Camera-ready copy submitted to AI in Design '92
(is in: "/../yeti/usr/cab/aid92/text.ps")

HIERARCHICAL GENERATE-AND-TEST vs. CONSTRAINT-DIRECTED SEARCH

A Comparison in the Context of Layout Synthesis

ULRICH FLEMMING
Department of Architecture and Engineering Design Research Center
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

CAN A. BAYKAN
The Robotics Institute
Carnegie Mellon University

ROBERT F. COYNE
Department of Architecture and Engineering Design Research Center
Carnegie Mellon University

MARK S. FOX
Department of Industrial Engineering
University of Toronto

Abstract. Two systems for layout synthesis, LOOS and WRIGHT, and the approaches underlying them are compared. LOOS uses a form of hierarchical generate-and-test and WRIGHT disjunctive constraint satisfaction, a form of constraint-directed search. LOOS implements a constructive approach that adds objects sequentially, while WRIGHT uses a reductionist approach that satisfies constraints incrementally. The comparisons are based on a series of experiments in which the systems were used to solve identical layout problems and to produce insights at very detailed levels. The conclusions are tentative, as the experiments are still going on at the time of writing.

1. Introduction

The present paper compares two space planning or layout synthesis systems, LOOS and WRIGHT, each of which implements a well-known approach toward solving design problems defined through feasibility constraints. The approach underlying LOOS is a form of *hierarchical generate-and-test*, in which solutions are constructed incrementally and intermediate states evaluated for constraint satisfaction; the overall control is based on these evaluations. WRIGHT implements a form of constraint-directed search called *disjunctive constraint satisfaction*, in which the constraints are incrementally satisfied.

Each approach has been independently implemented in a conceptually clean and clear fashion, and the resulting systems are able to solve *identical layout problems* from various domains. The authors seized the opportunities thus offered and conducted a series of experiments in order to gain concrete insights into the advantages and disadvantages of the underlying approaches that go beyond summary characterizations that dismiss, for example, the first approach as generally inefficient.

The present paper reports some initial results of these experiments. Section 2 characterizes the layout problems solved by the two systems. Sections 3 and 4 briefly describe the systems, and section 5 presents the results of the experiments. Our conclusions are summarized in Section 6.

2. Layout Problems

The layouts considered in this paper are arrangements of rectangles with sides parallel to the axes of an orthogonal system of Cartesian coordinates. The rectangles in a layout can be *loosely packed*; that is, the layout may have holes or an irregular boundary. This class of layouts is interesting across a broad spectrum of applications, domains and disciplines that range from digital and analog electronics design to building and graphic design.

A layout in this class is completely specified if the corner coordinates (or some equivalent set of values) are given for each rectangle in the layout. The problem of finding a feasible set of coordinates gains a considerable degree of complexity from the fact that values for the coordinates cannot be selected independently of each other. For example, the rectangles in a layout often represent physical objects that occupy space and therefore cannot overlap. The overall area available for placing the objects is also often restricted. This creates constraints that may vary with the way in which the objects are placed; an example is shown in Figure 1. These types of constraints have been called in the literature *dependent* (Flemming, 1978) or or *inter-element* (Eastman, 1973) constraints; they indicate that considerations of *structure* or *topology* and discrete decisions about structural or topological variables play a prominent role in layout synthesis, as they do in other design domains dealing with assemblies of discrete parts in 2- or 3-dimensional space.

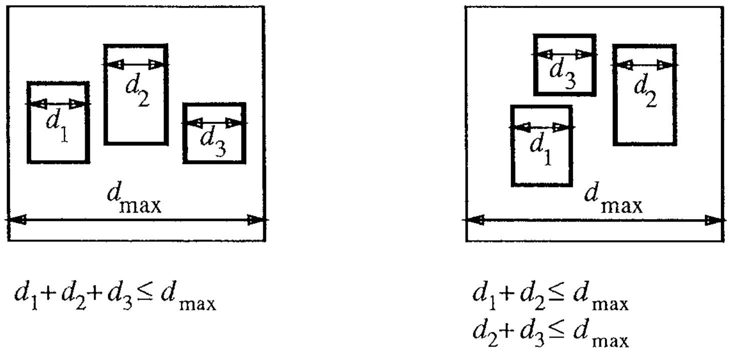


Figure 1. Examples of dependent constraints

In addition to the dependent constraints, layouts must satisfy constraints or criteria that are independent of a particular structure. Examples are constraints on the dimensions, area or orientation of an object and required or desired relations between objects (such as adjacency, proximity or physical access).

The general layout problem solved by the two systems can be summarized as follows: Given a set of objects to be allocated and a set of constraints on the shape and placement of these objects, find one or more layouts that satisfy the constraints. The objects to be allocated are called *design units* in the following. Table 1 specifies a very simple layout problem, which will serve as an illustration in succeeding sections. It calls for the design of an efficiency apartment within an area that is accessed from the east and receives natural light from the west [the example is taken from (Flemming, 1979)].

Table 1. Layout Problem 1

Spaces:	Living/sleeping area	Min. dimension	3.60 m
		Min. area	22.00 m^2
	Kitchenette	Min. dimension	1.80 m
		Min. area	4.20 m^2
	Vestibule or hall	Min. dimension	1.20 m
		Max. dimension	6.00 m
	Bathroom	Min. dimension	1.80 m
	Max. extent of overall area from west to east:		7.00 m

Required adjacencies (min. length of shared boundary in brackets)

Living area/vestibule (.90 m)	Living area/western border (3.60 m)
Living area/kitchenette (1.20 m)	Living area/southern border (3.60 m)
Vestibule/eastern border (1.20 m)	Vestibule/bathroom (.70 m)

In solving problems of this kind, both LOOS and WRIGHT perform state-space-search. The principal differences between the two systems stem from contrasting ways in which they represent and handle discrete decisions about structural variables, which result in contrasting ways of setting up and traversing the state space. The following sections briefly describe the two systems. Space limitations prevent us from giving more elaborate descriptions; readers interested in more details are referred to (Flemming, 1988, Flemming et al., 1989) for LOOS and (Baykan, 1991, Baykan & Fox, 1991) for WRIGHT.

3. LOOS

3.1. DESIGN VARIABLES

If given a set of design units, LOOS attempts to find feasible layouts of rectangles in which each rectangle represents one of the design units and no two rectangles overlap. It ultimately tries to determine for each rectangle r values x_r, X_r, y_r, Y_r, which can be interpreted as the coordinates of its corner points or as defining the four lines that lie on the boundary of r (see Figure 2).

The variables handled directly by LOOS are the spatial relations *above, below, to the left* and *to the right*, which are defined as follows: If q and r are two rectangles, then

q is *above* $r \iff y_q \geq Y_r$ $\quad$ r is *below* $q \iff q$ is *above* r

q is *to the right of* $r \iff x_q \geq X_r$ $\quad$ r is *to the left of* $q \iff q$ is *to the right of* r

Clearly, q and r *do not overlap* iff at least one of the spatial relations holds between them. But since each of the relations is non-reflexive, non-symmetric and transitive, they cannot be selected independently of each other. We call a

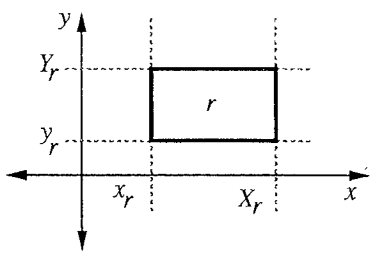


Figure 2. Basic design variables for a rectangle

set of relations that can be simultaneously realized and guarantee non-overlap for each pair of rectangles in a layout a *spatial structure.* The representation of spatial structures used by LOOS is derived from the *wall representation* of rectangular dissections (Flemming, 1978). A rectangular dissection is a layout of rectangles that completely fill the area of a larger rectangle without overlap and holes; an example is shown in Figure 3a. A *wall* in such a configuration is a maximal sequence of connected, collinear line segments separating the rectangles from each other (Figure 3a highlights one such wall). A *wall representation* of a rectangular dissection records all of its walls and the sequence of rectangles bordering each wall from above and below or from the left and right.

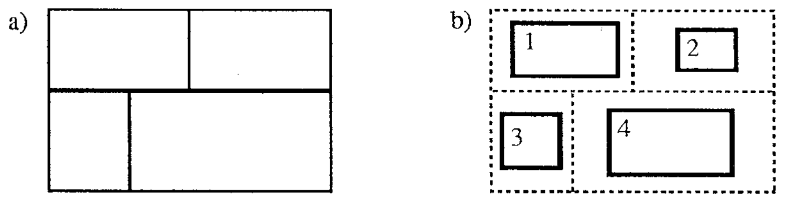


Figure 3. (a) a rectangular dissection and its walls; (b) a loosely packed layout with the same spatial structure

Each wall implies left/right or above/below relations between pairs of rectangles on opposing sides of the wall. Figure 3b demonstrates that the spatial relations implied by the walls of a rectangular dissection can be found also in loosely packed arrangements; for example, rectangles 1 and 2 are above both rectangles 3 and 4. Wall representations, or their equivalents, can consequently also be used to represent the spatial structure of loosely packed layouts if these layouts can be derived from a rectangular dissection by shrinking some rectangles or, conversely, if the layout can be turned into a rectangular dissection by expanding all rectangles until they touch other rectangles on all four sides. Since walls lose their significance in a loosely packed arrangement, we call the gaps separating rectangles from each other *channels*, following the terminology introduced by VLSI designers when they adapted the wall representation to their purposes (Supowit and Slutz, 1984). We indicate channels in a layout by dashed lines as shown in Figure 3b, which also makes the underlying spatial structure immediately recognizable.

The spatial structure of a loosely packed layout cannot be represented directly by a wall representation when the layout contains *non-trivial holes*, which are holes that cannot be eliminated by extending the rectangles in an arrangement until they touch other rectangles (Flemming, 1989); an example is shown in Figure 4a. LOOS circumvents this difficulty by representing non-trivial holes explicitly as rectangles that are marked by a special label to distinguish them from regular rectangles (see Figure 4b). The resulting *marked structures* are able to represent any spatial structure (Flemming, 1989) and form the basis for LOOS, which represents marked structures internally as directed graphs.

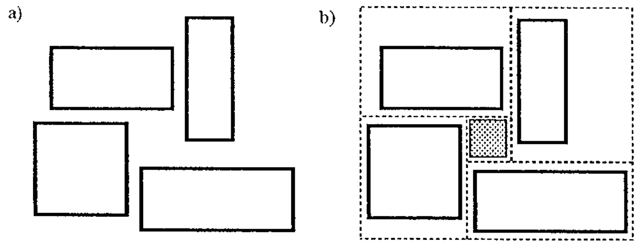


Figure 4. (a) A layout containing a non-trivial hole;
(b) representation of a non-trivial hole by a special rectangle (shown hatched)

In addition to the spatial relations holding between the rectangles it contains, the LOOS representation records explicitly lower and upper bounds for the coordinates of each rectangle; we call the area defined by these values the (dimensional) *range* of the rectangle. Figure 5 depicts the range of one rectangle in a spatial structure with the rectangle drawn in the center of its range. LOOS uses *slacks* to indicate how much a rectangle can move in the x- or y-direction within its range. The range and slacks are called the *dimensional attributes* of the rectangle. We store these attributes for each rectangle in a marked structure and call the resulting representation a *configuration*. Clearly, a configuration containing some positive slacks represents not a single layout, but a class of layouts because some rectangles can have several, if not infinitely many positions or dimensions, and every combination of these variations defines a different layout.

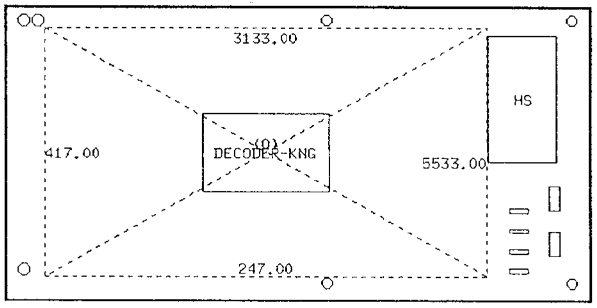


Figure 5. The dimensional range of a rectangle in a configuration

3.2. GENERATION AND PROPAGATION RULES

LOOS constructs and evaluates configurations by using operators or rules that work on configurations. These rules are described in this and the following section.

Generation rules generate marked structures from marked structures by adding one rectangle at a time, starting with a suitable initial structure which may represent nothing more than the enclosing rectangle or a configuration of preplaced objects. Alternative structures are generated if more than one possibility for adding individual rectangles is pursued. This incremental construction allows for intermediate evaluations that can be used for pruning. Figure 6 shows how alternative layouts can be incrementally constructed by successive rule applications, including the insertion of non-trivial holes.

The dimensional attributes for a newly inserted rectangle depend on those of the surrounding rectangles. Conversely, the dimensional bounds of the surrounding rectangles are likely to become tighter through the insertion. LOOS uses *propagation rules* to compute the dimensional attributes for the newly inserted rectangle and to propagate the resulting changes recursively through the structure. The propagation rules can handle rectangles with fixed or variable dimensions. They are also able to take different orientations for a rectangle into account (for example, in terms of its front and back).

3.3. TEST RULES

LOOS is able to evaluate a configuration by application of *test rules*. Each of these rules checks if a favorable or unfavorable condition exists and takes an appropriate action. For example, if a test rule discovers that a particular constraint is violated, it writes an appropriate entry into an *evaluation record* of the configuration; examples of such failing tests are shown in Figure 6.

Test rules can also estimate how well a configuration performs with respect to true criteria (that is, performance aspects that are measured on some sort of scale and differ from constraints which are either satisfied or not). An example is the minimum size of the overall area, for which LOOS is able to estimate lower bounds at any state.

Any aspect that can be evaluated based on the information contained in a configuration can be incorporated into a test rule, and since any configuration produced by a generation rule represents a formally complete layout of rectangles, test rules can be applied not only to terminal, but also to intermediate configurations that do not contain all of the design units in a given layout problem. The only restriction is that certain aspects can be evaluated *with certainty* only if all units have been allocated. An example is a forbidden adjacency between two units, which may exist in an intermediate state, but disappear with the placement of additional units.

The design units in a specific problem are instances of prototypes and inherit constraints from them. Domain knowledge about these prototypes is stored in a hierarchy that must be constructed for each application domain.

3.4. OVERALL ARCHITECTURE AND CONTROL

The original intent behind LOOS was to create a complement to human designers in the form of a system able to *systematically* enumerate solutions to layout problems characterized by diverse and possibly conflicting criteria or constraints. A specific goal was the generation of alternatives with interesting trade-offs in terms of these criteria. The architecture of LOOS reflects this goal and implements a hierarchical generate-and-test (HGT) approach (Stefik et al., 1983). HGT uses intermediate evaluations to guide the search for solutions into promising directions and to avoid the inefficiencies associated with blind generate-and-test. LOOS has strong similarities with and was inspired by DENDRAL, a system able to find chemical structures that are likely to produce a given mass spectrogram (Buchanan et al., 1969).

LOOS comprises five major components:

1. A *preprocessor* that accepts a problem description from the user and performs some initial computations. If some units are preplaced, for example, the preprocessor must construct a starting configuration that describes the spatial relations between these units.
2. A *generator* that accepts any configuration and is able to find all possible ways of adding a new rectangle. It applies the generation and propagation rules under a (rather complicated) control strategy which assures that the spatial relations between already allocated rectangles remain unchanged. This *monotonicity of the spatial relations during generation* makes it possible to evaluate and consequently prune intermediate states of the search space with certainty because constraints that are structure-dependent and not satisfied by an intermediate state cannot be satisfied by a configuration generated from it (exceptions are the constraints mentioned in the previous section that depend on complete configurations).
3. A *tester* that applies the test rules sequentially to evaluate any intermediate or terminal state according to domain-specific constraints or criteria. It is built and works very much like a diagnostic expert system.
4. A *controller* that mediates between generator and tester. After each expansion, it passes the new states to the tester for evaluation; inspects the test results; and terminates the search or selects a new state for expansion and passes it to the generator. In making its decisions, the controller follows a straight-forward branch-and-bound strategy; that is, it selects those and only those states for expansion that have currently the best record. The constraints and criteria are classified as strong, intermediate or weak. The controller counts the number of constraints violated in each class and selects states with the lowest counts, where the counts are ranked lexicographically over the constraint classes.
5. A *postprocessor* that finetunes the solutions thus produced. For example, it may inspect the ranges of rectangles with positive slacks in each terminal state and determine final values for the dimensional coordinates; that is, it selects a specific instance from among the layouts represented by the state. This selection can be the result of some form of optimization, for example, minimization of the total area occupied by the rectangles.

LOOS is able to eliminate certain states *before* they are generated. Even the earliest versions of the generator computed the dimensional range for the new rectangle for each possible expansion and executed only those that could accommodate the new object; that is, all layouts generated could at least be physically realized. Another pregeneration test that we added in the early stages of the system checks whether the insertion under consideration would interrupt a *hard arc*, an arc in the graph representing the spatial structure of the current configuration that has been declared unbreakable; that is, the direct spatial relation represented by that arc between two rectangles cannot be interrupted by putting another rectangle inbetween. Hard arcs can be established by the tester when checking for satisfaction of adjacency constraints and can be effective in preventing the generation of flawed configurations.

More recently, we have added the capability for additional pregeneration tests whose execution remains optional. These tests are restricted at the present time to desired topological properties such as adjacencies including those with the exterior. LOOS is able to determine if these adjacencies are possible *after* a particular application of a generation rule *before* it is applied. We call this mode *constrained generation.* It does not eliminate the need for tests after generation because the pregeneration tests currently performed by LOOS are not comprehensive; in particular, they do not consider constraints that govern the placement of previously allocated rectangles (except for those indicated by hard arcs).

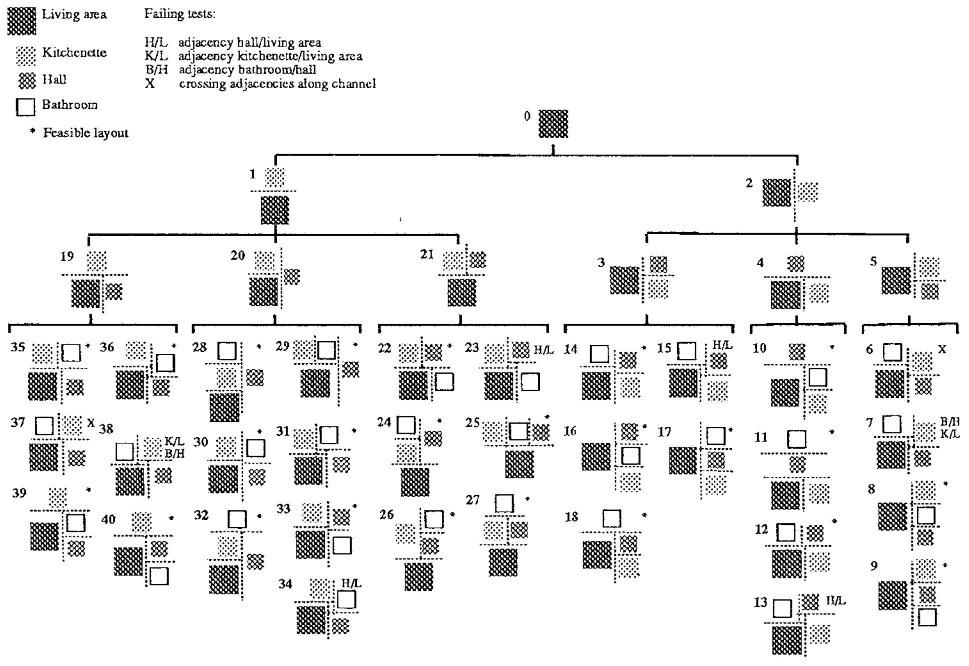


Figure 6. State space for Problem 1 as generated by LOOS

Figure 6 gives a complete trace of how LOOS solves Problem 1 as specified in Table 1 with hard arcs set and pretesting enabled for all required adjacencies. LOOS finds 24 feasible solutions and needs 40 states to find them. The states are numbered in the figure in the order in which they are generated. The figure also lists the constraints violated by a state. Since aside from dimensional constraints, required adjacencies are the only constraints specified for the problem, very few infeasible states are generated: pretesting prevents adjacencies that are satisfied in a state from being interrupted by insertion of a new object (hard arcs) and guarantees that all the adjacencies required for the new object are satisfied after insertion. Exceptions occur when a non-trivial hole is inserted, in which case the current generator does not pretest for hard arcs, but may interrupt adjacencies required for objects already placed (e.g configuration 7). The only other constraint violation occurs when required adjacencies cross each other along a channel and thus cannot be simultaneously satisfied (configurations 6 and 37); the rules that test for required adjacencies always check for this condition, but only *after* generation.

4. WRIGHT

4.1. DESIGN VARIABLES

WRIGHT represents a layout using algebraic equations and inequalities in variables that represent the border lines, dimensions, areas and orientations of the design units. A design unit r is defined by north, south, east and west *lines*. The north and south lines are horizontal, and their values are the coordinates Y_r and y_r; the east and west lines are vertical, and their values are the coordinates X_r and x_r (see Figure 2). A *dimension* is the distance between any two parallel lines or the area of a design unit. The variables in Problem 1 are listed in Table 2 for further reference. The domains of the variables are closed intervals, defined by a minimum and a maximum value.

Table 2. Design units and variables in Problem 1 as defined by WRIGHT

Design unit	***north-ln***	***south-ln***	***west-ln***	***east-ln***	***xdim***	***ydim***	***area***
Apartment	*apN*	*apS*	*apW*	*apE*			
Living room	*lrN*	*lrS*	*lrW*	*lrE*	*lrX*	*lrY*	*lrA*
Vestibule	*vbN*	*vbS*	*vbW*	*vbE*	*vbX*	*vbY*	
Kitchen	*ktN*	*ktS*	*ktW*	*ktE*	*ktX*	*ktY*	*ktA*
Bathroom	*btN*	*btS*	*btW*	*btE*	*btX*	*btY*	

4.2. ATOMIC CONSTRAINTS

The algebraic equations and inequalities that define a layout are called *atomic constraints*. For example, the absolute location of a line is expressed as a binary constraint between the line and a constant. If *apW* is a vertical line, and $100 \leq apW \leq 150$, the location of *apW* defined by this constraint is the grey area in Figure 7. Topology and alignment are expressed by $>$, $\geq$ and $=$ relations between two lines. The coordinate system used has the x-axis pointing to the right and the y-axis pointing down.

Figure 8 shows the atomic constraints defining the initial state of Problem 1. These constraints define the relationships between variables belonging to the same design unit. Any change in the bounds of a variable is propagated to the others via the constraints linking them; these relationships are thus maintained in all configurations.

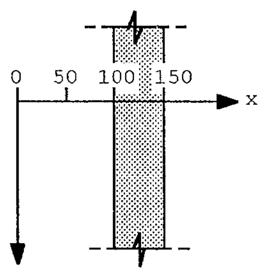


Figure 7. Location of *apW* defined by $apW \in [100, 150]$

$lrN + lrY = lrS$	$lrW + lrX = lrE$
$vbN + vbY = vbS$	$vbW + vbX = vbE$
$ktN + ktY = ktS$	$ktW + ktX = ktE$
$btN + btY = btS$	$btW + btX = btE$
$lrX \times lrY = lrA$	$ktX \times ktY = ktA$

Figure 8. Atomic constraints defining the initial state of Problem 1

$vbW >= ktE$	$ktS > vbN$	$vbE = apE$
$btS = vbN$	$ktS = lrN$	$lrE = vbW$
$lrW = apW$	$lrS = apS$	$btN = apN$
$btE = apE$	$btW = ktE$	$ktN = apN$
$ktW = apW$	$vbS = apS$	

Figure 9. Atomic constraints defining a solution to Problem 1

WRIGHT constructs solutions by asserting atomic constraints. For example, the configuration in state 25 in Figure 12 is defined by adding the constraints given in Figure 9 to the initial state. After adding some constraints, propagation updates variable domains and checks consistency of the constraint set.

A finite set of variables $V = \{v_1, v_2, ..., v_m\}$, each with an associated domain of values, and a set of atomic constraints in these variables $A = \{c_1, c_2, ..., c_n\}$ define a *constraint satisfaction problem* [CSP]. In WRIGHT, every configuration and search state is a CSP. The CSP is consistent if there exist values for all variables that simultaneously satisfy all constraints. During propagation, lower bounds can only increase and upper bounds can only decrease; that is, propagation behaves monotonically. An inconsistency is detected if the upper bound for some variable becomes less than its lower bound. The propagation algorithm used by WRIGHT while running the experiments discussed in this paper is *path-consistency* (Mackworth, 1977).

4.3. DISJUNCTIVE CONSTRAINTS

Design in general and intelligent CAD require a fundamental problem-solving methodology that is able to incorporate arbitrary amounts of knowledge in a principled manner. WRIGHT uses *disjunctive constraint satisfaction* to this end. It provides a formal method for representing expertise uniformly and declaratively in terms of disjunctive and conjunctive combinations of atomic constraints and selects efficient search strategies based on topological and other features of the constraints.

A *disjunctive constraint* is a Boolean combination of atomic constraints. The canonical form of a disjunctive constraint is defined to be its *disjunctive normal*: the top level elements, called *disjuncts*, are connected by an *or* ($\vee$); the second level elements, which are atomic constraints, by an *and* ($\wedge$); and there are at most two levels. Thus, a disjunctive constraint C_i has the form

$C_i = (d_{i1} \vee d_{i2} \vee \ldots \vee d_{ik(i)})$,

and each disjunct d_j the form

$d_j = (c_{j1} \wedge c_{j2} \wedge \ldots \wedge c_{jk(j)})$.

A disjunctive constraint can consist of a single disjunct, and a disjunct can consist of a single atomic constraint.

Each disjunct defines a partial configuration that satisfies a disjunctive constraint in a significantly different way, and the disjuncts, taken together, specify the *structural alternatives considered by* WRIGHT *for satisfying the constraint.* Consider for example two design units that must be adjacent. This can be achieved by placing the first design unit to the north, south, east or west of the second; thus an adjacency requirement can be represented by a disjunctive constraint with four disjuncts. Some requirements of Problem 1 are formulated as disjunctive constraints in Figure 10. *PC-23* formulates the four alternatives of placing the bathroom adjacent to hall. The requirement that the bathroom should be inside the apartment is expressed by *PC-12*, which has a single disjunct.

The requirements which define a problem form a set of disjunctive constraints, $D = \{C_1, C_2, \ldots, C_p\}$, all of which have to be satisfied by a solution. The resulting problem is called a *disjunctive CSP* (DCSP). In WRIGHT, there are no built-in constraints to ensure that design units are non-overlapping or that they are inside the configuration area. All requirements must be explicitly specified. Thus WRIGHT can solve problems containing design units at different *levels of aggregation* and generate tightly and loosely packed configurations by changing the problem requirements.

Figure 11 shows the disjunctive constraints defining Problem 1. Constraints *PC-19*—*PC-24* express adjacency requirements and are termed *performance constraints*. *PC-9*—*PC-18* specify that all interior spaces must be inside the apartment and interior spaces should not overlap; they are termed *realizability constraints*. Constraints *OR-1*—*OR-16* eliminate trivial holes by enforcing that every design unit is adjacent to either another design unit or to the boundary of the envelope on all sides. These latter are called *style* constraints in WRIGHT. Style constraints may also specify which design units can be adjacent to the boundaries of the envelope or occupy corners.

Variables, atomic constraints and disjunctive constraints form a *constraint graph*, which is an and/or network created by the *constraint compiler* at the outset of search. The inputs to the constraint compiler are a taxonomy of

---

PC-23 bathroom next-to vestibule ≥ 70
$(((vbN + [70,\infty] = btS) \wedge (btN + [70,\infty] = vbS) \wedge (btW = vbE)) \vee$
$((vbN + [70,\infty] = btS) \wedge (btN + [70,\infty] = vbS) \wedge (btE = vbW)) \vee$
$((vbW + [70,\infty] = btE) \wedge (btW + [70,\infty] = vbE) \wedge (btS = vbN)) \vee$
$((vbW + [70,\infty] = btE) \wedge (btW + [70,\infty] = vbE) \wedge (btN = vbS)))$

PC-15 vestibule non-overlap bathroom
$((vbN \geq btS) \vee (btN \geq vbS) \vee ((vbW \geq btE) \wedge (vbS > btN) \wedge (btS > vbN)) \vee$
$((btW \geq vbE) \wedge (vbS > btN) \wedge (btS > vbN)))$

PC-12 bathroom inside apartment
$((btN \geq apN) \wedge (btW \geq apW) \wedge (apE \geq btE) \wedge (apS \geq btS))$

OR-1 bathroom north-adj (livingroom $\vee$ *vestibule* $\vee$ *kitchen* $\vee$ *N)*
$(((btN = lrS) \wedge (btE > lrW) \wedge (lrE > btW)) \vee$
$((btN = vbS) \wedge (btE > vbW) \wedge (vbE > btW)) \vee$
$((btN = ktS) \wedge (btE > ktW) \wedge (ktE > btW)) \vee (btN = apN))$

Figure 10. Expressing spatial relations between design units as disjunctive constraints

prototype design units (also used in LOOS); the templates defining the spatial relations used in constraints; general knowledge about the design domain in the form of desired spatial relations between the prototype design units; and the design unit instances and variables in a problem. For example, given the domain constraint that rooms should not overlap and the design units in Problem 1, the constraint compiler creates the constraints *PC-13—PC-18*, and by using the templates defining the spatial relation non-overlap, it creates their atomic constraints. The design unit taxonomy, the templates defining spatial relations and the domain constraints are represented explicitly and declaratively, are extensible and can be modified by the user through a graphical interface. Users thus can have direct control over the behavior of WRIGHT and apply it to solve layout problems in different domains.

4.4. SOLUTION METHOD AND SEARCH CONTROL

WRIGHT solves the DCSP created by the constraint compiler by sequentially instantiating disjunctive constraints using backtracking search. A disjunctive constraint is instantiated by selecting one of its disjuncts. The atomic constraints in the disjuncts selected in a search path define the configuration. During search, *forward-checking* (Haralick & Elliott, 1980) removes disjuncts that are incompatible with already instantiated disjunctive constraints from further consideration and identifies disjuncts that are satisfied due to transitivity or constraint propagation. The *singleton-disjunct* heuristic instantiates any disjunctive constraint that has only one disjunct left in its domain by immediately asserting the disjunct. The *DCSP* is solved when all disjunctive constraints are instantiated.

Figure 12 shows the search tree that WRIGHT generates as it solves Problem 1. The numbers attached to states indicate the order of generation. The

PC-9	*vestibule inside apartment*
PC-10	*livingroom inside apartment*
PC-11	*kitchen inside apartment*
PC-12	*bathroom inside apartment*
PC-13	*vestibule non-overlap livingroom*
PC-14	*vestibule non-overlap kitchen*
PC-15	*vestibule non-overlap bathroom*
PC-16	*livingroom non-overlap kitchen*
PC-17	*livingroom non-overlap bathroom*
PC-18	*kitchen non-overlap bathroom*
PC-19	*livingroom completely-next-to S*
PC-20	*livingroom completely-next-to W*
PC-21	*livingroom next-to vestibule* ≥ 90
PC-22	*kitchen next-to livingroom* ≥ 120
PC-23	*bathroom next-to vestibule* ≥ 70
PC-24	*vestibule next-to E*

OR-1	*bathroom north-adj (livingroom* $\vee$ *vestibule* $\vee$ *kitchen* $\vee$ *N)*
OR-2	*bathroom south-adj (livingroom* $\vee$ *vestibule* $\vee$ *kitchen* $\vee$ *S)*
OR-3	*bathroom east-adj (livingroom* $\vee$ *vestibule* $\vee$ *kitchen* $\vee$ *E)*
OR-4	*bathroom west-adj (livingroom* $\vee$ *vestibule* $\vee$ *kitchen* $\vee$ *W)*
OR-5	*kitchen north-adj (vestibule* $\vee$ *livingroom* $\vee$ *bathroom* $\vee$ *N)*
OR-6	*kitchen south-adj (vestibule* $\vee$ *livingroom* $\vee$ *bathroom* $\vee$ *S)*
OR-7	*kitchen east-adj (vestibule* $\vee$ *livingroom* $\vee$ *bathroom* $\vee$ *E)*
OR-8	*kitchen west-adj (vestibule* $\vee$ *livingroom* $\vee$ *bathroom* $\vee$ *W)*
OR-9	*livingroom north-adj (vestibule* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *N)*
OR-10	*livingroom south-adj (vestibule* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *S)*
OR-11	*livingroom east-adj (vestibule* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *E)*
OR-12	*livingroom west-adj (vestibule* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *W)*
OR-13	*vestibule north-adj (livingroom* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *N)*
OR-14	*vestibule south-adj (livingroom* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *S)*
OR-15	*vestibule east-adj (livingroom* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *E)*
OR-16	*vestibule west-adj (livingroom* $\vee$ *kitchen* $\vee$ *bathroom* $\vee$ *W)*

Figure 11. Disjunctive constraints defining Problem 1

constraint identified beneath intermediate states is the disjunctive constraint that is instantiated in that state. States 1,3,5 and 8 have two lists beside them. The top list shows the disjunctive constraints which forward-checking identifies as satisfied. The bottom list shows the disjunctive constraints that are instantiated by the singleton-disjunct heuristic or identified by forward checking after applying the singleton-disjunct heuristic. Solution states have their configuration(s) shown under them. WRIGHT generates 39 states and finds 22 solutions without generating any dead-ends.

The search space expanded by backtracking is the Cartesian product of the domains of all disjunctive constraints and increases exponentially with the number of disjunctive constraints. However, adding constraints reduces the number of solutions and possibly also the number of search states that must be examined because additional constraints increase the probability of inconsistent combinations. A more realistic measure of the effort required to solve a problem is problem "difficulty" according to (Purdom, 1983): *Hard* problems have an exponential number of solutions, and it takes exponential time to solve them by backtracking search; *difficult* problems have an exponentially small number of solutions, but backtracking still takes exponential time; *easy* problems have an exponentially small number of solutions, and there are known procedures for solving them in polynomial time. Problem difficulty is reduced as the ratio of

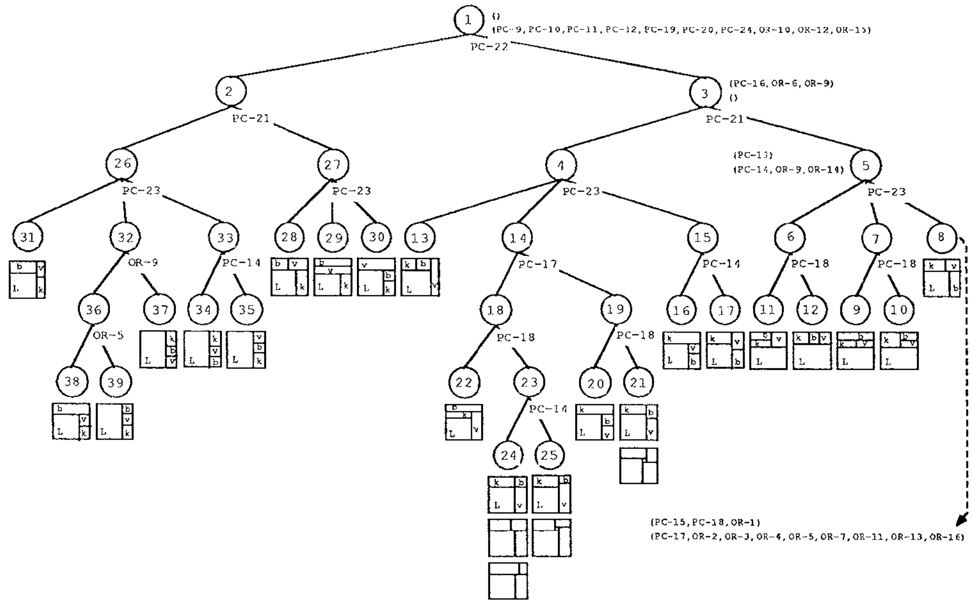


Figure 12. State space for Problem 1 as generated by WRIGHT

constraints to variables increases. Purdom showed that for a subset of difficult problems, backtracking takes polynomial time with dynamic instantiation and exponential time with fixed instantiation. He also conjectured that dynamic instantiation may save exponential time throughout the difficult region even though the resulting times may still be exponential. As they are initially given, layout problems usually do not contain enough constraints to restrict the solutions to an exponentially small set. Baykan (1991) conjectured that by modifying constraints, the designer changes hard problems into difficult ones. These are the problems on which the dynamic control strategy of WRIGHT leads to the greatest reduction in search effort.

WRIGHT selects the disjunctive constraint to instantiate dynamically in each state, using a function of *textures*, which are measures of topological and other features of the constraint graph (Fox, et al., 1989). We have defined three texture measures: *Looseness-1* implements a fail-first strategy by selecting a disjunctive constraint that has the fewest active disjuncts; thus the highest probability of failing and terminating the current branch of the search tree using minimum effort. *Looseness-2* calculates the reduction in the domains of interval variables due to satisfying a disjunctive constraint; it is a formulation of fail-first and prune-early strategies that takes into account the sizes of the design units, their current locations and the type of spatial relation in a uniform way. *Interaction* is a measure of the interaction between disjunctive constraints due to shared design units; it favors a constraint that interacts strongly with others, which reduces dead-ends and thrashing (Baykan, 1991).

Textures are applied lexicographically in WRIGHT. Each texture assigns ratings to all future disjunctive constraints and eliminates those with lower values. Dynamic selection using textures reduced search by between 80–90% in difficult problems and by 34–67% in easy problems over random instantiation orders (Baykan, 1991). WRIGHT demonstrates that analysis of problem structure in a domain independent fashion can lead to very good problem solving performance.

5. Comparison

Both LOOS and WRIGHT are portable across workstations running Unix, CommonLISP and X11. LOOS is written in *LISP* and CLOS (Common Lisp Object System). WRIGHT is written in LISP with a few of the critical procedures written in C. In our experiments, both systems were running on single-user DEC 5000/200 machines - LOOS on a machine with 48 megabytes of memory, WRIGHT on a machine with 64 megabytes of memory. But the CPU times given in the following tables should be seen as only broad indicators. Both systems were implemented in the context of research projects with the goal of demonstrating the underlying approach, and computational efficiency received little initial attention beyond attempts to follow established rules of "good programming". WRIGHT has since then undergone determined efforts to improve its computational performance, which caused improvements by several orders of magnitude. LOOS on the other hand keeps expanding in its capabilities without much attention being paid to this aspect; in particular, the abstraction and decomposition capabilities described below may add overhead even when they are not used in solving the problem (as is the case in all examples shown in this paper).

Table 3 gives some basic statistics on the computational efficiency of both LOOS and WRIGHT in solving Problem 1. For LOOS, the results are given for running the system with pretesting enabled and disabled. WRIGHT was run with or without style constraints.

Table 3. Computational performance of LOOS and WRIGHT in solving Problem 1

		No. of states	No. of solutions	CPU time (sec)
LOOS	pretesting enabled	40	24	5.1
	pretesting disabled	51	24	5.9
WRIGHT	with style constraints	39	22	0.2
	without style constraints	41	23	0.1

Given that both systems are based on formalizations that guarantee properties such as completeness of search, it is not surprising that they generate very similar solution sets when solving the same problem. This becomes especially obvious when one realizes that the representation underlying LOOS makes coarser distinctions in terms of spatial relations than WRIGHT; for example, the LOOS solution 30 represents the WRIGHT solutions 21 and 25 (without non-trivial holes) because LOOS suppresses the distinctions made by WRIGHT between these two possibilities; LOOS generates the variants with non-trivial holes, however, as distinct solutions.

The only real difference between the two solution sets is that LOOS generates solution 28, which has no equivalent in the set produced by WRIGHT, and this points to a deeper difference between the two systems. When WRIGHT is run with the no-trivial-hole constraint in Problem 1, it pushes in this particular case every space towards the external boundary and produces layouts that are as densely packed as possible; it also activates for the hall the maximum dimension constraint, which cannot be satisfied for solution 28 under these circumstances. LOOS cannot take maximum dimensions (or maximum areas) into consideration when it evaluates intermediate states because these constraints have consequences only for densely packed arrangements, which LOOS can only generate through its postprocessor, for example, an optimizer that attempts to eliminate trivial holes for a configuration of spatial relations generated by its generator. WRIGHT, on the other hand, can deal with the entire set of units at any level in the search. This difference points to a real distinction between the constructive approach taken by LOOS and the reductionist approach underlying WRIGHT.

However, upper bounds like the ones under consideration here often reflect more general concerns of efficient space planning. LOOS can take these into account during generation; for example, it can estimate the minimum area needed to accommodate placed units for any state and incorporate this as a true criterion into its branch-and-bound strategy because the minimum area can only increase through placement of additional objects; this will be demonstrated in the next example.

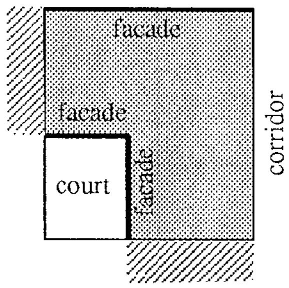
As a second problem, we chose a modified version of the problem described in (Flemming, 1978), which calls for the layout of a 3-bedroom apartment in an L-shaped area that borders an open court from the north and east and is accessed from a corridor on its eastern side; the western and southern sides of the available area are blocked from receiving natural light. Table 4 gives a general formulation of the problem that would assure minimum standards of comfort. The court is treated as a unit with variable dimensions and adjacency requirements that assure its position in the south-west corner of the overall area. The general requirement for natural light and ventilation for the major spaces is expressed through adjacencies with the northern border or the court. The adjacencies that force the living room to be adjacent to the entrance side and the master bedroom to the NW corner do not reflect standards of comfort, but general design heuristics that place these larger spaces immediately in the most appropriate zones within the given area.

This general problem formulation is severely underconstrained, and both LOOS and WRIGHT produce an unmanageably large number of feasible solutions. At the time of this writing, we are still experimenting with different ways of handling this problem for both systems. The question is how to introduce general guidelines of good or efficient layout design into the systems.

An obvious way for LOOS is to estimate the minimum overall area for any state as indicated above and take these estimates into account during branch-and-bound. In the current implementation, these estimates come into play after all constraints on the currently placed units have been considered; that is, the controller expands those states that violate not more constraints than any other state and have the lowest area estimates. This left the controller itself completely unchanged. The results produced by this approach are encouraging: LOOS generates 26 feasible "best" solutions in reasonable time (see Table 5), where the solutions are allowed to exceed the minimum area by a preset factor (this allows the generation of solutions that exceed the optimal area by a small

Table 4. Layout problem 2

Context



Spaces

Court	Min. dimension	3.60 m		
Living room	Min. dimension	3.60 m	Min. area	22.00 m^2
Master bedroom	Min. dimension	3.30 m	Max. dimension	5.40 m
	Min. area	14.00 m^2		
Bedroom 1	Min. dimension	2.40 m	Max. dimension	4.20 m
	Min. area	7.20 m^2	Max. area	10.00 m^2
Bedroom 2	Min. dimension	2.40 m	Max. dimension	4.20 m
	Min. area	7.20 m^2	Max. area	10.00 m^2
Hall	Min. dimension	1.20 m	Max. dimension	6.00 m
Kitchen	Min. dimension	2.10 m	Max. dimension	5.40 m
	Min. area	7.20 m^2		
Bathroom	Min. dimension	1.80 m	Max. dimension	4.20 m
	Min. area	4.20 m^2		
Max. extent of overall area from north to south:			18.00 m	
Max. extent of overall area from west to east:			12.00 m	

Required adjacencies (min. length of shared boundary in brackets)

Court/southern border (3.60 m)	Court/western border (3.60 m)
Living room/eastern border (3.60m)	Living room/northern border or court (3.60 m)
Living room/hall (.90 m)	Living room/kitchen (.90 m)
Bedroom/hall (.90 m)	Bedroom/northern border or court (1.20 m)
Master bedr./northern border (3.30 m)	Master bedr./western border (3.30 m)
Kitchen/northern border or court (.90 m)	

fraction, but may have other advantages).

One approach that can be used in WRIGHT is to declare style constraints that eliminate trivial holes and prevent the hall from being placed towards the outside. In the current problem, this reduces the number of feasible solutions from 3134 to 97 as seen in Table 5 and markedly improves solution quality. But WRIGHT could also adopt the optimizing approach taken by LOOS, which should lead to a reduction in search over the current satisficing formulation, but we have not made this extension. We experimented with the satisficing approach by placing a limit on the maximum area of the apartment. If a reasonable bound is not known at the outset, the system can be run repeatedly with increasing

Table 5. Computational performance of LOOS and WRIGHT in solving Problem 2

		No. of states	No. of solutions	CPU time (sec)
LOOS with area minimization	pretesting enabled	517	26	130.7
	pretesting disabled	831	26	188.8
WRIGHT	underconstrained problem	6930	3134	30.5
	with style constraints	630	97	14.1
	area $\leq$ 86 m^2	321	7	12.3

values until a good limit has been found. In the present example, an initial limit of $85m^2$ generates the solution seen in the top left corner in Figure 13, and a second run with the limit raised to $86m^2$ generates the 7 feasible solutions in the same figure.

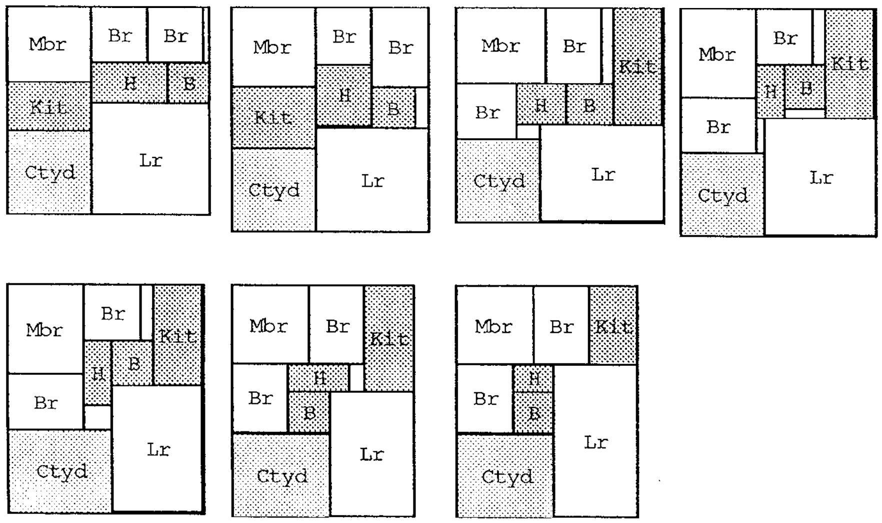


Figure 13. Minimum area solutions generated by WRIGHT for Problem 2

WRIGHT is again significantly faster that LOOS. But it is interesting to note that LOOS's performance improves relatively to WRIGHT, and this indicates a promising direction for further study. At the present time, we can only speculate that the explosion in the number of constraints that have to be explicitly considered by WRIGHT becomes more significant for larger problems. For example, the number of constraints that assure non-overlap in WRIGHT increases with the square of *n*, the number of design units. In LOOS, this constraint is automatically satisfied by the generator, and the number of tests to be performed increases roughly linearly with *n*. Thus, LOOS and WRIGHT may become more similar in computational efficiency as *n* increases, and this would certainly be a counterintuitive result.

WRIGHT is able to compute more accurate minimum values for the overall area due to achieving path-consistency. LOOS on the other hand executes its tests independent of each other and does take into account only a limited set of interactions (like the crossing of adjacencies along a channel mentioned above). As a result, two units may lay claim to the same area without realizing that they cannot occupy it simultaneously, and the first and very rough area estimates implemented for LOOS tend to underestimate the minimum area significantly; this explains why LOOS finds more solutions than WRIGHT. But this is a limitation not so much of the approach itself, but of the current implementation of the tester. Even so, the solutions sets produced by the two systems favor the same overall allocation of objects. This leads us to believe that the current very rough area estimates made by LOOS can be improved without more dramatic changes to the tester and indicates a fruitful direction for further study.

Some important differences between the two systems are not revealed by the two problems described so far. One is that there exist constraints in layout synthesis whose formulation would be difficult in WRIGHT. An example is the requirement that two units be physically accessible from each other, that is, that there be a path between the two units with at least minimal clearance at each point. The difficulty is that an arbitrary number of additional units may be involved in maintaining the path, and this number changes with the layouts themselves. LOOS can handle this constraint without difficulty through a test rule that tries to find this path in a configuration to be evaluated (this is easy because the channel representation indicates all possible paths; LOOS can even find the path with minimal distance along these channels). WRIGHT would have to come up with a disjunct that specifies *all possible paths* between the two units in terms of the various sets of additional units involved. A more promising solution may be to add a test after the DCSP has been solved; that is, WRIGHT may have to include some form of postgeneration test similar to LOOS.

WRIGHT deals with an overconstrained problem by relaxing constraints. It helps greatly if the user has specified which constraints can be relaxed, and in which order to relax the constraints. If these are not specified, the default is to relax style constraints first and performance constraints later. As relaxation possibilities increase, performance gets worse. The generate-and-test approach used by LOOS, on the other hand, does not need such devices because its evaluations can be carried out on feasible or infeasible solutions. A problem occurs, however, when pretesting becomes too strict and prevents the generation of any solution. This is another fruitful area for further study.

In general, the distinction between the generator and tester underlying LOOS enables the system to treat a broad set of constraints and criteria uniformly, while WRIGHT may have to make special provisions for certain classes of constraints. A limitation of the constructivist approach underlying LOOS is that certain aspects have to wait for evaluation until the relevant units have been placed; and the current implementation of the tester does not handle interactions between constraints in a consistent manner. WRIGHT, on the other hand, considers interactions automatically and does not have to delay the satisfaction of constraints for those constraint classes that fit well into its approach.

Another difference between the constructive approach of LOOS and the reductionist approach of WRIGHT stems from differences in degrees of interaction they allow. Every state generated by LOOS represents a formally complete layout that can be understood as such by an observer, who can thus follow the generation process step by step and interrupt at any state. Such an interactive editing capability is currently being developed for LOOS. Furthermore, transitions between modes of generation are easy at any state: a designer may take over and complete the layout through interactive editing, invoke an optimizer to test the potential of an intermediate solution or shift to an iterative improvement strategy. All of this can be done based on the same representation and may use the same rules. For example, any backwards application of a generation rule removes an object and produces again a formally complete layout which can be displayed as such. Removed objects can be reinserted by reapplication of the generation rules, and the tester is a general purpose tool that can evaluate any configuration independently of the way in which it was generated.

In WRIGHT, all design units are placed inside the design envelope at the outset, as it were, and their bounds overlap until their relative locations are determined by the incremental satisfaction of constraints. Thus, displaying an intermediate state poses some problems. On the other hand, an explicit representation of constraints and bounding boxes opens the possibility of constraint propagation in real time, which would change the whole configuration in response to user's actions. Graphical interaction with WRIGHT is one of the research topics being pursued.

Both LOOS and WRIGHT will eventually run into problems as the number of design units increases. Much of the recent work on LOOS has been devoted to this issue and resulted in an expanded version, ABLOOS (Abstraction-based LOOS) (Coyne, 1991, Coyne and Flemming, 1990), which indicates a direction for dealing with this problem that can, in principle, also be used by WRIGHT.

ABLOOS was conceived both as a hierarchical extension of the LOOS approach and as an extensible design framework that is evolving to incorporate a variety of design strategies and methods for producing alternative layouts. It provides designers with an interactive planning capability that allows a layout task to be hierarchically decomposed into subtasks. Each subtask represents a layout problem at a specific level of abstraction (scale or granularity); the subtasks can then be solved and recomposed to achieve an overall solution.

To achieve this, ABLOOS extends the representation underlying LOOS recursively so that a configuration can be decomposed into rectangular components and subcomponents; that is, a rectangle may represent a single design unit or a configuration of rectangles which may in turn represent configurations. This makes it possible to treat a layout as a hierarchy of components and to model decompositions typically found in artifact design; for example, a building is subdivided into floors, a floor into departments, a department into rooms and a room into clusters of furniture or equipment. Such a decomposition defines at the same time a division of the layout problem into tasks and subtasks; that is, it can be used to partition both the *process* and *product* of design. The construct used to decompose uniformly both the layout process and the components to be placed is called a *goal-object* (GOB) in ABLOOS. Each GOB specifies a complete layout task in terms of subgoals represented in turn as GOBs. A problem is decomposed by a designer by specifying a hierarchy of GOBs.

6. Conclusions

Our conclusions are still tentative at the time of this writing because our experiments with the two systems under review have not been completed yet. But we hope that even the short comparisons provided so far indicate how useful this type of close inspection in the context of realistic design problems can be.

Another note of caution has to be added with respect to the generalizations suggested by our findings. Positive aspects surely indicate circumstances in which the overall approach taken by a system (i.e. hierarchical generate-and-test vs. constraint-directed search) is working. But negative aspects have to be interpreted with greater care: they may not indicate shortcomings of the overall approach, but of either the design strategy implemented under the approach (constructive vs. reductionist) or, at an even lower level of abstraction, the particular form in which the design strategy has been implemented (representation, operators, control). We try in the following to keep these levels of distinction in mind when summarizing our findings.

At first sight, our data confirm what is generally known about the two approaches; CDS is computationally more efficient, but less general than HGT in the type of constraints and criteria it can incorporate. More interesting are indications that each approach can overcome some of its limitations by incorporating features of the other approach. CDS can add in principle postgeneration tests to account for constraints that are hard to formulate in the generality required at the outset. HGT can incorporate mechanisms that preclude many infeasible states from being generated.

If pushed hard enough, the two systems (and the approaches they represent) may become roughly equivalent in terms of efficiency when solving equivalent problems. That is, further work may blur the basic differences between the systems. But the actual development of the two systems has been moving in opposite directions. The developers of WRIGHT have been concentrating on identifying heuristic measures of features of the constraint graph that enable efficient search strategies, while the developers of LOOS have been working towards breaking its monolithic problem-solving strategy apart and making the mechanisms used individually available for implementing a broader range of design strategies and modes from a common toolkit. ABLOOS is a first version of a general framework that would allow for this.

These diverging directions may indicate deeper differences between the two approaches than are brought out by a comparison of run-time statistics, however instructive they may be: WRIGHT appears as a precision tool that executes the tasks for which it was designed very well, while LOOS appears as a collection of powerful mechanisms that can be combined to implement various contrasting design strategies, including and especially interactive ones, and allow for easy transitions between strategies.

Acknowledgments

Work on LOOS and ABLOOS has been supported by EDRC, the Engineering Design Research Center at Carnegie Mellon, an NSF-supported Engineering Research Center.

References

Baykan C.A. : 1991, *Formulating Spatial Layout as a Disjunctive Constraint Satisfaction Problem*. Doctoral dissertation, Dept. of Architecture, Carnegie Mellon University, Pittsburgh, PA.

Baykan C.A. and Fox M.S. : 1991, Constraint satisfaction techniques for spatial planning. In P.J.W.ten Hagen, P.J.Veerkamp (Ed.), *Intelligent CAD Systems III Practical Experience and Evaluation*. Berlin: Springer-Verlag.

Baykan C.A. and Fox M.S. : forthcoming, WRIGHT: A constraint-based spatial layout system. In C. Tong and D. Sriram (Eds.), *Artificial Intelligence in Engineering Design*. Academic Press.

Buchanan, B.; Sutherland, Georgia and Feigenbaum, E.A. : 1969, HEURISTIC DENDRAL: a program for generating explanatory hypotheses in organic chemistry. In Meltzer, B. and Michie, D. (Ed.), *Machine Intelligence 4*. Edinburgh: Edinburgh University Press.

Coyne, R.F. : 1991, *ABLOOS: An evolving hierarchical design framework*. Doctoral dissertation, Dept. of Architecture, Carnegie Mellon University, Pittsburgh, PA.

Coyne, R.F. and Flemming, U. : 1990, Planning in Design Synthesis - Abstraction-Based LOOS. In J. Gero (Ed.), *Artificial Intelligence in Engineering V. Vol 1: Design (Proceedings of the Fifth International Conference, Boston, MA)*. New York: Springer (Computational Mechanics Publications).

Eastman, Charles M. : 1973, Automated space planning. *Artificial Intelligence*, *4*, 41-64.

Flemming, U. : 1978, Wall representations of rectangular dissections and their use in automated space allocation. *Environment and Planning B*, *5*, 215-232.

Flemming, U. : 1979, Representing an infinite set of solutions through a finite set of principal options. A. Seidel and S. Danford (Eds.), *Proc. 10th Conf. of the Environmental Design Research Association*. Buffalo, NY.

Flemming, U.; Coyne, R.; Glavin, T. and Rychener, M. : 1988, A generative expert system for the design of building layouts - version 2. In J. Gero (Ed.), *Artificial Intelligence in Engineering: Design (Proceedings of the Third International Conference, Palo Alto, CA)*. New York: Elsevier (Computational Mechanics Publications).

Flemming, U. : 1989, More on the representation and generation of loosely packed arrangements of rectangles . *Environment and Planning B. Planning and Design*, *16*, 327-359.

Flemming, U., Coyne, R. F., Glavin, T., Hung Hsi, Rychener, M. D. : 1989, A generative expert system for the design of building layouts (final report). Report EDRC 48-15-89, Engineering Design Research Center, Carnegie-Mellon University.

Fox M.S., Sadeh N., and Baykan C. : 1989, Constrained heuristic search. *Proceedings of IJCAI-11*. , IJCAI.

Haralick R.M. and Elliott G.L. : 1980, Increasing tree search efficiency for constraint satisfaction problems. *AI*, *14*, 263-313.

Mackworth A.K. : 1977, Consistency in networks of relations. *AI*, *8*, 99-118.

Purdom P.W. : 1983, Search rearrangement backtracking and polynomial average time. *AI*, *21*, 117-133.

Stefik, M. et al. : 1983, Basic concepts for building expert systems. In Hayes-Roth, F. et al. (Eds.), *Building Expert Systems*. Reading, MA: Addison-Wesley.

Supowit, K.J. and Slutz, E.A. : 1984, Placement algorithms for custom VLSI. *Computer Aided Design*, *16*, 46-52.