigned or not. Thus, a commitment to a specific value of any design attribute is postponed until the last possible moment.

The efficiency of reasoning over intervals has been demonstrated outside the design domain in the constraint-satisfaction problem (CSP) literature (Davis, 1987; Navinchandra & Rinderle, 1990; Dechter et al., 1991; Hyvonen, 1992; Faltings, 1994) and the constraint-logic programming (CLP) literature (Benhamou et al., 1994; Van Hentenryck & Michel, 1995). Some CLP systems use a representation similar to the attribute-space representation, representing discrete choices as intervals (Van Hentenryck & Michel, 1995).

Any subset of attributes can also be represented as an attribute space. The *catalog attribute space* of a catalog is the set of intervals ***catalog*** = $\{\langle a_k, \boldsymbol{d_k}\rangle$: for all $\langle a_k, \boldsymbol{d_k}\rangle \in$ catalog$^{\text{attr}}\}$ within which all the parts in the catalog lie. The catalog attribute space is said to contain each part in the catalog: ***catalog*** contains part$_i$ iff for all $\langle a_k, \alpha\rangle \in$ part$_i$; $\alpha \in \boldsymbol{d_k}$, where $\langle a_k, \boldsymbol{d_k}\rangle \in$ ***catalog***. This is a useful property for pruning parts from a catalog if $\boldsymbol{d_k}$ defines the valid range of values. If $\alpha \notin \boldsymbol{d_k}$, then that part cannot be in a valid design.

A constraint can also be defined over the attribute-space representation of the constraint arguments, which we denote as $\boldsymbol{c_j}$. Intuitively, if $\boldsymbol{g_j}(A_j) = T$, then each assignment contained in the attribute space $\boldsymbol{A_j}$ satisfies the constraint (the attribute space $\boldsymbol{A_j}$ is a feasible space). Also, if $\boldsymbol{g_j}(A_j) = F$, then there is at least one assignment in the space $\boldsymbol{A_j}$ that violates the constraint (the space $\boldsymbol{A_j}$ is infeasible). In

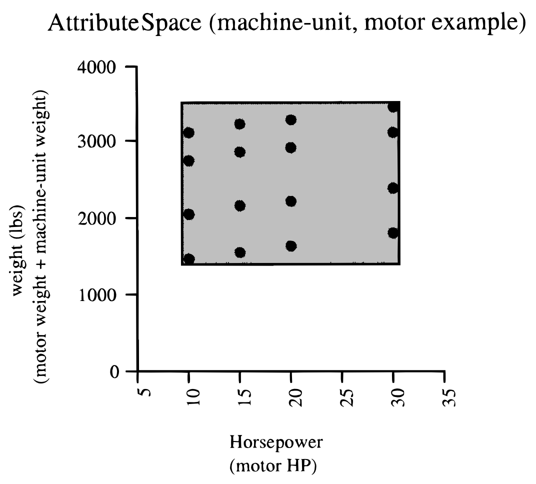

Fig. 4. Attribute-space representation.

design:
{<machine-unit-part-name, model18>, <motor-part-name, 10HP>,
<horsepower, 10 hp>, <total-weight, 1474 lbs>}

design space:
{{<machine-unit-part-name, model18>, <motor-part-name, 10HP>,
<horsepower, 10 hp>, <total-weight, 1474 lbs>},
{<machine-unit-part-name, model28>, <motor-part-name, 10HP>,
<horsepower, 10 hp>, <total-weight, 2074 lbs>},
...,
{<machine-unit-part-name, model58>, <motor-part-name, 40HP>,
<horsepower, 40 hp>, <total-weight, 3740 lbs>}}

attribute space:
[10 hp 40 hp], [1474 lbs. 3740 lbs.]

Fig. 5. Enumerated (point representation) design space.

other words, constraint $\boldsymbol{c_j}$ is satisfied if all designs in the design space are feasible with respect to $\boldsymbol{c_j}$. The constraint-propagation function $\boldsymbol{h_{j,i}}(A_j)$ maps to an interval $\boldsymbol{d_i^*} \subseteq \boldsymbol{d_i}$ with properties defined in Section 4.

Preconditions can also be defined over an attribute space $(\boldsymbol{c_j^{pre}})$. Intuitively, if $\boldsymbol{g_j^{pre}}(A_j^{\mathrm{pre}}) = T$, then there is at least one assignment in $\boldsymbol{A_j^{pre}}$ that satisfies the precondition and $\boldsymbol{c_j}$ is active. Also, if $\boldsymbol{g_j^{pre}}(A_j^{\mathrm{pre}}) = F$, then there is no assignment contained in $\boldsymbol{A_j^{pre}}$ that satisfies the precondition and $\boldsymbol{c_j}$ is inactive.

4. PROBLEM PROPERTIES

Defining design properties gives us structure to exploit in problem solving. As we have shown, constraints are central to the part-selection problem. Thus, we look to the constraint-satisfaction (CSP) literature to get important properties of constraint networks that we can use for design. This section defines properties of constraints, attribute spaces, parts, and catalogs that are motivated by the CSP literature. A CSP is a class of problems described by a set of variables, a set of domain values for each variable, and a set of constraints. A solution is an assignment of values to variables that satisfies the constraints. There are many parallels between CSPs and part-selection problems (Darr et al., 1998), some of which we enumerate here.

Consistency and *decomposability* are properties of a CSP (Mackworth 1977; Dechter et al., 1991; van Beek 1992*a*). Consistency eliminates all provably infeasible assignments to variables in a CSP (although some infeasible assignment may still remain in the context of the entire design); consistency allows infeasible designs to be eliminated. Decomposability is a condition of a CSP where all possible assignments to a variable are guaranteed to result in feasible solutions; decomposability is used to identify sets of legal designs, from which the final, possibly optimal, solution is chosen.

A constraint whose precondition is true is *consistent* if, for each attribute in the constraint's arguments and every assignment to that attribute, there exist values for all other attributes that satisfy the constraints (Freuder, 1978). A constraint whose precondition is false is trivially consistent. Consistency allows elimination of sets of provably infeasible designs, thereby reducing the search space.

Formally, c_j is consistent if $g_j^{pre}(A_j^{pre}) = T$ and for each assignment $\langle a_k, \alpha\rangle$, for all $\alpha \in d_k$, $a_k \in A_j$, there exist at least one assignment $\langle a_i, \beta\rangle$, for all $\beta \in d_i$, $a_i \in A_j$ (for all $i \neq k$) such that $g_j(A_j) = T$. If c_j is consistent, then each $a_k \in A_j$ is consistent with respect to c_j, and each $\alpha \in d_k$ is a *consistent value*. The predicate consistent (c_j) is true if c_j is consistent. The propagation function $h_{j,i}^{\mathrm{consistent}}(A_j) = d_i$ is defined such that a_i is consistent with respect to c_j.

A constraint defined over an attribute space $(\boldsymbol{c_j})$ is consistent (Davis, 1987; Navinchandra & Rinderle, 1990; Hyvonen, 1992; van Beek, 1992*a*; Faltings, 1994; Van Hentenryck & Michel, 1995) if $\boldsymbol{g_j^{pre}}(A_j^{pre}) = T$ and for all $a_k \in A_j$

- If c_j is monotone decreasing in a_k, $g_j(A_j) = T$ for the assignment $A_j^* = \langle a_k, \alpha_{\max}\rangle \cup \{\langle a_h, \beta\rangle$: for all $a_h \in A_j(h \neq k)\}$, where
  - $\beta = \alpha_{\min}$ if c_j is monotone decreasing in a_h, or,
  - $\beta = \alpha_{\max}$ if c_j is monotone increasing in a_h.
- If c_j is monotone increasing in a_k, $g_j(A_j) = T$ for the assignment $A_j^* = \langle a_k, \alpha_{min}\rangle \cup \{\langle a_h, \beta\rangle$: for all $a_h \in A_j(h \neq k)\}$, where
  - $\beta = \alpha_{\min}$ if c_j is monotone decreasing in a_h, or,
  - $\beta = \alpha_{\max}$ if c_j is monotone increasing in a_h.

For example, consider the constraint and the parts: machine-unit weight + motor weight $\leq$ 2760 lbs. Assume for this example that the attribute space is composed of the following: ⟨machine-unit weight, [1100 2750]⟩ and ⟨motor weight, [374 715]⟩. The constraint-propagation function for the machine-unit weight attribute is given by: machine-unit weight $\leq$ 2760 − motor weight. Evaluating this function using interval arithmetic (i.e., the endpoints of the intervals) yields the consistent range of values [1100 2386] for the machine-unit weight attribute. This calculation, therefore, relies only on evaluating the interval endpoints.

The predicate consistent $(\boldsymbol{c_j})$ is true if $\boldsymbol{c_j}$ is consistent. If $\boldsymbol{c_j}$ is consistent, then each $a_k \in A_j$ is consistent with respect to c_j, and each $\alpha \in \boldsymbol{d_k}$ is a consistent value. The propagation function $\boldsymbol{h_{j,i}^{\mathrm{consistent}}}(A_j) = \boldsymbol{d_i^*}$ is defined such that a_i is consistent with respect to c_j.

Consistency also applies to catalogs, design spaces, and attribute spaces.

- A catalog is a *consistent catalog* if for each $a_k \in$ catalog$^{\mathrm{attr}}$, and for every c_j or $\boldsymbol{c_j}$ such that $a_k \in A_j$, c_j or $\boldsymbol{c_j}$ is consistent. If catalog$_i$ is a consistent catalog, then each part $\in$ catalog$_i$ is a *consistent part*. The predicate consistent (catalog) is true if the catalog is consistent.
- A design space is a *consistent design space* if each c_j is consistent. The predicate consistent(DS) is true if DS is consistent. An attribute space is a *consistent attribute space* if each $\boldsymbol{c_j}$ is consistent. The predicate consistent$(\boldsymbol{AS})$ is true if $\boldsymbol{AS}$ is consistent.

A stronger property than consistency is *decomposability*. Decomposability means that all designs in the design space or attribute space satisfy all the constraints whose preconditions are true[9] (Mackworth & Freuder, 1985; van Beek, 1992*a*). Decomposability is defined differently depending on whether constraints are defined over design spaces or attribute spaces.

A constraint c_j is decomposable if $g_j^{pre}(A_j^{pre}) = T$ and $g_j(A_j^*) = T$ for each $A_j^* \in DS$. The predicate decomposable(c_j) is true if c_j is decomposable. A constraint $\boldsymbol{c_j}$ defined over an attribute space is decomposable if

[9] In the CSP literature, decomposability is often called n-consistency.

$\boldsymbol{g_j^{pre}}(A_j^{pre}) = T$ and $\boldsymbol{g_j}(A_j) = T$ for the assignment $\{\langle a_k, \beta\rangle$: for all $a_k \in A_j\}$, where

- $\beta = \alpha_{\max}$ if c_j is monotone decreasing in a_k, or,
- $\beta = \alpha_{\min}$ if c_j is monotone increasing in a_k.

The predicate decomposable$(\boldsymbol{c_j})$ is true if $\boldsymbol{c_j}$ is decomposable. If c_j or $\boldsymbol{c_j}$ is decomposable, then each $a_k \in A_j$ is decomposable with respect to c_j or $\boldsymbol{c_j}$, respectively.

Decomposability also applies to catalogs, design spaces, and attribute spaces. A catalog is a *decomposable catalog* if for each $a_k \in \text{catalog}^{\text{attr}}$, and for every c_j or $\boldsymbol{c_j}$ such that $a_k \in A_j$, c_j or $\boldsymbol{c_j}$ is decomposable. If catalog$_i$ is a decomposable catalog, then each part $\in$ catalog$_i$ is a *decomposable part*. The predicate decomposable(catalog$_i$) is true if catalog$_i$ is decomposable. DS is a *decomposable design space* if each c_j or $\boldsymbol{c_j}$ is decomposable. The predicate decomposable(DS) is true if DS is decomposable. $\boldsymbol{AS}$ is a *decomposable space* if each $\boldsymbol{c_j}$ is decomposable. The predicate decomposable$(\boldsymbol{AS})$ is true if $\boldsymbol{AS}$ is decomposable.

4.1. Comments on limitations and computational issues

The definitions for consistency and decomposability of an attribute space presented here assume that each constraint is monotonic and defined over ordered attributes, since only the interval endpoints are used in defining the property (Dechter et al., 1991; Hyvonen, 1992; van Beek, 1992*b*). The technique of using intervals to reason about constraints defined over discrete-value domains has been effectively used in constraint-logic programming (Benhamou et al., 1994; Carlson et al., 1994; Van Hentenryck & Michel, 1995). For nonmonotonic constraints or unordered attributes, or both, the computationally expensive process of checking each combination of values determines consistency or decomposability. Thus, a disadvantage of the attribute-space representation is that it does not apply to all constraints; however, its advantage is potentially significant improvements to computationally efficiency.

Consistency is achieved in polynomial time. Decomposability, however, is achieved in the worst case in exponential time. For a certain class of problems, decomposability can be efficiently achieved when the constraints have certain properties. In particular, it has been shown that for a certain class of CSPs, where the constraints are monotonic and the variable domain values are single valued, decomposability can be achieved in linear time (Van Hentenryck et al., 1992). Since many constraints in part-selection problems are monotonic, taking advantage of this property is worthwhile.

4.2. The boundary-part property

We can gain significant computational advantage by making use of the *boundary-part property*. This property guarantees that a boundary part is decomposable, regardless of the parts in other catalogs. Thus, it reduces the search space, and defines the conditions for backtrack-free search.

Catalog A	reliability	cost
Part 1	3	1
Part 2	1	3
Part 3	1	2

Fig. 6. Example boundary part.

Formally, part $\in$ catalog$_i$ is a boundary part iff for each $a_k \in \text{catalog}_i^{\text{attr}}(\langle a_k, d_k\rangle \in \text{part})$, and each c_j such that $a_k \in A_j$:

- if c_j is monotone decreasing in a_k, then $\langle a_k, \alpha_{\min}\rangle \in$ part.
- if c_j is monotone increasing in a_k, then $\langle a_k, \alpha_{\max}\rangle \in$ part.

Boundary parts commonly occur in catalogs since manufacturers generally differentiate the parts they produce and sell. An example part catalog is given in Figure 6. Assuming the constraint on cost is monotone decreasing and the constraint on reliability is monotone increasing, then Part 1 is a boundary part. This is because Part 1 has the least value in the catalog for cost and the greatest value for reliability.

5. PROBLEM DEFINITION

The design problem specifies the problem to be solved, and includes the functions to implement, the constraints to satisfy, the set of catalogs, a set of *fixed attributes*, and *designer preferences* over various designs, represented generally as a utility function [note: in this paper, for ease of explanation we use a linear, weighted value function (Fishburn, 1970)]. Thus, functions play a role in specifying the behavior of the design. As we define below, the design is complete when parts are selected to implement all functions.

Formally, the *design-problem* is a tuple <required-functions, constraints, catalogs, $u()$, fixed-attributes>, that describes the desired design in terms of a set of specifications:

- required-functions = $\{\langle f_k, fm_k\rangle$: fm_k instances of f_k are required$\}$.
  - f_k^{ϕ} is the ϕth instance of f_k ($\phi \in \{1, \ldots, fm_k\}$).
- constraints $= \cup_j c_j$, is the set of feasibility constraints, with preconditions $c_j^{pre}(A_j^{pre})$.
  - constraints$_{\text{support-fn}} \subseteq$ constraints is a set of *support-function* constraints.
- catalogs $= \cup_i$ catalog$_i$, is a set of part catalogs.
- $u()$ is a utility function representing the preferences of the designer. For this paper, we limit ourselves to lin-

```
precondition:
    elevator capacity <= 2500 lbs
    elevator platform width <= 84 inches
    elevator platform depth <= 60 inches

constraint:
    platform part = P2.5B
```

Fig. 7. Example constraint illustrating fixed attributes, unordered domains, and equality relationships.

ear, weighted *value* function that assigns a real number to a design or part, defines at least a partial order on the designs or parts, and represents the desirability of the part or design from the designer's perspective. Note, however, that we do not depend on an analytical utility function.

- $u(\text{part}) = \Sigma_k w_k * \upsilon(a_k), \Sigma_k w_k = 1, \langle a_k, d_k\rangle \in \text{part}$.[10]
- $u(\text{design}) = \Sigma_k w_k * \upsilon(a_k), \Sigma_k w_k = 1, \langle a_k, d_k\rangle \in \text{design}$.

- fixed-attributes $\subseteq A$, is a set of constant-valued attributes such that the value of each attribute domain cannot be changed during the design process. These values can be scalars or intervals.

Example fixed attributes from the VT domain include the building dimensions. An example value function is $u() = 0.3 * \upsilon(\text{HP}) + 0.7 * \upsilon(\text{total-weight})$. Figure 7 shows an example VT constraint representing an equality relationship defined over unordered attributes that uses fixed attributes in the precondition. This constraint guides the selection of an elevator platform dependent on the fixed attributes elevator capacity, elevator-platform width, and elevator-platform depth. If the elevator width is less than or equal to 84 inches, the elevator depth is less than or equal to 60 inches, and the elevator capacity is less than 2500 lbs., then the part P2.5B must be selected to implement the platform function.

Support-function constraints represent support function relationships, such as the requirement that a timer function be implemented for a particular CPU to operate (Mittal & Frayman, 1987, 1989; Mittal & Falkenhainer, 1990; Gupta et al., 1993; Haworth et al., 1993). By definition, support functions are functions that are not included in the list of required functions specified by the designer, but are necessary for the correct operation of some required function or other support function. These constraints have a precondition that defines when the support functions are required. The support functions are contained in the constraint expression.

[10] $\upsilon(a_k)$ is a normalizing function that assigns a value from 0 (worst) to 1 (best) to an attribute value for a_k.

```
Precondition
    Comp Cables Required > 0
Constraint
    compensation-cable quantity >= Comp Cables Required
```

Fig. 8. Example support function constraint.

Formally, c_j is a support-function constraint if there exists an $a_k \in A_j$ such that for all part $\in$ catalog$_i$, where $\langle a_k, d_k\rangle \in$ part (catalog$_i \in$ catalogs), there is an $\langle f_k, fm_k\rangle \in \text{part}^{\text{pfm}}$ such that $\langle f_k, fm_k\rangle \neq$ required-functions.

Figure 8 shows an example support-function constraint for the compensation-cable function.[11] The precondition specifies the number of required compensation cables, which is included as part of the problem definition. The constraint expression restricts the number of selected compensation cables (compensation-cable quantity) to be at least the number of required compensation cables (Comp Cables Required) included in the problem specification. If the precondition is not satisfied, the constraint is inactive and the compensation-cable function is not needed. If the precondition is satisfied, however, the compensation cable function is required.

A *solution* to the design problem is a set of parts $S = \{\text{part: part} \in \text{catalog}_i, i \in \{1, \ldots, M\}\}$ with the following properties:

- For each $f_k^{\phi} \in$ required-functions $\cup$ support-functions, there exists a part $\in S$ such that $\langle f_k, fm_k\rangle \in \text{part}^{\text{pfm}}$, for all part $\in S$.
- The attribute space (***AS***) that contains S is decomposable.
- For each $f_k^{\phi} \in$ required-functions $\cup$ support-functions, and each part$_i$, part$_j \in$ ***AS***, such that $f_k^{\phi} \in \text{part}_i^{\text{pfm}}$, and $f_k^{\phi} \in \text{part}_j^{\text{pfm}}$, $u(part_i) \le u(part_j)$. In other words, the parts that are chosen for the solution have the highest utility given all the parts that are admitted by the decomposable attribute space.

In multifunction part selection, the design topology is allowed to change depending on the set of parts selected. For example, inclusion of multifunction parts tends to decrease the number of parts in the final design, while parts that require support functions tend to increase the number of parts. In single-function part selection, the number of parts is fixed, since each part implements exactly one required function and there are no support-function relationships.

6. PROBLEM-SOLVING METHOD

Using the attribute-space representation, the properties defined in Section 4 are achieved by a series of *space-transformation operations* that transform an initial attribute

[11] It is assumed that the compensation cable function is not part of the problem definition.

space ($\boldsymbol{AS^0}$) into a consistent or decomposable attribute space. We assume here that these operators are nondeterministic, meaning that they will always make the correct choice when faced with a decision.

These operations create a consistent space from an inconsistent space ($\delta^{\text{consistent}}(\boldsymbol{AS^t}) \rightarrow \boldsymbol{AS^{t+1}}$), or shrink a consistent space ($\delta^{\text{shrink}}(\boldsymbol{AS^t}) \rightarrow \boldsymbol{AS^{t+1}}$).[12] The shrinking operation δ^{shrink} creates a space $\boldsymbol{AS^{t+1}}$ that lies within $\boldsymbol{AS^t}$. Embedded within these transformations are mapping operations from a discrete space consisting of parts, to an attribute space consisting of intervals. A similar technique has been used in constraint-logic programming (Benhamou et al., 1994; Carlson et al., 1994; Van Hentenryck & Michel, 1995) and part selection in the mechanical-engineering domain (Finch & Ward, 1995).

- design operation $\delta^{\text{consistent}}(\boldsymbol{AS^t})$ is a transformation from $\boldsymbol{AS^t}$ to $\boldsymbol{AS^{t+1}}$, where $\neg$consistent($\boldsymbol{AS^t}$) and consistent($\boldsymbol{AS^{t+1}}$).
- design operation $\delta^{\text{shrink}}(\boldsymbol{AS^t})$ is a transformation from $\boldsymbol{AS^t}$ to $\boldsymbol{AS^{t+1}}$, where consistent($\boldsymbol{AS^t}$) and $\boldsymbol{AS^{t+1}}$ is an attribute space where $\boldsymbol{d_k^{t+1}} \subseteq \boldsymbol{d_k^t}$, for all $\langle \boldsymbol{a_k^t}, \boldsymbol{d_k^t} \rangle \in \boldsymbol{AS^t}$, and for all $\langle \boldsymbol{a_k^{t+1}}, \boldsymbol{d_k^{t+1}} \rangle \in \boldsymbol{AS^{t+1}}$, and at least one $\boldsymbol{d_k^{t+1}} \subset \boldsymbol{d_k^t}$.

A decomposable space is achieved by combining these operations to alternate between creating consistent spaces from inconsistent spaces and shrinking consistent spaces:

$$\delta^{\text{consistent}}(\boldsymbol{AS^0}) \rightarrow \boldsymbol{AS^1};$$

$$\delta^{\text{shrink}}(\boldsymbol{AS^1}) \rightarrow \boldsymbol{AS^2};$$

$$\ldots$$

$$\delta^{\text{consistent}}(\boldsymbol{AS^{t-2}}) \rightarrow \boldsymbol{AS^{t-1}};$$

$$\delta^{\text{shrink}}(\boldsymbol{AS^{t-1}}) \rightarrow \boldsymbol{AS^t} \qquad \text{Algorithm (1)}$$

where decomposable($\boldsymbol{AS^t}$) and $\neg$decomposable($\boldsymbol{AS^k}$), $k = 0, \ldots, t-1$. Note that if any operation fails (e.g., there is no feasible solution), the algorithm halts.

This sequence of operations is driven by the constraints: the consistency operations remove inconsistent attribute-domain elements, thereby reducing the search space, and the shrinking operations use knowledge based on the type of constraints present in the design problem, or domain-dependent knowledge to shrink the attribute space by removing designs. Once decomposable($\boldsymbol{AS^t}$), a design can be easily generated by simply choosing parts from each catalog (which can be done concurrently). Parts should be chosen to be consistent with definitions in Section 5 (i.e., they should maximize the utility function for each catalog). This will yield a locally optimal solution, which may coincide with a globally optimal solution (but this is not guaranteed).

While consistency can be found *deterministically* without backtracking (with some restrictions on the problem), decomposability cannot always be found in such a way. Thus, in a deterministic implementation of Algorithm 1, there will be backtracking, except in either lucky or trivial cases. In fact, in the worst case the problem becomes exponential in the number of parts; this, however, is the bane of *all* part-selection problem solvers (as we discussed in the Introduction).

[12]Note, the use of a superscript here indicates that the design process unrolls over time.

6.1. Example problem

In this section, we present an example of the problem-solving method by solving the following *design-problem*:

$$\langle\{\langle\text{machine-unit, 1}\rangle\ \langle\text{motor, 1}\rangle\}, \{c_1, c_2\}, \{\text{machine-unit, motor}\}, u(), -\rangle, \text{ where}$$

- Required functions: there must be one each of *machine-unit* and *motor*.
- The constraints are as follows (we list only the propagation functions):

c_1: (min_hp, max_hp, hp): min_hp <= hp <= max_hp

$h_{1,1}$(min_hp, max_hp, hp): min_hp <= hp

$h_{1,2}$(min_hp, max_hp, hp): hp <= max_hp

c_2: (MU-weight, Motor-weight):

$$\sum(\text{MU-weight,Motor-weight}) <= 2760$$

$h_{2,1}$(MU-weight, Motor-weight):

MU-weight <= 2760 − Motor-weight

$h_{2,2}$(MU-weight, Motor-weight):

Motor-weight <= 2760 − MU-weight

- The catalogs are given in Figure 3:

$$\begin{aligned}\text{machine-unit}^{\text{attr}} &= \{\langle\text{min\_hp}, \{10, 15, 20, 40\}\rangle,\\ &\qquad \langle\text{max\_hp}, \{15, 20, 40\}\rangle,\\ &\qquad \langle\text{MU-weight}, \{1100, 1700, 2400, 2750\}\rangle\}\\ \text{motor}^{\text{attr}} &= \{\langle\text{hp}, \{10, 15, 20, 25, 30\}\rangle,\\ &\qquad \langle\text{Motor-weight}, \{374, 473, 534, 680, 715\}\rangle\}\\ \text{machine-unit}^{\text{pfm}} &= \{\langle\text{machine-unit}, 1\rangle\}\\ \text{motor}^{\text{pfm}} &= \{\langle\text{motor}, 1\rangle\}\end{aligned}$$

- The utility function is defined as:

$$U() = -[0.3(\text{hp}) + 0.7(\text{Motor-weight} + \text{MU-weight})],$$

where the closer to zero the better, and no scaling functions are given (for simplicity of explanation)

- There are no fixed attributes for this problem.

At the start of the design process—the application of the problem-solving method—$\boldsymbol{AS^0}$ is given by Figure 4. The first step of Algorithm 1 is the *consistent* operation; for this example, we will guide the shrinking operation by using the constraint propagation functions (we assume that the preconditions for both constraints are T). Consider that, if we want to make the catalogs consistent, we can use the propagation functions to infer bounds on the catalogs as follows:

$h_{1,1}()$: the lower bound on hp is 10.

$h_{1,2}()$: the upper bound on hp is 40.

$h_{2,1}()$: the upper bound on weight for the motor unit (MU) is 2386.

$h_{2,2}()$: the upper bound on weight for the motor is 1660.

Given this information, we apply $\delta^{\text{consistent}}(\boldsymbol{AS^0}) \rightarrow \boldsymbol{AS^1}$, where $\boldsymbol{AS^1}$ is given in Figure 9. Because the weight of the part model38 lies outside the consistent range, it is removed from the machine-unit catalog. Once this part is removed, however, the consistent space must be recalculated because the consistency of parts in the motor catalog may depend on model38. This is the case in this example, since model38 defined the upper bound on horsepower (max_hp for model38 is 40, the maximum value in the catalog). So, removing the part lowers the bound for propagation function $h_{1,2}()$ to 20, which causes the parts 25HP and 30HP to be removed from the motor catalog. At this point, the space is consistent, with the part catalogs as shown in Figure 10.

We can now chose a decomposable attribute space for the design; we do this operation nondeterministically, noting that there are alternative attribute spaces that could have been selected, and that search would probably be necessary in a deterministic search process. We, therefore, apply $\delta^{\text{shrink}}(\boldsymbol{AS^1}) \rightarrow \boldsymbol{AS^2}$, where decomposable$(\boldsymbol{AS^2})$. During the application of this operator, the motors with the least and greatest horsepower were removed, making c_1 decomposable. The part catalogs corresponding to $\boldsymbol{AS^2}$ are given in Figure 11.

The solution to the design problem, therefore, is $S = \{\text{model18, 15HP}\}$. The machine-unit part Model18 was chosen over Model28 because it weighs less giving it a better utility value. It is important to note that given $\boldsymbol{AS^2}$, S is optimal, but this is not necessarily the globally optimal design. The decision to form the decomposable space as we did caused us to miss the globally optimal design. This is not as bad a problem as it may at first seem, as we can choose other decomposable spaces if necessary to find the optimal solution (in a deterministic implementation); for a nondeterministic implementation, we can simply change the operator to chose the optimal decomposable space.

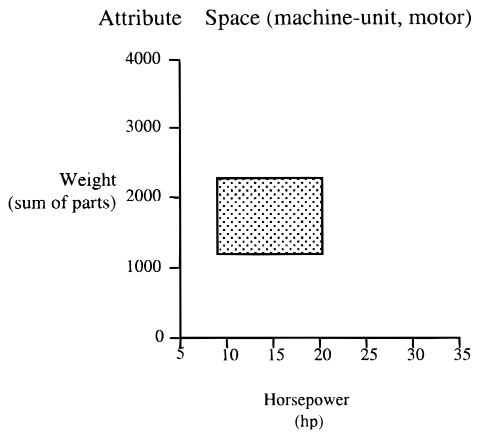

Fig. 9. $\boldsymbol{AS^1}$ depiction.

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

Fig. 10. Part catalogs resulting from application of $\delta^{\text{consistent}}(\boldsymbol{AS^1})$.

7. SUMMARY

This paper defined the part-selection problem, including definitions of parts, catalogs, constraints, and the problem specification. The representation is descriptive and accounts for combinatorial issues inherent in the class of part-selection problems. We have provided a multifacted definition of constraints, which we argue is needed to fully exploit the way constraints are actually used in problem solvers. As we have noted in the paper, our representation builds on the established work in the design and CSP fields, and attempts to unify many concepts from both fields. A formal description of part-selection problems helps to understand the nature of the problem, leading to both efficient algorithms, and rigorous comparisons among competing problem-solving methods. The VT-Sisyphus experiments (Schreiber & Birmingham, 1996) are an example where a formal problem representation provided many insights into design problem solving.

We defined the properties of “consistency” and “decomposability” for part-selection problems, borrowing con-

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

(a) machine-unit catalog

part_name	hp	weight
15HP	15	473

(b) motor catalog

Fig. 11. Final part catalogs, which form the decomposable attribute space.

cepts from the CSP literature. We also introduced the "boundary-part property," which can be used to define the conditions for backtrack-free search. These properties are useful for design problem solvers for part-selection problems. While not all problem solvers will try to achieve these properties, they are nonetheless useful in describing certain problem regularities.

The design properties are used in a describing the operation of a nondeterministic algorithm for solving part-selection problems. This algorithm exploits a set of operators that shrink the design space, finding a series of consistent, albeit "smaller," design spaces. The algorithm terminates with a decomposable design space, or recognition that no such space is possible. This algorithm, which cannot be directly implemented, yields a variety of algorithms that can be implemented.

ACKNOWLEDGMENTS

This research was funded in part by the DARPA ACORN project under subcontract 1-41480 and by General Motors Corp. This paper does not necessarily reflect the views of the funding agencies. We also thank Joe D'Ambrosio for many key suggestions and helpful comments.

REFERENCES

Benhamou, F., McAllester, D., & Van Hentenryck, P. (1994). *CLP (intervals) revisited*. Report CS-94-18. Brown University, Providence, Rhode Island.

Bowen, J. & Bahler, D. (1991). Conditional existence of variables in generalized constraint networks. *Proc. Ninth National Conf. on AI*, pp. 215–220. Anaheim, California.

Bradley, S.R. & Agogino, A.M. (1993). Computer-assisted catalog selection with multiple objectives. *Proc. Design Theory and Methodology—ASME 1993*, 139–147.

Carlson, B., Carlsson, M., & Diaz, D. (1994). Entailment of finite domain constraints. *Proc. Eleventh Int. Conf. of Logic Programming*, MIT Press.

Colton, J.S. & Ouellette, M.P. (1993). A form verification system for the conceptual design of complex mechanical systems. *Proc. ASME Advances in Design Automation (DE-Vol. 65-1)*, pp. 97–108. ASME, Albuquerque, New Mexico.

Darr, T. (1997). *A multi-attribute, interval distributed constraint-satisfaction problem approach to solving part-selection problems* (PhD Thesis). The University of Michigan, Michigan.

Darr, T.P., Birmingham, W.P., & Scala, N. (1998). A MAD approach to solving part-selection problems. In *Artificial Intelligence in Design*, pp. 251–270. Lisbon, Portugal.

Darr, T.P. & Dym, C.L. (1997). Configuration design: An overview. In *The Handbook of Applied Expert Systems*, (Liebowitz, J., Ed.), Chapter 21, pp. 21-1–21-15. CRC Press, Florida.

Davis, E. (1987). Constraint propagation with interval labels. *Artificial Intelligence 32(3)*, 281–331.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence 49(1-3)*, 61–95.

de Kleer, J. & Brown, J.S. (1984). A qualitative physics based on confluences. *Artificial Intelligence 24(3)*, 7–83.

Faltings, B. (1994). Arc-consistency for continuous variables. *Artificial Intelligence 65(2)*, 363–376.

Finch, W.W. & Ward, A.C. (1995). Generalized set-propagation operations over relations of more than three variables. *AIEDAM 9(3)*, 231–242.

Fishburn, P.C. (1970). *Utility Theory and Decision Making*. John Wiley & Sons, Inc., New York.

Freuder, E.C. (1978). Synthesizing constraint expressions. *Communications of the ACM 21(11)*, 958–966.

Gupta, A.P., Birmingham, W.P., & Siewiorek, D.P. (1993). Automating the design of computer systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12(4)*, 473–487.

Haworth, M.S., Birmingham, W.P., & Haworth, D.E. (1993). Optimal part selection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 12(10)*, 1611–1617.

Hyvonen, E. (1992). Constraint reasoning based on interval arithmetic: The tolerance propagation approach. *Artificial Intelligence 58(1-3)*, 71–112.

Kota, S. & Lee, C.-L. (1993). General framework for configuration design: Part 1—Methodology. *Journal of Engineering Design 4(4)*, 277–289.

Lyon, T.D. & Mistree, F. (1985). A computer-based method for the preliminary design of ships. *Journal of Ship Research 29(4)*, 251–269.

Mackworth, A.K. (1977). Consistency in networks of relations. *Artificial Intelligence 8(1)*, 99–118.

Mackworth, A.K. & Freuder, E.C. (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence 25(1)*, 65–74.

Marcus, S., Stout, J., & McDermott, J. (1988). VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine 9*, 95–112.

Mistree, F., Patel, B., & Vadde, S. (1994). On modeling multiple objectives and multi-level decisions in concurrent design. *Proc. Advances in Design Automation (DE-Vol. 69-2)*, pp. 151–161. ASME, Minneapolis, Minnesota.

Mittal, S. & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. Eighth National Conf. on Artificial Intelligence (AAAI-90)*, 25–32.

Mittal, S. & Frayman, F. (1987). COSSACK: A constraints-based expert system for configuration tasks. *Proc. Second Int. Conf. on Applications of AI to Engineering*, Boston, Massachusetts.

Mittal, S. & Frayman, F. (1989). Towards a generic model of configuration tasks. *Proc. Eleventh Int. Joint Conf. on Artificial Intelligence (IJCAI-89)*, pp. 1395–1401. Morgan Kaufmann, Detroit, Michigan.

Navinchandra, D. & Rinderle, J.R. (1990). Interval approaches for concurrent evaluation of design constraints. *Winter Annual Meeting of the ASME Symposium on Concurrent Product and Process Design DE-Vol. 21*, 101–108.

Rahmer, J. & Voss, A. (1998). Supporting explorative configuration. In *Artificial Intelligence in Design*, pp. 483–498. Lisbon, Portugal.

Schreiber, A.T. & Birmingham, W.P. (Eds.). (1996). Special Issue: The sisyphus-VT Initiative. *International Journal of Human-Computer Studies 44*. Academic Press, London

Sussman, G.J. & Steele, G.L. (1980). CONSTRAINTS—A language for expressing almost-hierarchical descriptions. *Artificial Intelligence 14*, 1–39.

Umeda, Y., Takeda, H., Tomiyama, T., & Yoshikawa, H. (1990). Function, behavior, and structure. *Proc. Applications of AI in Engineering V*, 177–193. Boston, Massachusetts.

van Beek, P. (1992*a*). On the minimality and decomposability of constraint networks. *Proc. Tenth National Conf. on Artificial Intelligence (AAAI-92)*, 447–452.

van Beek, P. (1992*b*). Reasoning about qualitative temporal information. *Artificial Intelligence 58*, 297–326.

Van Hentenryck, P., Deville, Y., & Teng, C.-M. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57(2-3)*, 291–321.

Van Hentenryck, P. & Michel, L. (1995). *Helios: A modeling language for nonlinear constraint solving and global optimization using interval analysis*. Report CS-95-33. Brown University, Providence, Rhode Island.

Yost, G.R. & Rothenfluh, T.R. (1996). Configuring elevator systems. *International Journal of Human-Computer Studies 44(3/4)*, 521–568.

Timothy P. Darr is a senior consultant and technical project manager at Trilogy Development Group in Austin, Texas. He received his Ph.D. in computer science at the University of Michigan, Ann Arbor. His research interests are in the areas of distributed design, concurrent engineering and constraint-satisfaction problems.

William P. Birmingham is an associate professor in the Electrical Engineering and Computer Science Department with a joint appointment in the School of Information, both at the University of Michigan, Ann Arbor. Birmingham's research interests are in the areas of distributed design, concurrent engineering, and AI applied to design. He is the editor of the journal ***AI**EDAM* (Artificial Intelligence for Engineering Design, Analysis, and Manufacture).