*Int. J. Human–Computer Studies* (1994) **40**, 221–241

# Solving Sisyphus by design

ALAN BALKANY, WILLIAM P. BIRMINGHAM AND JAY RUNKEL
*Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109, USA*

This paper demonstrates how the Domain-Independent Design System (DIDS) was used to solve the Sisyphus room-assignment problem by viewing it as a configuration-design task. We have developed a general problem-solving method for configuration design, based on constraint-satisfaction techniques. This method efficiently solves the Sisyphus problem, and provides strong guidance for knowledge acquisition. This paper presents both the problem solver and knowledge-acquisition support created by DIDS to solve the Sisyphus problem.

## 1. Introduction

Design problem solving finds solutions through a constructive process, where solutions are generated, rather than chosen as in classification or diagnostic problem solving. Construction is a more effective paradigm for design problems because the search space is large; even for restricted design problems, such as configuration, the search space can be exponential (Mittal & Frayman, 1989; Haworth, Birmingham & Haworth, 1992). Thus, it is not feasible to enumerate solutions; the process of creating these solutions is the hard part of the activity. When viewed in this light, design is ubiquitous, as many problems can be cast as design problems. For example, planning and scheduling can be solved very efficiently with design problem solvers.

The Sisyphus room-assignment problem, in our view, is a design task. The problem, as we demonstrate later in this paper, has a very large search space. Furthermore, design problem-solving methods (PSMs) (McDermott, 1988) are very effective in solving the Sisyphus problem.

In particular, Sisyphus can be modeled as a configuration problem. In a configuration problem, a set of needed functions is mapped to a part library such that every needed function is supplied by a part, and constraints are not violated (Balkany *et al.*, 1993). Preferences (often given in the form of utilities) are used to rank the solutions. In Sisyphus, rooms correspond to the part library, and people to the functions. People are placed in rooms consistent with the constraints given in the original problem statement. For example, if a smoker is assigned to a room, then only another smoker may be assigned to that room. The preference used in Sisyphus was to minimize distances between certain group members.

We have developed a model of configuration-design problem solving, called DIDS (Runkel *et al.*, 1992; Balkany *et al.*, 1993) (Domain-Independent Design System), that was used to solve the Sisyphus problem. DIDS uses a library of reusable software elements, called *mechanisms* [after the work of Klinker *et al.* (1990)], and *process models,* which integrate mechanisms into an efficient PSM. A *constraint-satisfaction-problem* (CSP) model was chosen for Sisyphus, using chronological



1071-5819/94/020221 + 21$08.00/0 © 1994 Academic Press Limited

backtracking and constraint propagation (Sussman & Steele, 1980) to find solutions. It is interesting to note that the same process model and PSM used for Sisyphus have been used to solve other configuration-design problems.

In this paper, we describe our approach to the Sisyphus problem. We begin with an overview of the DIDS system in Section 2, which is followed by a description of the problem formulation in Section 3. In Section 4, we describe the knowledge-acquisition process. Section 5 discusses a number of issues, such as reusability and computation efficiency, and Section 6 concludes the paper.

## 2. DIDS model

DIDS facilitates the design of knowledge systems through reusability. In studying configuration-design systems (Balkany, Birmingham & Tommelein, 1993), we have identified the following elements that were shared among systems: *knowledge structures*, operators on those structures that we call *mechanisms*, and models of computation called *process models*. The combination of these elements forms the DIDS model; a design-tool development environment based on the DIDS model has been created, but is not described here [see Runkel *et al.* (1992) and Balkany *et al.* (1993) for more information]. Each element of the DIDS model is discussed below.

### 2.1. MECHANISMS AND PSMs

Mechanisms implement techniques for solving design problems. Two mechanisms may perform the same function (e.g. part selection), but may differ in the algorithm used or the structures of knowledge used. Each mechanism is implemented by a code fragment, and is associated with a procedure for acquiring the domain knowledge required for that mechanism to operate. Each mechanism is characterized by a set of task features that describe when it should be used, and a description of its inputs and outputs. For example, Figure 1 shows the pseudo code and input and output parameters for a mechanism that performs part selection.

To be useful, mechanisms must be combined within a control structure, thereby forming a PSM. For example, Figure 2 shows a PSM generated by DIDS to perform

```
select-part mechanism
Input:      Function to be implemented, list of possible parts, constraints
Output:     A part, from part list input, that implements the function input without
            violating any constraints.
Returns:    TRUE if a part was selected.
Algorithm:  For each part in the part list:
                test to see if it violates any constraints
                if part does not violate any constraints,
                    then select it and return TRUE
            If all parts violate a constraint,
                then return FALSE.
```

FIGURE 1. A mechanism for selecting parts.

```
1.   WHILE (design-not-done (Functions functions-not-assigned-parts))
         Begin
2.       get-next-design-task (Functions functions-not-assigned-parts,
             Functions function-to-assign)
3.       IF (not (designed (Functions function-to-assign)))
4.           order-domain (Functions function-to-assign, Preferences prefs)
5.       WHILE (get-next-part (Functions function-to-assign, Part part))
             Begin
6.           add-part (Functions function-to-assign, Part part,
                 Functions functions-assigned-parts, Constraints consts)
7.           IF (designed (Functions function-to-assign)) go to 8.
             End
8.       IF (not (designed (Functions function-to-assign)))
9.           chronological-backtrack (Functions functions-not-assigned-parts,
                 Functions functions-assigned-parts, Functions function-to-assign)
         End
10.  display-solution (Functions functions-assigned-parts)
```

FIGURE 2. Sisyphus PSM.

the Sisyphus task. The mechanisms are highlighted in boldface, and the outermost WHILE loop defines the loop over which the problem solver iterates. Mechanism parameters are represented in the form: (parameter-type parameter-name [, parameter-name . . .]). Mechanisms that execute conditionally are contained within the IF and WHILE statements inside of the outer WHILE loop. A complete description of the functions of the mechanisms used in the PSM, along with their characterizations, is given in Appendix 1.

As mentioned previously, each mechanism has a procedure for acquiring the knowledge used by it. These procedures, which are called *Mechanisms for Knowledge Acquisition* (MeKAs), define a model-based knowledge-acquisition tool for acquiring the knowledge structures used by a mechanism (Runkel & Birmingham, 1992). MeKAs are model based because they use the mechanism's assump-

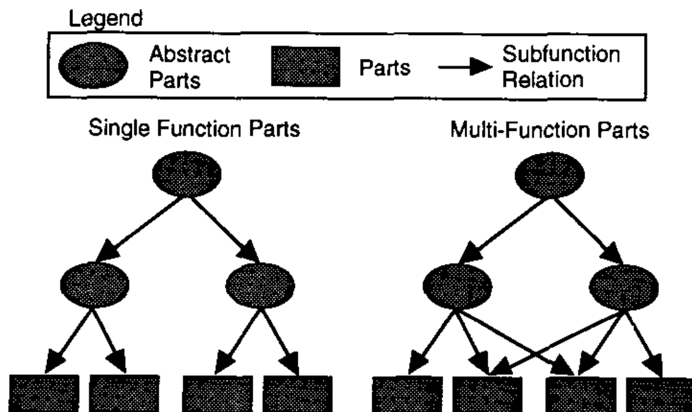


FIGURE 3. Two alternate decompositions of abstract parts.

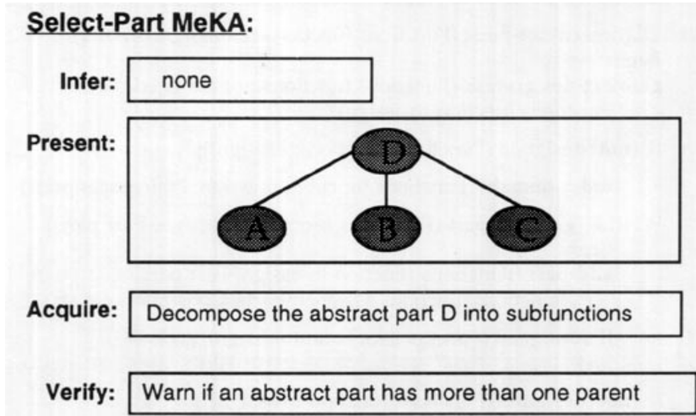


FIGURE 4. A MeKA for a mechanism that selects parts.

tions concerning the knowledge structures available in the domain and the relationships between these knowledge structures to guide knowledge acquisition. For example, if a mechanism assumes that the domain will use abstract-part decomposition, like the example in Figure 3, then the MeKA will ensure that all abstract parts are the subfunction of at most one abstract part. This MeKA is shown in Figure 4.

A MeKA has four elements (see Figure 4)—infer, present, acquire, and verify—which correspond to the four-step process used to acquire knowledge for a mechanism. The *infer* element uses a MeKA-specific inference procedure to automatically derive the necessary knowledge. The *present* element of a MeKA acts as a filter, presenting only the relevant elements of the knowledge base to the domain expert. The information displayed provides enough details to give the domain expert the appropriate context for the knowledge being requested, without overwhelming him with the complexities of the knowledge base. The *acquire* element describes how to acquire knowledge in the context of the currently active portion of the knowledge base. The *verify* element describes how to check the integrity of a newly acquired piece of knowledge.

### 2.2. KNOWLEDGE STRUCTURES

The DIDS model defines a set of ten knowledge structures that identify what knowledge is required to perform configuration design. These knowledge structures were identified by studying existing configuration-design systems (Balkany, Birmingham & Tommelein, 1993), and we believe that they are sufficient to represent all the domain knowledge necessary for any configuration task. Appendix 2 provides a complete list of knowledge structures. Of the ten possible structures, the following were used to solve the Sisyphus task:

*Parts* are artifacts (things in the real world) that implement functions of interest for a particular design. Parts have a name and function, and may have an arbitrary number of attributes.

*Functions* define what is required of the artifact being designed. Functions have a name and may have an arbitrary number of attributes.

*Constraints* restrict the space that is searched to find a solution. Constraints are defined over the attributes of parts and functions. In DIDS's problem formulations, the result of the design process cannot violate any constraints.

*Preferences* define what is preferred in solutions. Preferences provide a gradient on the space defined by the constraints. DIDS represents preferences as utility functions.

Note that instantiation of these knowledge structures depends on the process model chosen. Section 2.3 illustrates how a constraint-satisfaction model uses these knowledge structures, except for preferences, the use of which is described in Section 3.2.2.

The knowledge structures also act as a set of primitives that are used to define the functionality of mechanisms, the knowledge communicated between mechanisms, and the domain knowledge that is used by mechanisms. All mechanisms are defined by the operations that they perform on the knowledge structures; the inputs and outputs of mechanisms must be in terms of the knowledge structures. For example, a mechanism may take an abstract part (i.e. a function) as an input, and return a part that implements it, or a mechanism can be given a set of parts and determine the connections among them.

Defining mechanisms in this way has two advantages. First, the mechanisms will be reusable and combinable. The definition ensures reusability, because it places no restrictions on the specific domain concepts that must be supplied, or on the source domain of the concepts. The only requirement is that the knowledge-type classification of domain concepts is the same as the inputs to the mechanism. This guarantees that a mechanism can be applied to any configuration task where the domain contains the appropriate knowledge structures. The knowledge structure definition also ensures the combinability of mechanisms since all the mechanisms share a common representation of these structures. Therefore, any two mechanisms that use the same knowledge structures can share information, and can be easily combined. Second, this definition makes clear exactly which knowledge must be in the knowledge base for each mechanism to operate. This information can be used to guide the selection of mechanisms when constructing a PSM and to guide the construction of a knowledge-acquisition tool for the method.

### 2.3. THE PROCESS MODEL

The process model provides both a mapping from knowledge structures to data structures, and basic inference techniques to support a PSM. The model describes how each knowledge structure is best represented for a particular problem. The inference techniques support mechanisms, but are not usefully represented as mechanisms because they do not require domain knowledge for their operation, and they rely on a particular knowledge representation that is not portable. The relationship between mechanisms and the process model is illustrated in Figure 5.

Different process models can be used for the same task. For example, configuration design can make use of either a table (Haworth, Birmingham & Haworth, 1992) or a constraint-network representation. Tables are best used for problems where parts are easily organized as a hierarchy, and finding parts to cover high-level functions is the primary consideration. The constraint network is applicable to problems where problem solving is dominated by constraint satisfaction.

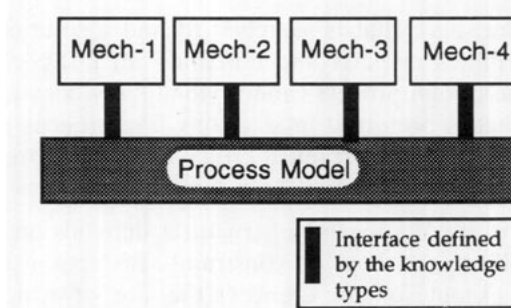


FIGURE 5. Relationship between the process model and mechanisms.

For the Sisyphus problem, the constraint network was chosen because of the simple, relatively non-hierarchic part library (set of rooms) structure. The constraint network represents a dynamic constraint-satisfaction problem with preferences (DCSPP), which is a variation of the dynamic constraint-satisfaction problem (Mittal & Falkenhainer, 1990).

The DCSPP can be viewed as a graph, where variables are nodes and constraints are arcs. Each variable has a domain, which is the set of values that the variable can assume. Each constraint and variable has a predicate, called an *in-list,* that determines when it is active. Thus, constraints and variables can change as the design progresses. In our formulation, *functions* are variables, and have domains corresponding to the *set of parts* that implement that function. As described earlier, functions and parts have attributes. Constraints are formed over these attributes. A simple DCSPP network is given in Figure 6 for two functions and a part library, which correspond to people and rooms, respectively, in the Sisyphus example. The constraint is applicable if both people share a room, making its predicate TRUE.

Specifically, the network performs a special form of arc consistency (Mackworth, 1977) called *constraint propagation* as variables are assigned values. It uses the constraints and the known values of attributes to compute values of attributes whose values are not known. Furthermore, arc- and node-consistency operations ensure that the network is consistent; i.e. all values that are provably infeasible are removed from domains. These operations, which are assumed by mechanisms, and hence the PSM, provide supporting inference techniques. As node- and arc-consistency operations are tightly linked to the DCSPP representation, they are not especially reusable for other process models. Hence, these operations are not made into mechanisms.

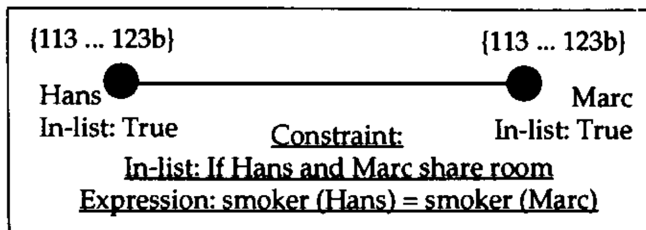


FIGURE 6. Simple DCSPP network.

## 3. Problem formulation

In this section, we discuss how Sisyphus was solved using the DIDS model elements described in Section 2. We begin with a list of assumptions about the problem statement.

### 3.1. ASSUMPTIONS

The Sisyphus problem is intended to model a real-world situation. As such, there are a number of statements that require interpretation. The following are the assumptions made:

1. All problem requirements can be classified as *constraints* or *preferences.* Constraints are conditions that must be satisfied, such as "a smoker and a non-smoker may not share a double office". Preferences are desirable solution characteristics that can be partially satisfied, such as "the heads of large projects should be close to the head of the group and the secretaries".
2. The central offices are C5-116, C5-117, and C5-119.
3. Doorways are assumed to be in the middle of the wall between the office and the hallway. For room C5-119, the doorway is in the left wall, and for C5-123, it is on the bottom wall in the floor plan (see Section 3.2.2 for floor plan).
4. The hacker attribute is irrelevant; it is not explicitly mentioned in any of Siggi D.'s annotations. In addition, in the two pairings that are said to create synergy (Werner L./Jürgen L., and Harry C./Michael T.) both are hackers.
5. The works-with attribute is irrelevant. It does not appear in any of the constraints or preferences, and crosses project boundaries.

### 3.2. FORMING THE DSCPP

Each person is modeled as a function whose part library consists of the rooms in the chateau. The function (people) and part (rooms) models are given in Figure 7.

Names are used to identify people and rooms. The role, project, room size and smoker attributes on people, and the central and size attributes on rooms appear in constraints. These attributes are determined during knowledge acquisition, making the system robust with respect to changes in the function or part models.

#### *3.2.1. Constraints*

All constraints used in our formulation of the problem are given in Figure 8. (The numbers in square brackets refer to Siggi D.'s annotations.) As constraints are dynamic, each has an in-list.

| **Function attributes:** | **Part attributes:** |
|---|---|
| Name | Name |
| Role | Central |
| Project | Size |
| Smoker | |
| Room size required | |

FIGURE 7. Function model and part model in the Sisyphus problem.

| | |
|---|---|
| a) The room should not already be occupied.† | |
| b) Small rooms can hold only one person | |
| c) The group head should have his/her own large room | [1a] |
| d) A secretary should have a large room. | [2a] |
| e) A manager should have a central room. | [3a] |
| f) A manager should have a small room. | [3a] |
| g) A project leader should have small room | [Note 1] |
| h) If a room is large, both or neither occupants should be secretaries. | [2a] |
| i) If a room is large, both or neither occupants should be smokers. | [7a] |
| j) Researchers must have a large room | |

FIGURE 8. Constraints of the Sisyphus problem. † Large rooms were represented internally as a pair of single rooms, e.g. 117a and 117b. This representation had no impact on the solution.

In the constraint network, both variables and domain values may have attributes. A variable has a special attribute, called *value*, that represents the value that has been assigned to it. Some examples are the following: `Monika.smokes` refers to the *smokes* attribute of the variable representing Monika. `Monika.value` represents the value (the room) that has been assigned to the Monika variable. Finally, `Monika.value.central` refers to the central attribute of the value that has been assigned to the Monika variable, i.e. whether Monika's room is central.

At the beginning of the problem, each person can potentially go into any room; the domain of each person is all the rooms in the chateau. During problem solving, the domains are pruned using the operations discussed previously.

We now consider the representation of three of the Sisyphus constraints in DIDS. (The others are formulated in a similar manner.) The constraint that the group head must have a large room is represented as follows:

```
(Thomas.value.size is-equal ''large'')
```

This constraint would be repeated for every group head, if there were more than one. The "value" attribute of the variable, Thomas, represents the room that Thomas has been assigned to. The "size" attribute of this room is, of course, the room's size. A similar constraint is used for project leaders, as shown below:

```
(Katharina.value.size is-equal ''small'')
```

The constraint that a room should not already be occupied is implemented by testing if any two people occupy the same room after each room assignment, such as:

```
(Thomas.value not-equal Monika.value)
```

Through constraint propagation, a person would never actually be assigned to a room that is already occupied. This constraint is duplicated for every possible pair of people. Since there are 15 people, there are $\binom{15}{2} = 105$ constraints of this form.

Finally, the constraints on pairs of people in large rooms must be duplicated for every possible pair of people in every large room. For example, to ensure that both roommates are or are not smokers, we use constraints of the form:

```
(Thomas.smokes is-equal Monika.smokes)
```

with the following in-list:

```
(((Thomas.value is-equal Rm117a) AND (Monika.value is-equal
Rm117b)) OR
((Thomas.value is-equal Rm117b) AND (Monika.-value is-equal
Rm117a)))
```

This means that if Thomas and Monika share a large room (which makes the in-list TRUE), we test if both or neither are smokers. Since there are 105 possible pairs of people, and six large rooms, we must use $105 * 6 = 630$ constraints of this form. Because of the in-lists, however, only a small fraction of these constraints are active at any one time.

Two things need to be considered: acquiring the constraints and checking them (run-time efficiency). These constraints are generated automatically by a knowledge-acquisition tool from a simple format, as discussed in Section 4. Furthermore, not all constraints will be checked during a design, only those that are active. Thus, it is the number of constraint checks that is important to run-time efficiency, not the actual number of constraints. The number of checks made is a function of the problem. Thus, the size of the network is not a significant issue. A more detailed discussion is given in Section 5.2.

When backtracking is used in a CSP, a given domain value may be tried and retracted many times. If it can be determined in advance that this domain value cannot be used successfully, considerable time may be saved. Constraint propagation accomplishes this, using constraints between variables to determine domain values that can never satisfy them, and pruning these values. In some cases, constraint propagation prunes variable domains so drastically that no search is required to produce a solution. (See the discussion in Section 5.2.)

*3.2.2. Preferences*

The preferences in this problem specify the closeness of the rooms assigned to various people. All preferences are of the form "A should be close to B". Preferences provide a way of ranking the desirability of different room assignments that satisfy the constraints. The preferences used are given in Figure 9, with references to Siggi D.'s annotations given in square brackets.

| | |
|---|---|
| a) Group head should be close to group members. | [1a] |
| b) Group head should be close to secretaries. | [2a] |
| c) Manager should be close to group head. | [3a] |
| d) Manager should be close to secretaries. | [3a] |
| e) Project heads should be close to group head. | [4a-6a] |
| f) Project heads should be close to secretaries. | [4a-6a] |

FIGURE 9. Preferences of the Sisyphus problem.

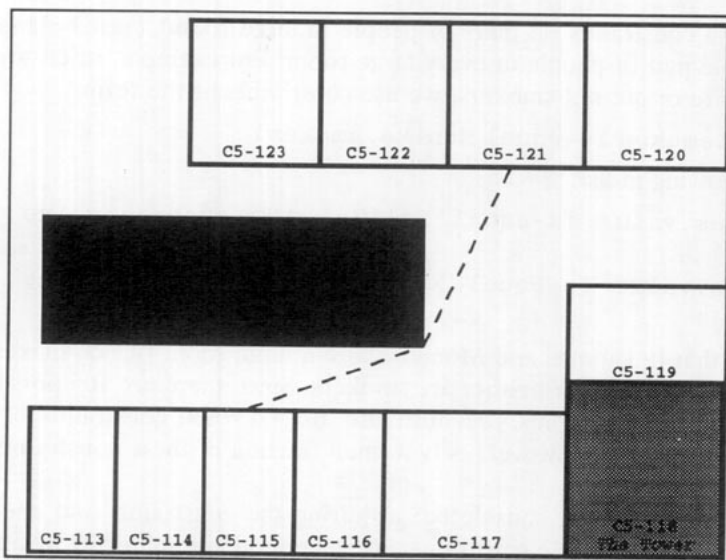


FIGURE 10. A path on the chateau's floor plan.

In order to reason about preferences involving closeness, we need a numerical representation of how well the close-to preference is satisfied. We define the *distance* between offices as the length of the shortest path through hallways from one doorway to the other, as shown in Figure 10.

Let $p_{ij}$ be the closeness preference that applies to person $i$ and person $j$. Each preference $p_{ij}$ states that person $i$ and person $j$ should be assigned rooms that are close to each other. We define the *violation level* for person $i$ as the sum of squares of the distances between person $i$ and person $j$, for each preference concerning person $i$:

$$\text{violation level}_i = \sum_{p_{ij}} [\text{distance}_{i,j}]^2$$

The violation level is a measure of how many and how severely the room assignment for person $i$ "violates" preferences. This is a *least-squares* criterion that tends to favor room assignments without severe preference violations. In our solution to the Sisyphus problem, a greedy algorithm was used, in which rooms with lower violation levels (that minimize the preference violations) are tried first. This ordering guarantees that the first value found for a variable that satisfies the constraints will be the most preferable in the domain.

For a complete solution that satisfies all constraints, we define the *total violation level* as the sum of the violation levels over all the preferences. The total variation level gives us a metric for comparing different solutions that satisfy all the constraints.

Note that since this is local optimization, it is vulnerable to the horizon effect, so global optimization is not guaranteed. Still, the algorithm uses preferences to produce a solution that is probably better than a solution produced by an algorithm that did not take preferences into account.

Our results, described in the next section, indicate that preferences have little effect in a highly constrained problem, such as room assignment. This is because constraints have precedence over preferences. Thus, as the number of solutions approaches one, the effect of preferences degrades to having no effect at all. On other runs of the room-assignment problem with fewer constraints, and therefore more solutions, preferences did improve the solutions.

### 3.3. SOLUTIONS AND RUN TIMES

A discussion of the solution to the original problem is given in the following sections.

#### *3.3.1. The original problem*

The solution to the Sisyphus problem produced by DIDS is given in Figure 11, which is the same as Siggi D.'s solution.

#### *3.3.2. The extended problem*

A second problem was given in which Katharina N. left and Christian I. joins the group. Christian I. works on the same project as Katharina N. did, and smokes. Replacing Katharina N. by Christian I. renders the original problem unsolvable without violating a constraint, as either a researcher must be placed in a single room or a smoker and non-smoker must share a room. To solve the extended problem, we decided to place the smoking managers in the same large room, and leave all other constraints the same.

Changes to the data to handle the change in personnel were trivial. The following changes were made through the knowledge-acquisition tools:

1. Katharina's name was replaced by Christian's in the list of variables. (The domain remains the same as Katharina's; all of the rooms.)
2. Christian's attributes were defined by setting his role to researcher, his project to MLT, and his smoker attribute to TRUE.
3. The preferences that involved Katharina were removed from the knowledge base. These specified that Katharina had to be close to Thomas D., Ulrike U., and Monika X.

There was a change made to the PSM. The PSM requires that all constraints be satisfied for a valid solution. Since this is not possible, we decided to *turn off* various constraints, thereby relaxing the problem and allowing a solution. Turning off a constraint is simple: a variable (valid) is created for each class of constraint, and is added to the in-list of each constraint; the truth value of this variable is controlled

| | | | |
|---|---|---|---|
| Katharina_N | Rm113 | Hans_W | Rm114 |
| Joachim_I | Rm115 | Eva_I | Rm116 |
| Michael_T | Rm123 | Harry_C | Rm123 |
| Angi_W | Rm122 | Marc_M | Rm122 |
| Jurgen_L | Rm121 | Werner_L | Rm121 |
| Uwe_T | Rm120 | Andy_L | Rm120 |
| Ulrike_U | Rm119 | Monika_X | Rm119 |
| Thomas_D | Rm117 | | |

FIGURE 11. DIDS solution to the Sisyphus problem.

| | | | |
|---|---|---|---|
| Christian_I | Rm123 | Hans_W | Rm123 |
| Joachim_I | Rm115 | Eva_I | Rm116 |
| Michael_T | Rm122 | Harry_C | Rm122 |
| Angi_W | Rm121 | Marc_M | Rm121 |
| Jurgen_L | Rm120 | Werner_L | Rm120 |
| Uwe_T | Rm114 | Andy_L | Rm113 |
| Ulrike_U | Rm119 | Monika_X | Rm119 |
| Thomas_D | Rm117 | | |

FIGURE 12. DIDS solution to the extended problem.

by the PSM. Thus, making valid false for a class of constraints will effectively remove them from consideration during problem solving. This change was made by hand.

To produce a solution for the extended problem, the PSM, when it has determined that there is no solution, turns off constraints *j* (programmers in large rooms), and—for smokers only—*g* (project leaders in small rooms). The new PSM yielded the solution shown in Figure 12.

## 4. Knowledge acquisition

*The process model and knowledge structures provide a strong model for the* knowledge-acquisition task. In particular, a knowledge-acquisition tool is associated with each knowledge type. Each tool provides an interface that facilitates capturing the knowledge type with which it is associated. Further, in some cases, inferences can be made to check the correctness and completeness of the acquired knowledge. The process model defines how the acquired knowledge should be represented as specific data structures that the PSM can use.

The assistance provided by the DIDS knowledge-acquisition tools, as demonstrated in this section, is substantial. For example, only a few high-level constraints need be entered, instead of the hundreds required by the constraint network. Design problems are generally more knowledge intensive than the Sisyphus problem. As described by Runkel and Birmingham (1992), DIDS requires a sophisticated set of heuristics to create knowledge-acquisition tools with the power of model-based tools typically used in design problems, such as CGEN (Birmingham & Siewiorek, 1989) and SALT (Marcus, 1988). This is an area of ongoing research.

In the following sections, we describe how knowledge was acquired for the Sisyphus task. In each case, we show a representative piece of each knowledge type acquired using the DIDS knowledge-acquisition tools. The following sections assume that the knowledge engineer has determined the knowledge structures required to solve the problem. Each knowledge-acquisition tool is then invoked from a menu. We are currently developing task-level modeling tools that will automatically determine both the knowledge structures required for a problem and the sequence in which to invoke the knowledge-acquisition tools.

### 4.1. DEFINING ROOMS AND PEOPLE

The first step of the knowledge-acquisition process is defining the rooms and people. This includes specifying the attributes and providing instances. Figure 13 shows Werner's definition. Since people are considered functions in the DCSPP model,

Constraints

InList

TRUE

| Attribute | Type | Domain | Value |
|---|---|---|---|
| smoker | CHARACTERISTIC | NIL | NO |
| project | CHARACTERISTIC | NIL | RESPECT |
| role | CHARACTERISTIC | NIL | RESEARCHER |
| name | CHARACTERISTIC | NIL | WERNER_L |

FIGURE 13. Defining a person.

they have an in-list. Like all occupants of the castle, Werner's in-list is TRUE since he must always be considered in the problem.

The knowledge-type *functions* has attributes, and the Function knowledge-acquisition tool allows the knowledge engineer to define attributes for each person. In an earlier step, not shown here, the class of attributes that are applied to all people was defined. In Figure 13, Werner's attributes are defined. A frame like the one in Figure 13 is generated for each person in the castle.

Rooms are defined in a similar way to people. A frame is generated for each room, similar to the one in Figure 14, where room 117a is defined. The figure assumes that the attributes for each room have been previously defined. As explained in Section 3, rooms that hold two people are split in two, denoted as 'a' and 'b'.

During the knowledge-acquisition process, the person and room instances can be accessed in several ways via a collection of *browsers*, as shown in Figure 15. The function and part browsers organize information alphabetically by instance. Clicking on an instance provides access to a part or function frame.

In addition, the relationships between people and rooms can be shown. In Figure 16, the rooms that Hans can be assigned to are displayed. The tool that displays

Cut Row | New Row | Paste Row | OK

| Attribute | Type | Domain | Value |
|---|---|---|---|
| size | CHARACTERISTIC | NIL | LARGE |
| central | CHARACTERISTIC | NIL | NO |
| name | CHARACTERISTIC | NIL | CS_117A |

FIGURE 14. Defining a room.

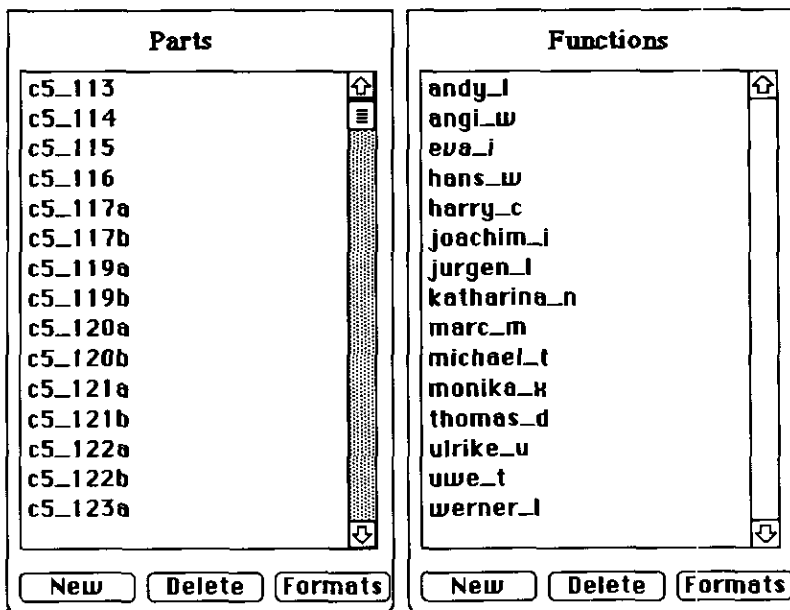


FIGURE 15. Part and function browsers.

these relationships is the Hierarchy Browser; so called because parts and functions commonly form a hierarchy in the design domain (Haworth & Birmingham, 1992). In general, the hierarchy may be arbitrarily deep, and the browser is able to display any such hierarchy. For the room-assignment problem, however, a two-level hierarchy is sufficient.

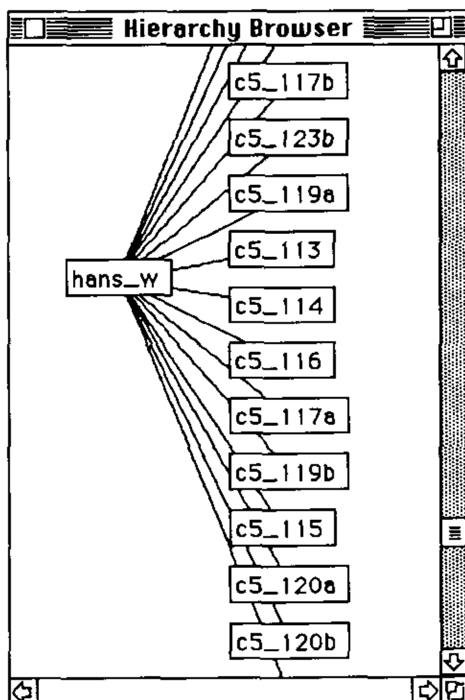


FIGURE 16. Function-part relationship.

| | |
|---|---|
| **Formula** | (harry_c.value != thomas_d.value) |
| **In-List** | TRUE |

FIGURE 17. Person incompatibility defined as a constraint.

### 4.2. DEFINING CONSTRAINTS

Once people and rooms are defined, constraints can be defined. As defined by the constraint knowledge structure, there are two parts to the constraint definition: the in-list and the formula. Thus, the knowledge-acquisition tool provides facilities for both parts.

Figure 17 shows a constraint stating that Harry_C and Thomas_D cannot be assigned to the same room position. Remember the two positions in a double room are represented by decomposing it into two parts. The constraint in Figure 17 ensures that Harry_C and Thomas_D are not assigned to the same part, i.e., room position.

A compatibility constraint is defined in Figure 18. It was generated from the problem statement requiring that the two secretaries share the same large room. The constraint in Figure 18 represents a portion of this problem statement by requiring that when Uwe_T and Thomas_D share room C5-123, they must either both be secretaries or neither of them may be secretaries. The in-list specifies that the constraint applies only when Uwe_T and Thomas_D are assigned to room C5-123, and the formula specifies that either both or neither of them must be secretaries. In order to completely represent the problem statement, a similar constraint must be generated between every pair of people and all possible large room assignments of these people.

## 5. Discussion

In this section, we discuss the adequacy of DIDS's representations, the computational complexity of the solution technique, and reusability issues.

### 5.1. REPRESENTATIONAL ADEQUACY

The DIDS model has been shown to be general enough to represent and solve not only the Sisyphus problem, but also a wide range of other configuration-design

| | |
|---|---|
| **Formula** | (((thomas_d.role = 'secretary') AND (uwe_t.role = 'secretary')) OR ((thomas_d.role != 'secretary') AND (uwe_t.role != 'secretary'))) |
| **In-List** | (((thomas_d.value = c5_123a) AND (uwe_t.value = c5_123b)) OR ((thomas_d.value = c5_123b) AND (uwe_t.value = c5_123a))) |

FIGURE 18. Person compatibility defined as a constraint.

problems, although some customization was needed for this problem. This customization regarded the preferences. DIDS assumes a piecewise linear utility function, and the room-assignment problem's preferences are not conveniently represented this way. So, the closeness preferences were modeled as a list of pairs of people who should be close to each other. The sum of squared distances was used for selecting the least-squares alternatives, which required a table of the distances between rooms.

It is possible that a small set of preference criteria could provide good coverage of this general type of problem. Minimizing and maximizing distances are natural preferences when a problem involves positions. In other domains, minimizing the cost of the components is a commonly used preference. The DIDS approach allows these preferences to be encapsulated into mechanisms, which can then be incorporated into the PSM. Knowledge acquisition for preferences would then involve selecting the preference criteria from a list of alternatives, and then using the selection to retrieve the corresponding mechanism(s).

### 5.2. COMPUTATIONAL COMPLEXITY

We did not solve the problem the way Siggi D. did. A more search-intensive approach was used, with chronological backtracking through the search space defined by the CSP formulation. Constraint propagation was used to prune variables' domains of values that could never participate in a solution. This reduced, or eliminated in some cases, the amount of backtracking required, making the search more efficient. We provide an analytic formulation of the run time of our approach below.

The running time of our algorithm depends on the sum of two factors:

A. the cost to enumerate solutions, and
B. the cost to prune domains.

The worst case for a general CSP, where a sizeable portion of the search space is searched with backtracking (factor A), has been shown to be NP-complete (Mohr & Henderson, 1986). Pruning domains is quadratic in the size of the domains in the worst case (Mohr & Henderson, 1986). There is also an interaction between pruning and search: successful pruning will drastically reduce the size of the space to search. The running time will fall somewhere on a spectrum between two cases: the general problem where pruning is not performed, and the DIDS case where pruning is aggressively performed. We examine each case in turn.

*Case 1: The general problem*

In the general problem, constraint propagation is not performed so there is little or no pruning. Here, the cost for factor B is eliminated, and since the search space is exhaustively searched, factor A gives the running time. We can calculate the size of this space for the Sisyphus problem with no pruning (the size corresponds to run time with some constant factor). There are 15 people, six double rooms, and four single rooms. The number of ways of filling the first double room is:

$$\binom{15}{2} \quad \text{where} \quad \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

which is the number of ways of choosing $k$ items from a set of $n$, without regard to order.

For the second double room, there are 13 people left, so the number of ways of filling it is:

$$\binom{13}{2}$$

After the six double rooms have been filled, there are 4! ways of putting the last 3 people in the 4 single rooms. Thus, the size of the search space in Sisyphus is:

$$\binom{15}{2}\times\binom{13}{2}\times\binom{11}{2}\times\binom{9}{2}\times\binom{7}{2}\times\binom{5}{2}\times 4! = \frac{15!\times 13!\times 11!\times 9!\times 7!\times 5!\times 4!}{2^6\times 13!\times 11!\times 9!\times 7!\times 5!\times 3!}$$

$$= \frac{15!\times 4!}{2^6\times 3!} \cong 8{\cdot}17\times 10^{10}$$

In general, for $n$ people ($n$ even), and all double rooms, the size of the search space is:

$$\prod_{i=1}^{n/2}\binom{2i}{2} = \frac{2!}{2!\times 0!}\times\frac{4!}{2!\times 2!}\times\cdots\times\frac{(n-4)!}{2!\times(n-6)!}$$

$$\times\frac{(n-2)!}{2!\times(n-4)!}\times\frac{n!}{2!(n-2)!} = \frac{n!}{2^{n/2}} = O(n!)$$

*Case 2: DIDS*

In this case, there is extensive pruning, with a corresponding reduction in the amount of search required. Thus, factor A is greatly diminished, and factor B dominates the running time. Two types of pruning are done: node consistency and arc consistency (constraint propagation). Node consistency involves pruning domains of values that violate a constraint on a single variable; a *unary* constraint. This takes $O(nua)$, where $n$ is the number of nodes (variables), $u$ is the number of unary constraints, and $a$ is the average domain size. Pruning using arc consistency is shown by Mohr and Henderson (1986) to be $o(ea^2)$, where $e$ is the number of constraints.

$O(ea^2)$ and $O(nua)$, both worst case running times, are a vast improvement over the $O(n!)$ running time in the general case. Empirical evidence from our runs of the Sisyphus problem indicate that the running time of the DIDS solution is dominated by pruning.

### 5.3. REUSABILITY

Much of the power of the DIDS approach comes from the reuse of mechanisms, MeKAs, and PSMs. Reuse is facilitated by two factors. First, the domain-independent nature of mechanisms and PSMs allows them to be reused for different domains, with a knowledge-acquisition tool to supply the needed domain knowledge. Second, the atomic functionality of the mechanisms and standardized knowledge structures allow mechanisms to be recombined in different ways for different problems, thus facilitating reuse.

An example of mechanism reuse can be seen in the DIDS solution to the VT task.

The VT elevator-configuration task (Marcus, Stout & McDermott, 1987) involves selection of models of parts to construct an elevator, subject to constraints and cost preferences. The PSM for this solution was composed entirely of mechanisms taken from the PSM used to solve Sisyphus. (The mechanism to reorder domains from the Sisyphus PSM was not used in the VT PSM, because the knowledge-acquisition tool preordered VT's parts by cost.) We are continuing to apply the mechanisms to new tests to determine the degree of reuse.

## 6. Conclusion

Solving the Sisyphus room-assignment problem requires searching a large space; the size of this space is $O(n!)$ where $n$ is the number of people. Thus, the problem is ideally cast as a design problem, since design techniques are created specifically to deal with large search spaces.

The PSM described in this paper formulates the Sisyphus problem as a constraint-satisfaction problem. People are variables, and rooms are values that can be taken by people. By exploiting properties of the constraint network that results from this formulation, we are able to solve the room-assignment problem very efficiently (search space of approximately $O(ea^2)$, where $e$ is the number of constraints and $a$ is the number of rooms).

Furthermore, this formulation provides a strong model of problem-solving behavior that can be used by knowledge-acquisition tools to ease the knowledge-acquisition task. The combination of the knowledge-acquisition tools and the constraint network provide a powerful system for solving complex design tasks.

The knowledge-acquisition tools and the PSM were constructed from a library of reusable elements in a framework called the DIDS system. The elements are composed of mechanisms, common knowledge structures, and a process model. The process model provides a set of basic inference methods and data structures to facilitate mechanism and knowledge structure reusability. In fact, the PSM (which is a combination of mechanisms that solves a family of tasks) has been used for a variety of other design problems.

This work was funded, in part, by a gift from Digital Equipment Corporation and by the National Science Foundation Grant MIPS-905781. The opinions expressed in this paper are those of the authors, and do not necessarily reflect those of Digital Equipment Corporation or the NSF. Manjote Haworth was instrumental in developing the constraint network.

## References

BALKANY, A., BIRMINGHAM, W. P., MAXIM, B. R., RUNKEL, J. T. & TOMMELEIN, I. D. (1993). DIDS: rapidly prototyping configuration design systems. *The Journal of Intelligent Manufacturing* (in press).

BALKANY, A., BIRMINGHAM, W. P. & TOMMELEIN, I. D. (1993). An analysis of several design tools. *Artificial Intelligence in Engineering, Design, and Manufacturing* (in press).

BIRMINGHAM, W. P. & SIEWIOREK, D. (1989). Automated knowledge acquisition for a computer hardware synthesis system. *Knowledge Acquisition*, **1(4)**.

HAWORTH, M. S. & BIRMINGHAM, W. P. (1992). *Towards optimal system-level design.* Technical report CSE-TR-144-92, The University of Michigan, Department of Electrical Engineering and Computer Science, Ann Arbor, MI, USA.

HAWORTH, M. S., BIRMINGHAM, W. P. & HAWORTH, D. E. (1992). *Optimal part selection.* Technical report CSE-TR-127-92, The University of Michigan, Department of Electrical Engineering and Computer Science, Ann Arbor, MI, USA.

KLINKER, G., BHOLA, C., DALLEMAGNE, G., MARQUES, D. & MCDERMOTT, J. (1990). Usable and reusable programming constructs. *Proceedings of the 5th Knowledge Acquisition Workshop,* American Association for Artificial Intelligence.

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

MARCUS, S. (1988). *Automating Knowledge Acquisition for Expert Systems.* Boston, MA: Kluwer.

MARCUS, S., STOUT, J. & MCDERMOTT, J. (1987). VI: an expert elevator designer that uses knowledge-based backtracking. *AI Magazine,* pp. 41–57.

MCDERMOTT, J. (1988). Preliminary steps toward a taxonomy of problem-solving methods. In S. MARCUS, Ed. *Automating Knowledge Acquisition for Expert Systems.* Boston, MA: Kluwer.

MITTAL, S. & FALKENHAINER, B. (1990). Dynamic constraint-satisfaction problems. *Proceedings of the Eighth National Conference on Artificial Intelligence,* pp. 25–32, Menlo Park, CA, American Association for Artificial Intelligence.

MITTAL, S. & FRAYMAN, F. (1989). Towards a generic model of configuration tasks. *Proceedings of the 11th IJCAI.*

MOHR, R. & HENDERSON, T. C. (1986). Arc and path consistency revisited. *Artificial Intelligence,* **28(2),** 225–233.

RUNKEL, J. T. & BIRMINGHAM, W. P. (1992). Knowledge acquisition in the small: issues in building knowledge-acquisition tools from pieces. *Proceedings of the 7th Knowledge Acquisition for Knowledge-Based Systems Workshop,* Banff, Canada.

RUNKEL, J. T., BIRMINGHAM, W. P., DARR, T. P., MAXIM, B. R. & TOMMELEIN, I. D. (1992). Domain-independent design system: environment for rapid development of configuration design systems. In J. S. GERO, Ed. *Proc. 2nd International Conference on Artificial Intelligence in Design, AID 92,* pp. 21–40, 22–25 June. Pittsburgh, PA: Kluwer.

SUSSMAN, G. J. & STEELE, G. L., Jr. (1980). CONSTRAINTS—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence,* **14,** 1–39.

## Appendix 1: the mechanism library

**add-part**

input: Constraints consts, Part part
both: Functions function-to-assign, functions-assigned-parts
Description: Attempts to use a part to cover the given function (*function-to-assign*). If part satisfies the constraints, it is associated with *function-to-assign,* and the function is added to *functions-assigned-parts.* If part does not satisfy the constraints, *function-to-assign* remains undesigned.

**chronological-backtrack**

input: Functions function-to-assign
both: Functions functions-not-assigned-parts, functions-assigned-parts
Description: Returns *function-to-assign* to the set, *functions-not-assigned-parts,* because no part could be found for it, and removes the last function added to *functions-assigned-parts.* This last function becomes the new *function-to-assign.*

**designed**

input: Functions function-to-assign
returns: TRUE if the function represented by *function-to-assign* has been assigned a part, else FALSE.

Description: In the Sisyphus PSM, this mechanism is used to determine whether to reorder the parts that make up the function's domain. This is only done before any parts have been tried, so that the parts that satisfy the preference the most are tried first.

**design-not-done**
input: Functions functions-not-assigned-parts
returns: TRUE if the set, *functions-not-assigned-parts*, is non-empty, indicating that there are tasks remaining.
Description: In the Sisyphus PSM, this mechanism is used to control the outermost loop.

**display-solution**
input: Functions functions-assigned-parts
output: The part (room) assignments that have been made
Description: This mechanism is invoked when the design problem has been completed.

**get-next-design-task**
input: Functions functions-not-assigned-parts
output; Functions function-to-assign
Description: Selects and removes the next task to perform from the set of pending tasks.

**get-next-part**
input: Functions function-to-assign
output: Part part
Description: Selects the next part from the function's domain.

**order-domain**
input: Preferences prefs
both: Functions function-to-assign
Description: Orders the parts (rooms) in the function's domain in order of decreasing preference.

## Appendix 2: the knowledge structures

The ten knowledge structures are described in the paragraphs below.

**1. Parts:** The part knowledge structure represents the elements in the part library. Parts are defined by a set of attributes and ports. The attributes of a part define the properties of a part that can be expressed by a name and a scalar value, and the ports define where it can be connected to other parts. The attributes, which are called *characteristics*, and their values, are defined before problem solving begins and cannot change during problem solving. Each part provides a *function*.

**2. Functions:** Functions define what is required of the artifact being designed for a particular problem instance. This knowledge structure is needed for the part-selection subtask of design. It drives the design process, as parts are selected to provide the functions required.

**3. Abstract Parts:** Abstract parts represent all the functions and subfunctions that an artifact being designed may perform. Abstract parts are defined by their characteristics, ports, and specifications. Specifications are attributes whose values depend upon the design problem being solved, and, therefore, their values must be computed during problem solving. This knowledge structure provides a hierarchical decomposition of the functions required of the artifact.

**4. Subfunction:** The subfunction knowledge structure successively decomposes the artifact being designed along functional lines. It describes the functional relationship between the parts and abstract parts in the domain. This relationship describes how abstract parts may be realized by combining sets of lower-level functions, which may include parts.

**5. Required Functions:** Parts and abstract parts often require functions performed by other parts to support their operation. This information is contained in the required-function knowledge structure. Associated with each function performed by a part is a list of required functions that must be realized by the artifact. The part will not realize its intended functions unless the artifact realizes the required functions. Required functions do not add any desired functionality to the artifact; rather, they perform a function that is necessary for other parts to operate.

**6. Constraints:** Constraints specify algebraic relationships among the attributes of parts and abstract parts that must be maintained. Constraints enable the problem solver to distinguish acceptable from unacceptable solutions and to compute attributes' values.

**7. Preference Knowledge:** Preference knowledge enables a design system to choose between sets of acceptable design alternatives. Preferences differ from constraints in that constraints eliminate alternatives, while preferences rank a set of acceptable alternatives so that optimal designs can be produced.

**8. Ordering Knowledge:** Problem solvers use task-ordering knowledge to determine the most effective order of designing the various abstract parts in the domain. For some problems, the order in which subtasks are performed affects both the quality of the resultant design and the speed at which the design is generated.

**9. Connection Knowledge:** Connection knowledge constrains the set of possible connections that can be made among the ports of parts and abstract parts. It may either specify illegal connections or sets of connections that have been found to be useful in the past.

**10. Arrangement Knowledge:** Arrangement knowledge specifies how parts can be geometrically or topologically arranged. It constrains the positions parts may occupy, and is used in conjunction with part boundaries.