Chapter 13
WRIGHT: A CONSTRAINT BASED SPATIAL LAYOUT SYSTEM

Can A. Baykan and Mark S. Fox

Abstract

WRIGHT formulates the problem of generating two dimensional layouts consisting of rectangular design units as a Boolean constraint satisfaction problem. Each layout is represented as a constraint satisfaction problem defined by a set of numerical variables with interval domains and algebraic constraints on them. A layout problem is defined by Boolean combinations of the algebraic constraints. Constraints are used to represent arbitrary amounts of expertise in a uniform and principled manner, and a function of texture measures, which are heuristic measures of the topological and other features of the constraint graph, controls the focus of attention during search in order to implement a fail-first strategy.

13.1. INTRODUCTION

WRIGHT is a constraint based spatial layout design system. It formulates layout problems as *constrained optimization problems* (COP) , and solves them by *constrained heuristic search* (CHS) . CHS combines constraint satisfaction with heuristic search , and adds to the definition of problem space composed of states, operators and an evaluation function, *problem textures* which are measures of problem topology that allows search to be focused in a way that reduces backtracking [13].

Spatial layout deals with the design of two dimensional configurations, such as site plans, floor plans, manufacturing facility layouts, and arrangement of

Artificial Intelligence in Engineering Design
Volume I
Design Representation and Models of Routine Design

ISBN 0-12-660561-0

equipment in rooms. In spatial layout, topological relations such as adjacency, alignment, grouping, and properties such as shape, dimension, distance, and other functions of spatial arrangement are a principal concern [8]. Spatial layout is a design task. It is an important aspect of architectural design and other fields that deal with physical design.

Design is the process of constructing a description of an artifact that satisfies a functional specification, meets explicit or implicit performance criteria, is realizable and satisfies restrictions on the design process itself [20]. There are requirements on performance in terms of time, space, energy consumption, simplicity, reliability, maintainability, fabrication and cost. These may either be specified by the client or by design codes or be implicit in established practice of good design in the field. Realizability means that the artifact conforms to limitations of the target medium, i.e. is a building that can be built by some means. It is natural to define design problems in terms of constraints. WRIGHT uses constraints to represent arbitrary amounts of expertise in a uniform and principled manner, and derives an understanding of the problem (search) space that leads to more efficient search from constraints.

For each design task, the availability of an implicitly specified set of primitive components and a set of primitive relations between the components can be assumed. For example, in electronics the primitive components are transistor, capacitor; and the relations are serial and parallel connections. In spatial layout, the primitive components can be a set of rooms with different functions, and the relations are topological and geometrical relations such as adjacency, distance and alignment.

The primitive objects are called *design units*, and the relations are called *spatial relations* in WRIGHT. Design units are rectangular shapes with discrete orientations pointing in one of the four principal directions. Spatial relations are topological or geometrical, such as adjacency, distance and overlap. The set of possible spatial relations is very large. Therefore, instead of defining a complete and fixed set, we have defined the relations that are required most often, and supplied a template for defining new spatial relations. WRIGHT formulates spatial layout as the generation of configurations of design units satisfying given spatial relations and limits on dimensions. A spatial layout problem is defined by the following inputs.

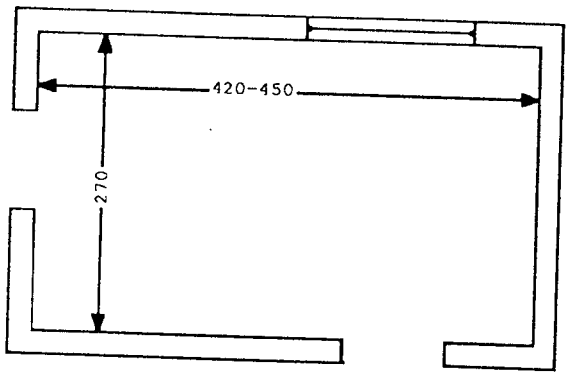


Figure 13-1: Plan Showing Initial Configuration of Kitchen

- An initial layout which may be an empty space,
- Design units to locate and/or dimension,

    1. Sink
    2. Refrigerator
    3. Range
    4. Sink center
    5. Mix center
    6. Range center
    7. Circulation area

- Constraints specifying spatial relations between design units and limits on their dimensions.

    1. Sink should be inside sink center
    2. Sink should be completely next to circulation area
    3. Sink should be facing circulation area
    4. Sink should be completely next to window
    5. Sink length $\geq$ 90 cm.

Spatial layout has the following characteristics:

- The variables under consideration such as length, width, area, and location of objects are continuous. Though dimensions and locations can be discretized using a grid, this arbitrarily eliminates some solutions.

DPS [21] and GSP [8] use drawing based representations. In GSP design un must be rectangles, and in DPS they can be arbitrary polygons. Dimensions the design units must be fixed. In drawing based systems, locations tried placing a design unit depends on the existing layou. as seen in Figure 13-3. a result of this, configurations generated depend on the order in which desi units enter the layout. Since GSP and DPS try only one ordering, they may m possible solutions. Their correctness is not guaranteed.

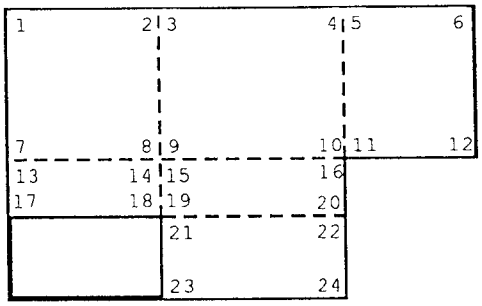


Figure 13-3: Locations Considered by GSP for Placing the Next Design U

Locations are defined by lines projected by the edges of the space and the objects that are in place. Placing an object at every location above, in four possible orientations, results in 96 new configurations.

Reprinted by permission of Elsevier Science Publishers B.V. from 'Automated Space Planning,' by C.M. Eastman, in *Artificial Intelligence* (Vol. 4; p. 57, 1973).

- Relational representations use nodes to denote points, lines, design unit some combination of these and edges to denote adjacencies between them. possible representation is an adjacency graph, where nodes denote design and edges between them denote adjacency, as in GRAMPA [15]. Another sibility is to use adjacencies between design units and the maximal lines bo ing them in an arrangement. This is called a wall-representation and is us DIS [10] and LOOS [12].

  The representation used in DIS and LOOS has two steps: determinin relational structure of an arrangement using north-of and east-of relations

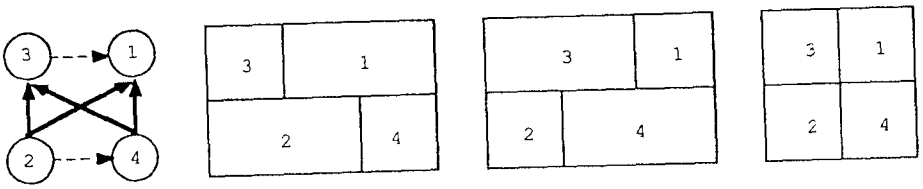


Figure 13-4: A DIS Structure and Possible Configurations Represented by it

Reprinted by permission of Pion, Ltd., London, from 'Wall Representations of Rectangular Dissections and Their Use in Automated Space Allocation,' by U. Flemming, in *Environment and Planning B* (Vol. 5; p. 225, 1978).

deriving the constraints on the dimensions of the design units based on the topology defined by the wall-representation.

Figure 13-4 shows a configuration of four design units labeled 1 to 4. The relational structure is seen at left, where north-of relation is indicated by solid arrows and east-of relation is shown by dotted lines and arrows. Design units 1 and 3 are north-of 2 and 4, 1 is east-of 3, and 4 is east-of 2. The three configurations in the same Figure show possible layouts that are represented by this relational structure. This structure gives rise to constraints on the dimensions of the design units called dependent constraints. Let x_i and y_i be the x and y-dimensions of the i^{th} design unit. The dependent constraints for the configuration seen in Figure 13-4 are $x_1 + x_3 = x_2 + x_4$, $y_1 = y_3$, $y_2 = y_4$. Required adjacencies between design units also result in dependent constraints. For example, the requirement design unit 3 must be adjacent to design unit 4 for at least L units results in $x_3 - x_2 \geq L$.

Both WRIGHT and DIS/LOOS use constraints to define an equivalence class of configurations, but WRIGHT uses constraints to define both topology and dimensions whereas DIS and LOOS use a relational structure to define topology and to derive dependent constraints. Relational systems have built in assumptions that permit only well-formed arrangements to be described. In the three relational systems above, GRAMPA, DIS and LOOS, well-formedness means that design units do not overlap. Relational systems use a restricted set of relations to describe configurations so it may not be possible to describe all aspects of a configuration we are interested in. For example, adjacency graphs do not

describe alignment or relative location such as north-of or south-of. The wall-representation does not explicitly describe alignment or adjacencies between regions using the relations but uses the dependent constraints to represent them.

WRIGHT expresses topology by algebraic relations between the lines of the design units, which is also how spatial relations *spatial relations* are defined. A configuration is represented by a CSP where the variables such as the locations and dimensions of the design units are interval variables, and the attributes of the layout such as adjacencies and distances are algebraic constraints on the variables. Alternative configurations are generated by solving a discrete CSP, where the variables are the spatial relations to be satisfied and their values are the distinct ways of satisfying them. WRIGHT employs a priority strategy, where search operators determine only the attributes specified by the selected constraint. The topology and dimensions of a configuration can be decided in any order due to the CSP formulation used.

GSP and DPS implement fail-first using domain heuristics for selecting a design unit, such as selecting the largest one or the one most strongly connected to those already located. Since DPS can deal with arbitrary polygons, it also uses a priority strategy by forming macro design units out of those that are strongly connected, and then treating it as one object. In DIS and LOOS, the order of entering the design units is given by the user.

Issues in CSP literature relevant to WRIGHT's method are balancing search and consistency methods, variable and value selection heuristics, and comparing dynamic versus fixed variable selection. REF-ARF [9] combines constraint manipulation with assigning values to variables by backtracking search. A variable is selected by first looking at the constraints which have the least number of free variables. Among that set, it attempts to use constraints which most severly restrict the values of the variables recurring in them. Constraints are mathematical equations, inequalities and disjunctions. The relations specified in constraints are ordered from most to least restrictive. Equations are assumed to be most restrictive and disjunctions least restrictive. Among those variables occurring in the most restrictive set of constraints, the one with the smallest range is selected for assigning a value at that search level. Other heuristics mentioned are selecting a variable with least number of constraints, selecting a variable with most number of constraints, and selecting a variable connected most strongly to previous variables [6]; partitioning constraint graphs into stable sets [14]. Purdom [22] determined that dynamic variable ordering during search is efficient only in problems with an exponentially small number of solutions but that require exponential search.

13.3. USER INTERFACE

Design is engaged in determining the specifications as much as in searching for solutions. In descriptive studies of design, it is observed that designers identify new constraints throughout the design process [1, 3, 7]. Thus there are two aspects to design [1, 3]:

1. creating an artifact that satisfies the constraints: *problem solving*,
2. defining or modifying a problem by identifying, refining, relaxing, and retracting constraints: *problem structuring*.

A model has been proposed by Simon [23] to account for both types of behavior. The model consists of a problem solver which operates in a well-structured problem space at any given point in time, and a noticing and evoking mechanism which modifies that problem space. The user interface is based on the premise that WRIGHT finds the solutions satisfying the set of constraints, and the designer is the problem structuring agent, even though s/he may also search for solutions.

Below is a list of possible tasks that may be carried out during design, using WRIGHT:

1. Defining new design units. It is possible to modify the hierarchy of design units in the domain, for example to define a new a type of room or appliance to be used as a primitive at some level of design.
2. Identifying new spatial relations. Some domains such as site layout or kitchen layout may require a new spatial relation in order to express desired configurations. Spatial relations are defined in terms of algebraic relations and the designer can introduce new ones.
3. Changing the set of design units in a configuration. After looking at some candidate solutions, the designer may determine that it is possible to place another bedroom in the house or a that hallway is needed.
4. Identifying new constraints. Looking at a particular configuration, the designer may identify additional constraints and need to include them in the knowledge base.
5. Relaxing constraints. When it is not possible to satisfy all the constraints, some have to be relaxed.
6. Maintaining multiple alternatives. These are pareto optimal partial solutions that are significantly different from each other.

7. Selecting a partial solution to expand.
8. Selecting an operator for generating new alternatives.

The first four operations are carried out only by the designer. Constraints, design units and spatial relations are defined declaratively and can easily be changed by the designer during the design process. The changes may become part of the knowledge base. The system carries out the last four operations, using the knowledge defined by design units, spatial relations and constraints.

Constraints specifying relations between design units at any level of the class hierarchy, including particular instances of design units, are posted, relaxed and retracted by selecting the elements from pop-up menus. The designer can interact with WRIGHT to make the layout decisions. There are commands to create, size, locate, and orient design units. The designer can define a rectangle by clicking at its top left and bottom right corners in the graphics window using the mouse. Rectangles are used to input minimum size, maximum size and bounding box of the location of objects. It is possible to think of a rectangle as a constraint, because it indicates bounds. During interactive sizing and locating operations, WRIGHT will not allow the user to violate existing bounds on a design unit. For relaxing bounds, one needs to move up in the search tree to a state where those values have not been determined yet or have looser bounds.

13.4. KNOWLEDGE-BASE

WRIGHT expresses domain knowledge using prototype design units, spatial relations and constraints. It has knowledge bases for designing kitchens, houses, manufacturing facilities and for solving bin-packing problems.

13.4.1. Design Unit Hierarchy

The taxonomy of design units in some layout domain are defined by prototype design units. These are organized hierarchically using *is-a* links. The design units used in the design of small home kitchens is seen in Figure 13-5.

Configuration knowledge is expressed as constraints on the prototypes. Constraints are inherited through the hierarchy, therefore its structure should facilitate organizing domain knowledge. A new prototype can be created and placed at the appropriate point in the hierarchy, so that it inherits constraints and

lternatives.

by the designer. Constraints
eclaratively and can easily be
ss. The changes may become
s out the last four operations,
ial relations and constraints.
a units at any level of the class
a units, are posted, relaxed and
nenus. The designer can inter-
There are commands to create,
ner can define a rectangle by
the graphics window using the
ize, maximum size and bound-
o think of a rectangle as a con-
nteractive sizing and locating
o violate existing bounds on a
move up in the search tree to a
d yet or have looser bounds.

; prototype design units, spatial
for designing kitchens, houses,
ing problems.

domain are defined by prototype
y using *is-a* links. The design
s seen in Figure 13-5.
straints on the prototypes. Con-
therefore its structure should
w prototype can be created and
so that it inherits constraints and

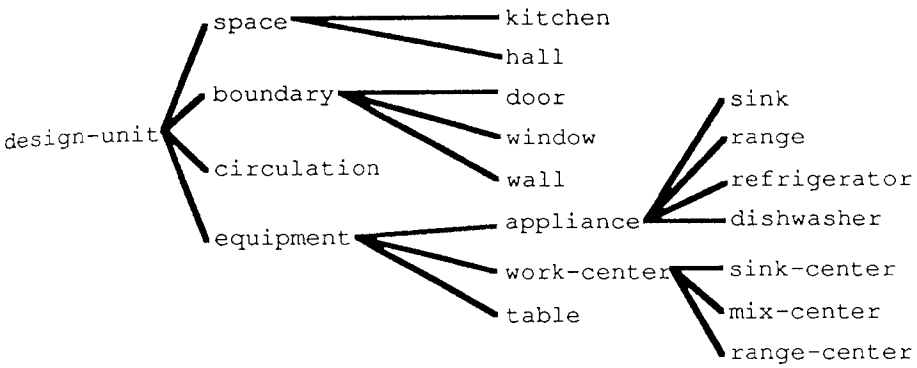


Figure 13-5: Taxonomy of Kitchen Design Units

values from above, and those below inherit from it. Inheritance of constraints eliminates duplication.

Abstraction by *aggregation* combines design units into larger design units, which are the primitive objects of configurations at another level of aggregation. A design problem may span more than one level. For example, in the design of a housing complex, the levels of aggregation are building, apartment, room, and furniture. The hierarchy in Figure 13-5 contains design units at three levels of aggregation: spaces, work centers and appliances. A kitchen contains the work centers and circulation. A sink-center may contain sink and dishwasher. WRIGHT can represent and solve spatial layout problems involving multiple levels. There is no difference in the way objects at different levels of aggregation are treated.

13.4.2. Spatial Relations and Limits on Dimensions

WRIGHT's constraints specify spatial relations between design units or limits on their dimensions. Spatial relations indicate the location of one design unit with respect to another. For example, adjacency is a spatial relation. Some spa-

tial relations are purely topological, independent of any dimensions, such as adjacency. Others such as distance involve a dimension. Some spatial relations are dependent on the orientations of the design units.

There is a very large number of possible spatial relations. Therefore in WRIGHT, we have defined a set of spatial relations with the goal of expressing the characteristics of configurations that are of interest in spatial planning. If in some domain we need to express other relationships, it is possible to define new spatial relations using a grammar defined for this purpose.

The spatial relations currently defined in WRIGHT are seen in Figure 13-6. Spatial relations are grouped in two, based on whether the orientation of the design units is considered, or whether the relations are defined with respect to the global coordinates of the configuration. *Object-centered relations* are defined with respect to the orientation of one of the design units involved. *Global relations* are defined with respect to the global coordinates, which has the y-axis pointing down and the x-axis pointing towards the right.[2]

The types of global relations are position, spatial-overlap, alignment and adjacency. Position relations indicate the location of one object with respect to another. Spatial-overlap deals with combinations of overlapping or non-overlapping of the x and/or y components of rectangles. Alignment relations specify that the north, south, east or west lines of two rectangles are equal. Global relations are seen in Figure 13-7.

Note that the relations are not mutually exclusive. For example, non-overlap, west-of, completely-next-to, and align-one-side relations can hold at the same time between two design units. Also some relations are inverses, i.e. inside is the inverse of has-inside, and east-of is the inverse of west-of. Inverses are seen under the same picture.

There are object-centered relations corresponding to all global relations except spatial-overlap. These are similar to their global counterparts except they also depend on the orientation of the first design unit. Direction relations are on the orientations of both design units. Some object-centered relations are seen in Figure 13-8.

The set of spatial relations are not fixed in WRIGHT. It is possible to define new relations by specifying the semantics of the relation using a grammar defined for this purpose.

The second group of constraint types are limits on dimensions. Limits are *greater-than*, *greater-than-or-equal*, *less-than*, *less-than-or-equal*, and *equal-to*. They are for expressing constraints on dimensions. The use of spatial relations and limits in constraints are described below.

---

[2]This is based the convention used in most graphics systems today, and is defined for ease of displaying text by starting from the origin and going from left to right in positive x direction and top to bottom in positive y direction.

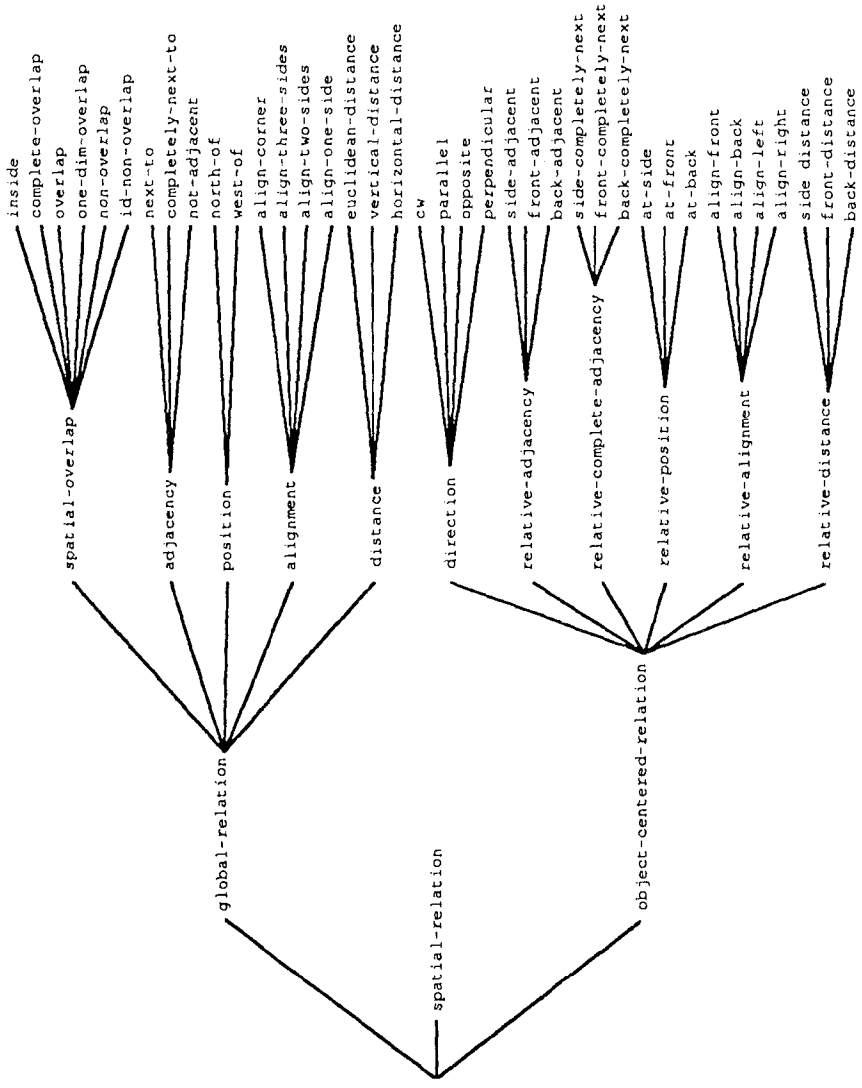


Figure 13-6: The Set of Spatial Relations in WRIGHT

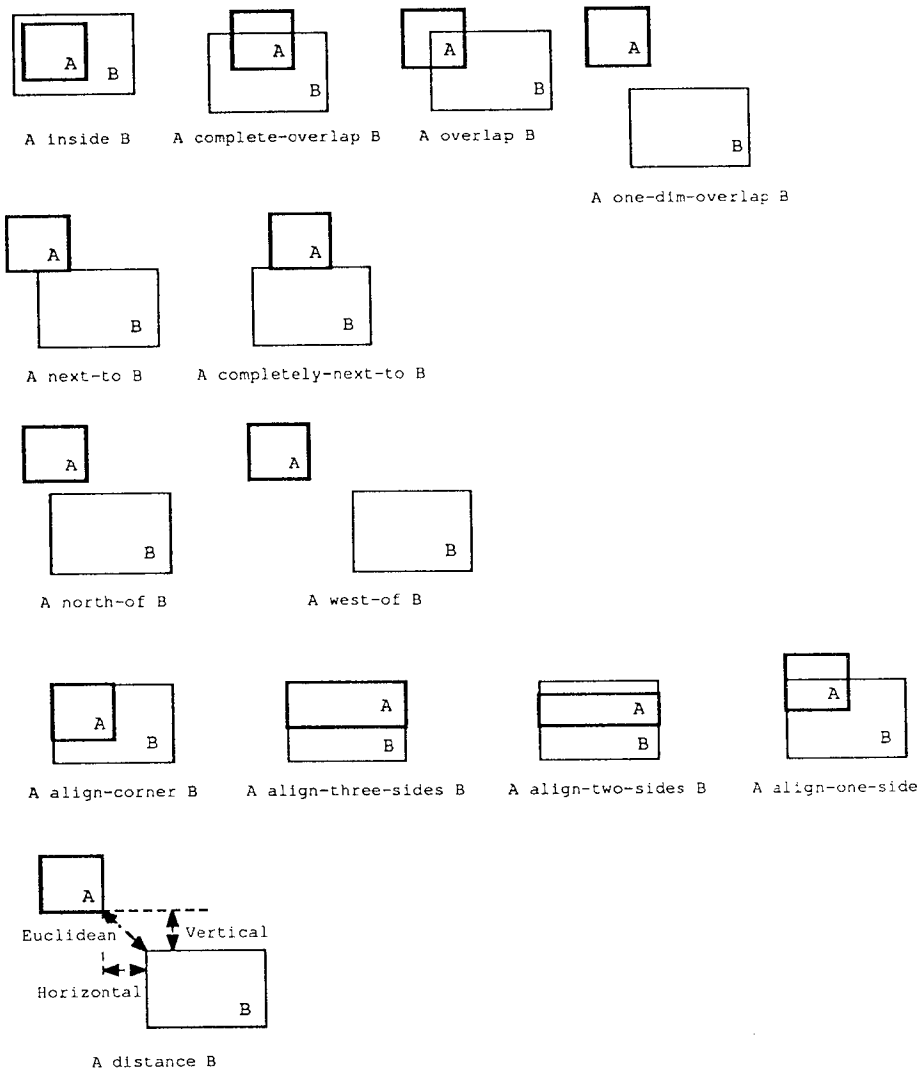


Figure 13-7: Global Relations

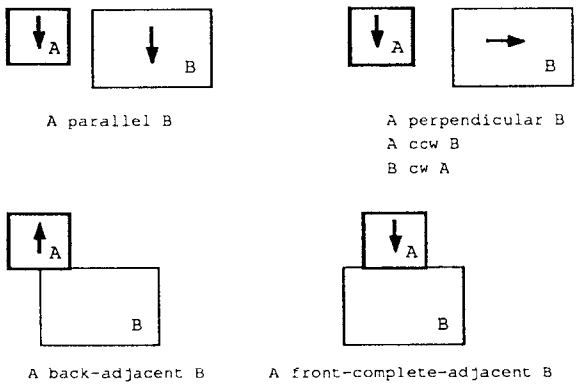


Figure 13-8: Object-centered Relations

13.4.3. Domain Constraints

Constraints express knowledge of the design domain in the form of desired relations between design units, *spatial constraints*, or limits on their dimensions, *dimensional constraints*.

Spatial constraints specify a relation between two design units. Since constraints expressing domain knowledge are posted to prototype design units, they must also contain quantifiers designating how they apply to instances. The quantifiers in WRIGHT are *all* and *some*. Some spatial relations, such as distance or next-to may require numerical values specifying a minimum or maximum. The following are spatial domain constraints:

```
All sink completely-next-to some window
All sink next-to some window ≥ 50 cm.
```

Dimensional constraints specify a design unit, a dimension of the design unit,[3] and an algebraic relation. Dimensional constraints also contain quantifiers.

```
Some sink length greater-eq 90 cm.
```

The constraint above requires that there must be at least one sink longer than 90 cm, while the constraint below requires that all sinks must be longer than 90 cm. in a layout.

```
All sink length greater-eq 90 cm.
```

The dimensional constraints above are *unary* dimensional constraints. *Binary* dimensional constraints specify a limit between two dimensions. There are no binary dimensional constraints in kitchens, so the following example is from the domain of house layout:

```
All masterbedroom area greater-than all bedroom area.
```

Every domain constraint is assigned an *importance* value between *0* and *1*, used for rating solutions. *Relaxations* are tried when a constraint can not be satisfied. Relaxation of a constraint is another constraint that specifies alternative relations, alternative design units, or looser bounds on numerical variables. Relaxations are specified explicitly, either by denoting one or more constraints as relaxations of some constraint, or by specifying that it is possible to omit the constraint, i.e., the empty relaxation. Constraints that may not be relaxed cause a configuration to be eliminated when they are violated. Relaxations have lower importance values than the constraint they relax, and an empty relaxation contributes an importance of *0*.

Design knowledge is expressed in terms of required spatial relations in WRIGHT. Consider the relationship of the sink to windows: "The average housekeeper spends nearly 1 and 1/4 hours at the sink each day so there is a good case for putting the sink at a window for good light and view." [2], p.72. One way of satisfying the requirements is placing the back of the sink completely next to the window, which is expressed by the following constraints:

- `All sink completely-next-to some window, importance=1`
- `All sink at-back some window, importance=1`

When it is not possible to put the sink completely next to the window, placing it in front of and perpendicular to the window will allow direct light and a view of

[3]Dimensional variables associated with design units are length, width and area. They are defined in the section on layout representation.

outside. The sink must also be close enough to the window. The following constraints express this case:

- All sink distance some window ≤ 120 cm., importance=0.8
- All sink one-dim-overlap some window ≥ 30 cm., importance=0.6
- `All sink perpendicular-to some window, importance=0.8`

The second set of constraints are a relaxation of the first two, and have lower importance values. Distance is measured between closest points, and one dimensional overlap means overlap in either the vertical direction or the horizontal direction.

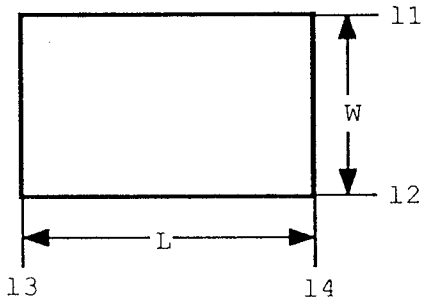
13.5. REPRESENTATION OF CONFIGURATIONS

Configurations are made up of design unit instances and algebraic constraints which define their relative positions. A design unit instance is a structured variable which consists of 8 variables: north-line, south-line, east-line, west-line, length, width, area, and orientation. North-line, south-line, east-line and west-line are the *locations* of the four lines of the rectangle. Length and width are *dimensions*, indicating distances between pairs of lines. Area is another dimension, equal to length times width. Locations and dimensions are interval variables defined by a minimum and a maximum value. For locations, the domain initially is $[-\infty, \infty]$, and for dimensions $[0, \infty]$. Orientation indicates which way the front of the design unit is facing. The domain of orientation variables is $\{0, 90, 180, 270\}$. The algebraic constraints are: $=, >, \geq, +$, and $\times$.

A design unit defines constraints between its lines, dimensions and area, as seen in Figure 13-9.

Configurations are defined by algebraic relations between variables. In Figure 13-10, the sink is south of the window, and adjacent to it for 50 cm. or longer. The algebraic relations which define this configuration are seen in the same Figure. Variables *v1* and *v2* are created for expressing the adjacent distance between sink and window.

A configuration is formed by adding relations and sometimes new variables incrementally. After each change, local propagation using interval arithmetic maintains the consistency of the layout.




```
l1 + W = l2
l1 < l2
l3 + L = l4
l3 < l4
L * W = Area
```

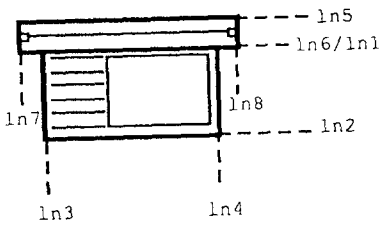
Figure 13-9: Constraining Relations Defined by a Design Unit

Reprinted by permission of Springer Verlag from 'Constraint Satisfaction Techniques for Spatial Planning,' by Can Baykan and Mark Fox, in *Intelligent CAD Systems III: Practical Experience and Evaluation* (p. 194, 1991).

13.6. CONSTRAINT COMPILER

The constraint compiler takes the prototype design units, domain constraints and spatial relations in the knowledge base and the design unit instances in a given problem, and creates a *constraint graph* which will be used for generating and testing solutions.

The constraint graph is an and/or network that refines design knowledge represented by constraints on prototype design units into a design specification represented as combinations of algebraic constraints on the components of the design unit instances.




```
ln1 = ln6
ln7 + v1 = ln4
ln3 + v2 = ln8
v1 ≥ 50
v2 ≥ 50
```

Figure 13-10: A Configuration and the Algebraic Constraints Defining It

Reprinted by permission of Springer Verlag from 'Constraint Satisfaction Techniques for Spatial Planning,' by Can Baykan and Mark Fox, in *Intelligent CAD Systems III: Practical Experience and Evaluation* (p. 194, 1991).

13.6.1. Defining Spatial Relations

The prototype design unit hierarchy, spatial relations, domain constraints, and design unit instances have been defined above. Spatial relations are defined in terms of and/or combinations of algebraic constraints on the lines of two design units. The terms used in the grammar are: *and*, *or* or algebraic constraint such as *equal-to* or *less-eq* between two components. The first component is from the design unit listed first in the constraint, and the second component is from the design unit listed second.

The definition of completely-next-to relation is seen in Figure 13-11. There are four topologically distinct ways of satisfying the completely-next-to relation. These alternatives split the domains of location variables into discontinuous intervals, defining topologically different alternatives. The mapping of the spatial relations must be defined such that the alternatives are exhaustive and mutually exclusive, because they will be used for generating solutions.

```
(*OR* (*AND* (equal-to west-line east-line)
             (less-eq south-line south-line)
             (greater-eq north-line north-line))
      (*AND* (equal-to east-line west-line)
             (less-eq south-line south-line)
             (greater-eq north-line north-line))
      (*AND* (equal-to south-line north-line)
             (less-eq east-line east-line)
             (greater-eq west-line west-line))
      (*AND* (equal-to north-line south-line)
             (less-eq east-line east-line)
             (greater-eq west-line west-line)))
```

Figure 13-11: Definition of *competely-next-to* using WRIGHT's Mapping Grammar

13.6.2. Constraint Graph

The mapping of the domain constraint

```
All sink completely-next-to some window
```

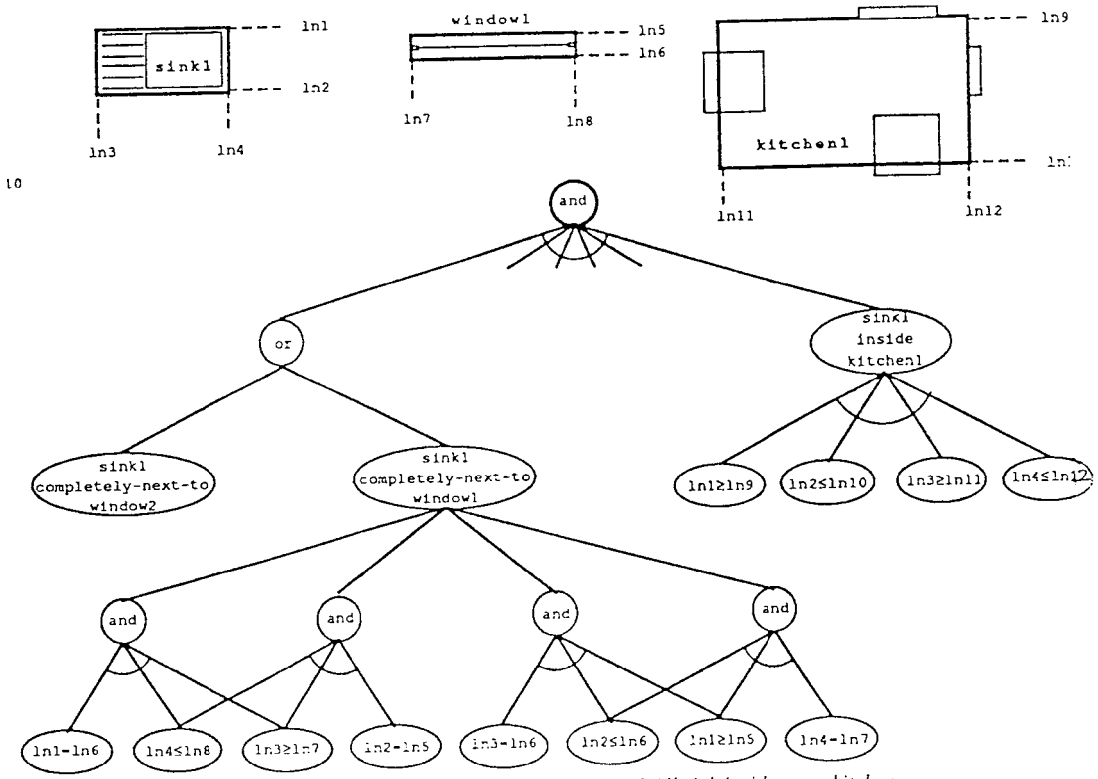
into algebraic constraints on the component lines of sink1, window1 and window2 using the definition of completely-next-to given above is seen in Figure 13-12.

A constraint graph consists of nodes and links as seen in Figure 13-12. Internal nodes are of two types: and-nodes and or-nodes. And-nodes are expressed by connecting the links leaving the node by an arc. The links in the constraint graph indicate *reliance* between constraints. Leaf nodes are algebraic constraints. The leaf nodes are shown in abbreviated form, where an algebraic constraint and the variables it connects, such as *line1* = *line2*, are represented by a single node.

The constraint graph specifies alternative ways of satisfying a constraint. Prototype design units that have more than one instance and spatial relations that can be satisfied in different ways introduce disjuncts to the constraint graph. The top level of the graph is in *conjoint normal form.*

13.6.3. Abstract Constraints

When there are conditions which hold true in all the alternatives, they can be used to bound solutions without committing to a specific alternative. These are called abstract constraints. Abstract constraints exist for adjacency and distance relations, and for dimensional constraints.



All sink completely-next-to some window and *All sink inside some kitchen* to constraints on the components of sink1, window1 and kitchen1.

Figure 13-12: Partial Constraint Graph Mapping the Constraints

Reprinted by permission of Springer Verlag from 'Constraint Satisfaction Techniques for Spatial Planning,' by Can Baykan and Mark Fox, in *Intelligent CAD Systems III: Practical Experience and Evaluation* (p. 196, 1991).

The abstract constraints for:

```
sink1 completely-next-to window1
```

where sink1 and window1 are as seen in Figure 13-12, are given below:

```
ln4 ≥ ln7
ln8 ≥ ln3
ln2 ≥ ln5
ln6 ≥ ln1
```

When it is determined that sink1 should be completely-next-to window1, these abstract constraints may be used to prune other alternatives without committing to a particular way of satisfying the sink1—window1 adjacency.

13.6.4. Formulating Spatial Layout as Constrained Optimization

A constraint satisfaction problem (CSP) [19] consists of a set of variables with predefined domains, and constraints between them. All variables and constraints are given at the start. The goal is to find one or all combinations of values that are consistent. The COP formulation of WRIGHT extends the CSP model by assigning importances to the values.

A spatial layout problem can be formulated as a CSP where the variables are the locations, dimensions and orientations. But location and dimension variables have continuous values, thus trying possible values using generate and test is infeasible. Though dimensions and locations can be discretized using a grid, this arbitrarily eliminates some solutions. Also, solutions found as a result of assigning values to interval variables will not be different from each other in significant ways.

In WRIGHT's formulation, the variables are the nodes connected to the root of the constraint graph. The values for the variables are the alternative ways of satisfying them, as given below them in the constraint graph. The consistency of the layout is ensured by keeping the interval values for locations and dimensions legal. For example, the variables in the constraint graph in Figure 13-12 are the two nodes connected to the root. The first variable has 8 alternative values. Four of them are the distinct ways of placing sink1 completely-next-to window1, and the other 4 are ways of placing sink1 completely next-to window2. The second variable has only one value as there is only one way of placing sink1 inside kitchen1.

The importance of each value is determined by the importance of the domain constraint it is derived form. When an alternative is due to a relaxation, it will have the importance of the relaxation. If a null relaxation has been specified for a constraint, it means that the variable can be removed from the COP, and not assigned a value. WRIGHT tries to find all pareto optimal solutions.

This is the dual of the problem where the variables are lines and dimensions, and the constraints are spatial relations and limits. The advantage of WRIGHT's formulation is that it becomes a discrete problem where the alternatives are structurally different.

13.7. SEARCH

WRIGHT formulates spatial layout as a COP and solves it by *constrained heuristic search*(CHS). CHS combines constraint satisfaction with heuristic search [13]. It retains heuristic search's synthetic capabilities, and adds to it the structural capabilities of constraint satisfaction. The CHS model adds *problem textures* to the definition of a problem space composed of states, operators and an evaluation function. Problem textures are based on the topology of the constraint graph and they allow search to be focused in a way that reduces backtracking.

The problem is solved by backtracking search combined with constraint propagation. Search operates by selecting a variable and assigning values to it. In this case, variables are the nodes that are connected to the root of the constraint graph, and possible values are the algebraic constraints it maps into. Satisfying the algebraic constraints removes values from the domains of numerical variables by constraint propagation. If the minimum of an interval variable becomes greater than its maximum, then the algebraic constraints are inconsistent. Reducing the domains of lines and dimensions may remove alternatives from search variables. If the range of a variable becomes empty, then that constraint is violated.

The cycle repeated in every state is

1. Select a dual variable with alternative values, using texture measures.
2. Create new states by assigning a different possible value to the variable in each state.
3. Propagate constraints, changing values of numerical variables. Test algebraic and orientation constraints, which will determine the status of nodes above them in the constraint graph. Satisfy dual variables with one remaining alternative.

The third step itself is a cycle that is repeated until quiescence. The whole cycle is seen in Figure 13-13.

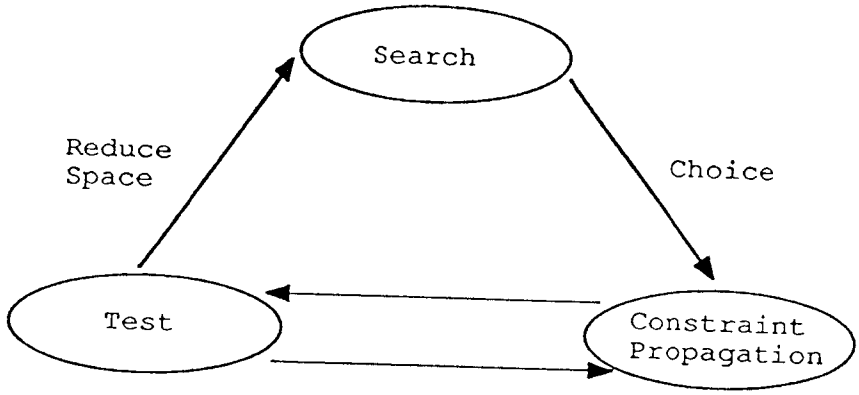


Figure 13-13: The Cycle of Operations in Every Search State

Given the constraint graph in Figure 13-12, the constraint

```
Sink1 inside kitchen1
```

will be satisfied first, because there is one way of satisfying it. The algebraic constraints at its leaves are satisfied by propagating values. Propagation will change the location of the sink so that when the active nodes in the constraint graph are checked, there remains two alternatives for the or-node, which are

1. placing sink1 south-of window1,
2. placing sink1 west-of window2.

Since it is the only active variable, search continues by trying its two values, resulting in two alternatives. At this point, all constraints are satisfied and the problem is solved. There are two equally good and significantly different solutions. When there is more than one active variable at some point in search, textures are used to select the next variable to assign values.

The search formulation described above constitutes a priority solution strategy, where operators can create macro objects or configurations in unbounded space. Textures select constraints that can be satisfied with high certainty or those most useful for simplifying search. Textures implement a fail-first and prune-early strategy.

Properties of the search architecture used in WRIGHT are

- Search is *monotonic*. States are generated by satisfying new algebraic constraints. Therefore a requirement that is satisfied can not be violated later.
- Disjuncts specified in the constraint graph are *mutually exclusive*. Therefore it is not possible to get duplicate solutions.
- Search efficiency depends on the order constraints are satisfied. Adding a set of constraints to the CSP in any order leads to the same solution.

13.7.1. Texture Measures

The philosophy behind this research is to use constraints to understand the structure of the problem space and make search efficient. The constraint graph and texture measures help in selecting an efficient ordering of variables. An efficient ordering reduces backtracking and requires assigning values to fewer variables before the values of all variables are determined.

Texture measures use two perspectives, a constraint perspective and a variable perspective. Textures using a constraint perspective look at the attributes of the constraint graph, such as the alternative ways of satisfying a constraint. Texture measures using a variable perspective evaluate constraints with respect to attributes of the variable they constrain, such as the number of active constraints on a design unit.

The heuristic implemented by textures is fail first. We try to pick a variable which will lead to fewer alternatives and which will eliminate more values from the domains of remaining variables. Since we are looking for all solutions, only variable selection heuristics are useful. Value selection heuristics do not come into play because all values of a selected variable must be tried.

The texture measures used in WRIGHT are reliance, contention and looseness. Contention uses a variable perspective, and reliance and looseness use a constraint perspective. The textures can be applied in any order and combination. They are applied lexicographically. The first texture assigns ratings to all the active variables, and eliminates those with lower values. If only one variable remains, there is no need to apply other textures. If there are more than one, the next texture is applied. If after applying all textures more than one variable remains, one is selected at random. How each texture assigns values to nodes in the constraint graph, and how these values are combined are described below.

Textures used in WRIGHT are

- *Reliance:* looks at the number of remaining values for each variable. The number of values is the number of states that will be generated at the next level if that variable is selected. This texture selects a variable with fewer values.
- *Contention:* looks at design units and as yet undetermined variables. The contention value for each design unit is the number of variables expressing a requirement for that design unit, that are not yet assigned values. The contention value for a variable is the sum of the contention values of the design units it is related to. This texture favors variables related to design units having a large number of requirements.
- *Looseness:* considers the location and dimension variables involved in each search variable and averages their domain size resulting from satisfying the relation. For example: let *l1*=[*100, 200*], *l2*=[*150, 180*], and *alt1*= *l1* ≥ *l2*. The resulting domains will be *l1*=[*150, 200*], *l2*=[*150, 180*]. The sizes of the domains are *50* and *30*, and the average is *40*. Looseness values are combined by averaging at and-nodes and taking the maximum at or-nodes. Looseness tends to favor larger design units and spatial relations which project tight locations.

The textures can be applied in any order and any combination. A variable is selected dynamically at each state, rather than fixing the order of variables before search starts.

13.7.2. Testing

Nodes in the constraint graph can have one of three values *satisfied, violated* or *undetermined.* Undetermined means that the bounds of the interval variables are so large that the constraint can be violated or satisfied depending on decisions that will be made later.

An and-node is satisfied when all of the nodes below it are satisfied. It is violated when one of the nodes below it are violated. An or-node is satisfied when one of the nodes below it is satisfied and violated when all of the nodes below it are violated. For example, if all nodes below an or-node are contradicted except one which is undetermined, the status of the or-node will be undetermined.

The structure of the constraint graph is such that every node connected to the root must be satisfied. Therefore when one of these nodes is violated, the rating of the state needs to be changed.

The result of checking an algebraic constraint is *satisfied*, *violated* or *undetermined* just as for other nodes in the constraint graph defined earlier. An algebraic constraint is satisfied when every combination of values in the domains of the variables satisfy the constraint. A constraint is violated if no combination of values in the domains of the variables satisfy the constraint. And when some combinations of values satisfy the constraint, and some don't, the constraint is undetermined. The conditions where a *greater-or-equal* constraint is satisfied, violated or undetermined is given below.

A *greater-or-equal* constraint: $[min_1, max_1] \geq [min_2, max_2]$, is satisfied when $min_1 \geq max_2$, and violated when $max_1 < min_2$. It is undetermined if $max_1 \geq min_2$ and $min_1 < max_2$.

An orientation constraint is violated if no combination of values in the domains satisfy the constraint, and satisfied when all combinations of values satisfy the constraint. For example, parallel requires two orientations to be equal. The constraint: *Orientation1 parallel orientation2* is satisfied when *Orientation1*={0} *and orientation2*={0}, undetermined when *Orientation1*={0, 90} and *orientation2*={0, 90}, and violated when *Orientation1*={0, 90} and *orientation2*={180, 270}.

Constraint propagation removes the values that can not be part of any solution from the domains of variables. Constraint propagation is a least-commitment formulation. Therefore sometimes a constraint that is satisfied transitively will not be detected as satisfied by checking values. Selecting it and propagating values will fail. Thus it is possible to also test constraints by propagation of markers and checking the existence of constraint paths. This is computationally expensive.

13.7.3. Constraint Propagation

Constraint propagation is started by selecting a new algebraic or orientation constraint to satisfy. The values of all variables in the constraint are made consistent. When the value of any variable changes, all of the constraints incident to it are used to propagate values to their variables. If during propagation, the range of an orientation becomes empty, or when the lower bound of an interval exceeds its upper bound, that means the constraint added last is inconsistent with the previous ones. How propagation is carried out for some orientation and algebraic constraints is given below.

The *v1 parallel v2* be an orientation constraint, and the domains of the variables be *v1*={0, 180} and *v2*={0, 90}. When the constraint is satisfied, {0, 180}

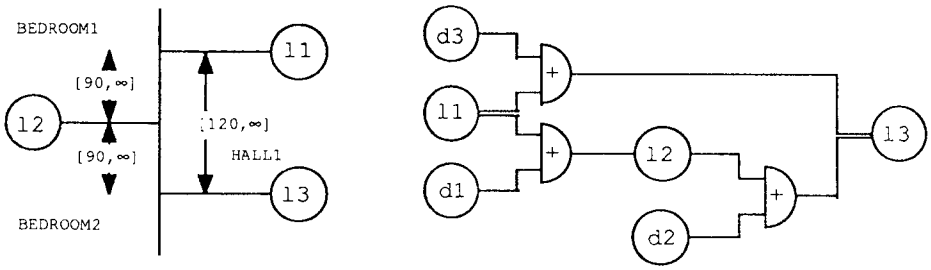


Figure 13-14: A Configuration and its Constraint Graph Containing a Loop

13.7.4. Adjacency Graph

Another reasoning mechanism is based on an adjacency graph representation. The nodes of the graph are the design unit instances and the edges denote adjacency. Edges are directed and of two types: horizontal and vertical edges. The graph representation is created at the time the constraint graph is compiled. When an adjacency constraint is created, its nodes are marked as vertical or horizontal edges. This representation is useful for two types of reasoning, as follows.

When a node corresponding to an edge is satisfied, other edges can be marked as violated and removed from consideration, based on rules about adjacency structures of rectangles. This is more efficient than checking constraints, and removes some alternatives that would not be detected by other tests but only detected during constraint propagation.

Edges have weights denoting the length of common border between the design units. The sum of weights going in to a design unit must be equal to its dimension, and must be equal to the sum of weights of the edges going out. This provides the additional constraints that maintain path consistency, when added to a configuration such as the one seen in Figure 13-14

13.8. PERFORMANCE

WRIGHT has been tested on kitchen layout, house layout and bin-packing/blocks problems. We have tried five kitchen layout problems that contain 7 design units to be located, have 2-6 solutions and approximately 80 conjunctive requirements. The house layout problem has 9 design units to locate, approximately 200 solutions and about 64 conjunctive requirements. The block problem has 6 design units, 24 and 72 solutions in its two variations, and 21 or 27 requirements.

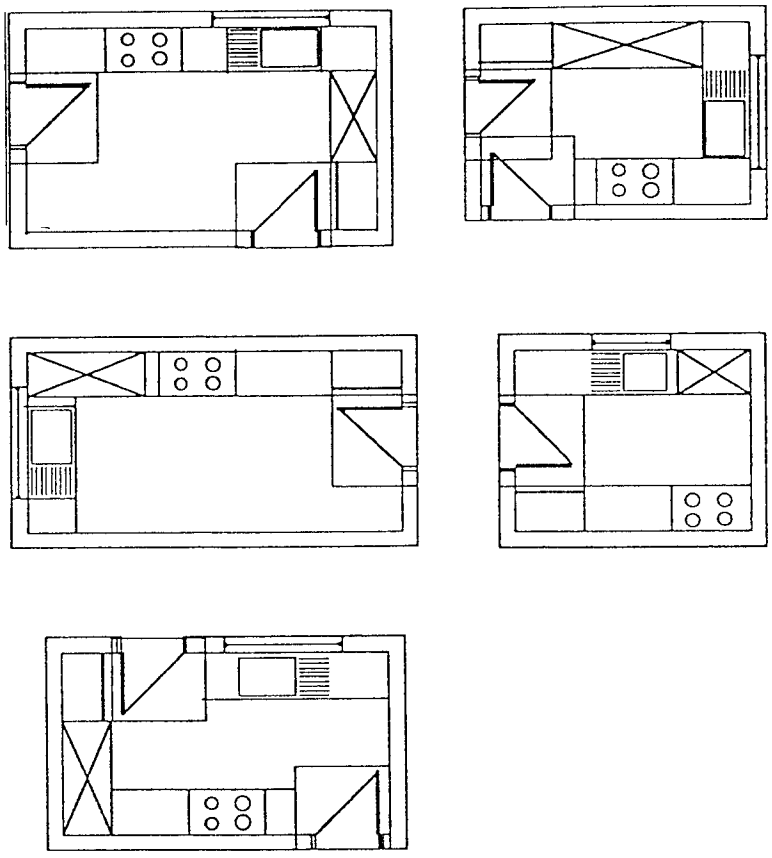


Figure 13-15: WRIGHT's Solutions to Five Kitchen Layout Problems

Adequacy of the knowledge representation and solution quality is evaluated by looking at the solutions WRIGHT generates. Five kitchens with different dimensions and door and window locations are selected from a kitchen design handbook [24]. These are configured by WRIGHT using the same domain knowledge. The solutions given by WRIGHT are compared against the solution given in [24]. WRIGHT finds the solution in the handbook in every case. One solution found by WRIGHT for each kitchen is seen in Figure 13-15. For the kitchens seen at top left and bottom, WRIGHT finds three equally good solutions, and for the kitchen seen at top right, it finds two. The design unit with the diagonals is the mix center. Sink center and range center are the rectangles containing the sink and range respectively.

Rather than applying all textures to all variables and combining the ratings, we apply textures sequentially, in order to minimize the processing time associated with dynamic selection. The first texture used assigns a rating to all active variables and removes from consideration all but the top rated ones. If more than one variable remains, the next texture is applied only to those, or if there is no other texture, one variable is selected at random.

The results of the experiments, as reported previously in [4, 13] are as follows:

1. A priority strategy is more efficient than an organize-by-design-unit strategy, leading to 50% fewer search states when solving the identical problem. Organizing by design unit forces determining all aspects at the same time, whereas priority strategy enables a least-commitment approach. Pursuing this strategy in WRIGHT is possible because of the CSP representation of configurations that enables incremental addition of constraints in any order.
2. Textures reduce search. Compared to random selection of variables, using all 3 textures reduces search states by 70% in kitchen problems and 84% in bin-packing problem.
3. The order textures are applied in has a significant effect on search efficiency. Since the first texture used eliminates most of the variables, it has the greatest effect. As a result, we have tried applying the textures in different orders and combinations. Domain size was the most useful texture in blocks problem, looseness in house layout, and contention in kitchen layout.

Figure 13-16 shows the number of search states required for finding all solutions to five kitchen layout problems, under different combinations of texture measures.

The combinations tested are

- *method 0:* select a constraint at random,
- *method 1:* contention
- *method 2:* reliance,
- *method 3:* contention and reliance,
- *method 4:* contention, reliance and looseness.

When a combination of measures is used, they are applied in the order: contention, reliance and looseness. Each measure eliminates some constraints from consideration. If more than one constraint remains after applying the texture measure(s), specified by the method, a constraint is selected at random. The number of states given for each problem-method combination is the average of three runs. In the second problem, method 4 reduces search by more than 80% compared to method 0, and in the third problem by 35%.

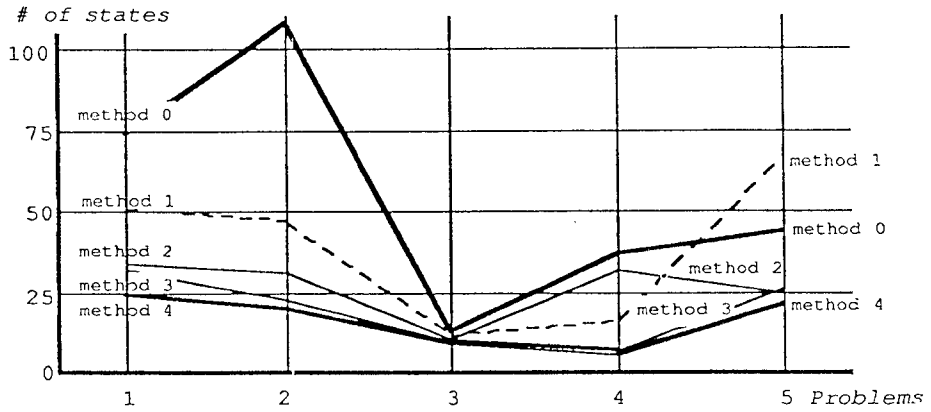


Figure 13-16: Effectiveness of Texture Measures in Reducing Search

In order to compare the CHS approach with generate and test, WRIGHT is compared with two space planning programs, DPS [21] which uses a drawing based representation, and LOOS [11] which uses a relational representation.[4]

[4]see Section 13.2 for a discussion of these programs

The problem used in the comparison is arranging six fixed size blocks in a box such that no blocks overlap. Due to the simplicity of the problem, exactly the same set of constraints can be used by all three programs. The programs are compared in terms of the number of states and search levels generated when finding the first solution and when finding all 24 solutions, seen in Figure 13-17; number of search levels is the number of intermediate states on a path from the initial state to a solution state.

	First Solution	All 24 Solutions
WRIGHT	5 levels 14 states	5-6 levels 111 states
LOOS	6 levels 68 states	6 levels 232 states
DPS	6 levels 72 states	*(not available)*

Figure 13-17: Comparison of WRIGHT, DPS and LOOS in Terms of Search Efficiency

In DPS and LOOS, the number of search levels is always equal to the number of objects to be located, as a result of the organize-by-design-unit strategy. WRIGHT'S performance in terms of number of search levels and number of search states depends on number and strength of available constraints and their interactions. Although the constraints in this problem are not as varied as in kitchen layout, WRIGHT performs better than DPS and LOOS. WRIGHT looks at a smaller number of search states by selecting decisions with fewer alternatives, and by eliminating inferior alternatives earlier.

Performance of the system depends on available constraints. Having additional constraints improves performance as they reduce the number of solutions. Once the problem becomes overconstrained, performance degrades. In order to counteract this, explicit relaxations for some constraints are given in the knowledge base.

In an underconstrained problem, DPS and LOOS find the first solution faster, but there will be a large number of solutions. WRIGHT also finds the first solution faster, and will avoid generating a large number of solutions by having solutions at a higher level of abstraction. In an overconstrained problem, DPS will not be able find any solutions because it rejects a solution that fails any con-

ix fixed size blocks in a box
of the problem, exactly the
ograms. The programs are
arch levels generated when
utions, seen in Figure 13-17;
ate states on a path from the

24 Solutions
levels 111 states
levels 232 states
(not available)

DPS and LOOS
Efficiency

always equal to the number
ize-by-design-unit strategy.
arch levels and number of
ailable constraints and their
em are not as varied as in
nd LOOS. WRIGHT looks
decisions with fewer alter-
r.
e constraints. Having ad-
educe the number of solu-
rformance degrades. In or-
constraints are given in the

ind the first solution faster,
HT also finds the first solu-
of solutions by having solu-
strained problem, DPS will
olution that fails any con-

straint. For LOOS, overconstrained problems pose the same difficulty as underconstrained ones: too many states with equivalent scores. Finding the first solution will take much longer too. Overconstrained problems will cause WRIGHT to search longer before finding the first solution. When all constraints can be satisfied, solutions are defined by alternative ways of satisfying all constraints. When all constraints can not be satisfied, combinations of constraints that result in equal ratings need to be tried. By defining explicit relaxations for some domain constraints in its knowledge base, WRIGHT avoids searching a large number of constraint combinations.

13.9. CONCLUSION

WRIGHT defines spatial planning as a constrained optimization problem and demonstrates the utility of textures and CHS. Advantages of its representation are as follows:

- Topology and dimensions are solved uniformly using algebraic constraints, and constraint propagation.
- Design units at different levels of aggregation can be handled uniformly by representing both inter-level and intra-level constraints explicitly and uniformly.
- Using constraints to guide the generation of significantly different alternatives permits solutions at a higher level of abstraction than in other layout systems, but enables determination of relevant aspects at a very detailed level.

This formulation takes a least-commitment approach by

- selecting constraints to satisfy rather than locations for design units, and
- removing from variable domains only those values which violate a constraint.

The abstraction mechanisms it makes possible are

- abstraction by aggregation, and
- abstract constraints.

The philosophy behind this approach is understanding the structure of the search space to make search efficient. Important points about WRIGHT's approach to search efficiency are

- Constraint propagation techniques dramatically narrow the space of alternative solutions prior to selection/search.
- Properties of the constraint network, known here as *textures*, can be used to focus attention of search (i.e., node and value selection), thereby reducing the amount of backtracking.
    - Contention selects a design unit which has a large number of conjunctive constraints remaining.
    - Reliance chooses to satisfy a constraint for which there are fewer alternative disjunctive decisions.
    - Looseness chooses to satisfy a constraint which reduces range of variables more.
- Both domain independent and dependent knowledge is represented uniformly as constraints thereby enabling the alteration of search behavior and the solutions generated by the search alteration or addition of constraints.

13.10. ACKNOWLEDGMENTS

Work on WRIGHT has been supported by a grant from Digital Equipment Corporation.

13.11. BIBLIOGRAPHY

[1] Akin O., Dave B., and Pithavadian S., *A Paradigm for Problem Structuring in Design*, unpublished working paper, September 1987, [Department of Architecture, Carnegie Mellon University].

[2] Architects Journal, "Domestic Kitchen Design: Conventional Planning," *Architects Journal*, pp. 71-78, 3 October 1984.

[3] Baykan, C.A., *Heuristic Methods for Structuring Architectural Design Problems*, unpublished working paper, 1984.

[4] Baykan, C. and Fox, M.S., "Constraint Satisfaction Techniques for Spatial Planning," *Preliminary Proceedings of the Third Eurographics Workshop on Intelligent CAD Systems*, CWI, Amsterdam, pp. 211-227, 1989.

[5] Davis, E., "Constraint Propagation with Interval Labels," *AI*, Vol. 32, No. 3, pp. 281-331, July 1987.

[6] Dechter, R. and Pearl, J., "Tree Clustering for Constraint Networks," *AI*, Vol. 38, pp. 353-366, 1989.

[7] Eastman, C.M., "On the Analysis of Intuitive Design Processes," in *Emerging Methods in Environmental Design and Planning*, Moore, Gary T., Ed., MIT Press, Cambridge,Mass., 1970.

[8] Eastman, C.M., "Automated Space Planning," *AI*, Vol. 4, pp. 41-64, 1973.

[9] Fikes, R.E., "REF-ARF: A System for Solving Problems Stated as Procedures," *AI*, Vol. 1, pp. 27-120, 1970.

[10] Flemming, U., "Wall Representations of Rectangular Dissections and their Use in Automated Space Allocation," *Environment and Planning B*, Vol. 5, pp. 215-232, 1978.

[11] Flemming, U., *On the Representation and Generation of Loosely Packed Arrangements of Rectangles*, Technical Report DRC-48-05-85, Carnegie-Mellon University Design Research Center, 1985.

[12] Flemming, U., "On the Representation and Generation of Loosely Packed Arrangements of Rectangles," *Environment and Planning B*, Vol. 13, pp. 189-205, 1986.

[13] Fox, M.S., Sadeh, N., and Baykan, C., "Constrained Heuristic Search," *Proceedings of IJCAI-11*, IJCAI, pp. 309-315, 1989.

[14] Freuder, E.C. and Quinn, M.J., "Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems," *Proc. IJCAI-9*, IJCAI, pp. 1076-1078, 1985.

[15] Grason, J., *Methods for the Computer-implemented Solution of a Class of Floor Plan Design Problems,* unpublished Ph.D. Dissertation, Carnegie-Mellon University, May 1970.

[16] Koopmans, J.C., Beckmann, M.J., "Assignment Problems and the Location of Economic Activities," *Econometrica*, Vol. 25, pp. 53-76, 1957.

[17] Liggett, R.S., "The Quadratic Assignment Problem: An Analysis of Applications and Solution Strategies," *Environment and Planning B,* Vol. 7, pp. 141-162, 1980.

[18] Mackworth, A.K., "Consistency in Networks of Relations," *AI,* Vol. 8, pp. 99-118, 1977.

[19] Mackworth, A.K., and Freuder, E.C., "The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *AI,* Vol. 25, pp. 65-74, 1985.

[20] Mostow, J., "Toward Better Models of the Design Process," *AI Magazine,* Vol. 6, No. 1, pp. 44-57, 1985.

[21] Pfeffercorn, C., *Computer Design of Equipment Layouts Using the Design Problem Solver,* unpublished Ph.D. Dissertation, Carnegie-Mellon University, May 1971.

[22] Purdom, P.W., "Search Rearrangement Backtracking and Polynomial Average Time," *AI,* Vol. 21, pp. 117-133, 1983.

[23] Simon, H.A., "Structure of Ill-structured Problems," *AI,* Vol. 4, pp. 181-201, 1973.

[24] Small Homes Council, *Handbook of Kitchen Design,* University of Illinois, Urbana, Illinois, 1950, [Circular C5.32R].