Design as Exploring Constraints

by

Mark Donald Gross

S.B. Art and Design
Massachusetts Institute of Technology
1978

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF DOCTOR OF PHILOSOPHY IN DESIGN THEORY AND METHODS AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY FEBRUARY, 1986

Signature of the author ______________

Mark Donald Gross
Department of Architecture
October 31, 1985

Certified by ______________

N. John Habraken
Professor of Architecture
Thesis Supervisor

Accepted by ______________

Stanford Anderson
Chairman
Departmental Committe for Graduate Students

Design as Exploring Constraints

by

Mark Donald Gross

Submitted to the Department of Architecture on October 31, 1985
in partial fulfilment of the requirements for the Degree of Doctor of Philosophy in
Design Theory and Methods.

ABSTRACT

A theory of designing is proposed, developed, and illustrated with examples from the domain of physical form. Designing is seen as the exploration of alternative sets of constraints and of the regions of alternative solutions they bound. Designers with different objectives reach different solutions within the same set of constraints, as do designers with the same objectives operating under different constraints. Constraints represent design rules, relations, conventions, and natural laws to be maintained. Some constraints and objectives are given at the outset of a design but many more are adopted along the way. Varying the constraints and the objectives is part of the design process. The theory accounts for various kinds of expertise in designing: knowledge of particular constraints in a design domain; inference--calculating the consequences of design decisions; preference--using objectives to guide decision-making; and partitioning--skill in dividing a large and complicated design into sets of simpler pieces, and understanding the dependencies between decisions. The ability to manage ambiguity and vagueness is an important aspect of design expertise.

A computational model supporting the theory is proposed and its implementation discussed briefly. The constraint explorer, a computational environment for designing based on constraint descriptions is described. We see how the constraint explorer might be used in connection with a simple space-planning problem. The problem is taken from the procedures of the Stichting Architecten Research (S.A.R.), a specific architectural design methodology developed to help architects systematically explore layout variability in alternative floorplan designs. Finally, a selected review of related work in constraint-based programming environments, architectural design methods, and the intersection of the two fields is presented.

Thesis Supervisor: N. John Habraken

Title: Professor of Architecture

Acknowledgements.

I am grateful to the following people.

My parents, Sonja Osaki Keller Gross and Eugene Paul Gross, for making me in the first place, for showing me the beauty of nature, and for encouraging me to follow my interests.

My dissertation committee, for their patience, intellectual community, continued confidence, and friendship; in particular,

Aaron Fleisher, for good arguments and hard questions;
N. John Habraken, for revealing a new way to understand built environments;
Seymour Papert, for articulating a vision of the computer as laboratory for learning.

Annette Dula, who taught me how to write, using this dissertation as a vehicle, and for continuing and patient criticism throughout the writing.

Jean Nilsson, whose insightful comments are always extremely useful.

Catherine Chimits and Fred Wu, who intrepidly implemented parts of the constraint explorer in constantly changing computing environments, and for continued interest in and criticism of the ideas presented here.

Gary Drescher, David Levitt, Margaret Minsky, and everyone at the Atari Cambridge Research Laboratory (1982-1984) for being a stimulating intellectual community. I am especially lucky to have been part of this unique group of friends that also includes many of the other people named on this page.

Danny Hillis and Ken Haase for suggesting some good references early on.

Steven Ervin, for helping to debug many of the ideas, for patiently sorting out confused arguments, and for following in my footsteps nevertheless; Sandy Isenstadt for improving an early draft of chapter two; Ming Wang for good discussions at an early stage.

Coral Software Corporation, for their dedication to excellence in personal computing and for technical support, and Linda Laplante Okun for knowledgeable administrative guidance through M.I.T.

Maurice K. Smith, whose unparalleled clarity in articulating principles of form first convinced me to undertake the present work.

Alfonso Govela and Michael Gerzso, for taking me seriously and directing my early efforts as an undergraduate.

Ranko Bon, Louis Bucciarelli, and William Porter, for asking sharp questions, and Donald Schon, for enthusiastic support despite skepticism; Robert Lawler and Patrick Purcell, for kind support when it was needed most.

Table of Contents

CHAPTER 1

Introduction

The art of design involves considerable expertise. We do not learn this expertise--how to design--explicitly, as a set of procedures to follow, as for example, we learn how to add, subtract, and multiply. Rather, we learn to design gradually, by observing more expert designers, studying designs that are known to be good, and through constant practice and criticism. We learn many techniques, rules-of-thumb, formulae, and tricks, but never a systematic method. Many design disciplines are now approaching a "complexity barrier" [Winograd 73] where traditional methods fail to produce acceptable solutions. More systematic design methods are needed to coordinate the efforts of teams of more than a very few designers, and also to tackle more complex problems. This thesis proposes a theoretical framework for understanding design processes, viewing design as exploring constraints and alternatives. The test of this theoretical framework shall be in the performance of a computer program that implements the operations of the theory. The dissertation also describes this computer program, called "the constraint explorer".

Some will object strenuously to the idea of a systematic design method, arguing that design is a creative endeavor and that efforts to make an explicit account of the design process remove some essential artfulness. The objection is often amplified when the efforts involve a computer, and to many readers, the word "constraint" connotes an unpleasant restriction of free will and creativity.* It is not my intent to argue here against this point of view. My purpose is only to advance explicit understanding of design processes. I believe that such an understanding is necessary in order to build a foundation for more comprehensive and powerful ways to design.

The theory presented here is not a normative theory of design; it does not distinguish good designs from bad ones. It is also not a psychological theory of design; it does not attempt to account for what goes on in designers' minds. Nor does the present theory prescribe any particular design expertise. That job remains for applications of the theory to particular design domains. For example, we might apply the theory to the design of housing, bridges, or integrated circuits. Each such design domain entails vast amounts of specific expertise. This theory is about how designers manage and manipulate this expertise. The theory rests on the assumption that designers work within rules, principles, conventions, and laws. We can describe all these as constraints on design attributes, or variables. Then we can see a design process as exploring alternative sets of constraints and exploring alternative solutions within each set of constraints. These two parts of the process are not seen as separate phases of design; rather they are integrated in time.

We begin to design by selecting a first constraint: "I need 40,000 square feet of floor space", or "bedrooms must be in a quiet part of the house". We add to the constraints and we change them. We explore alternatives by setting, or fixing the values of design variables. Each fix may affect other parts of the design. For example, the

* One might just as properly call the present theory "design as exploring objectives", or "design as exploring relations". We shall see that constraints, relations, and objectives are intimately related.

placement of one room may affect the position of another. Thus design expertise involves predicting the consequences of a fix on the rest of the design. Each decision is tentative at first and becomes more definite with time; hence we sometimes need to undo or retract previous fixes.

We work with many constraints. They come from many sources and are more and less flexible. The design of a building, for example, includes structural constraints, use-dimension constraints, temperature constraints, surface-material constraints, and others. A few constraints, such as gravity, are fundamental laws of nature and cannot be altered; others, chosen by the designer, can be changed; for example, stylistic conventions or the choice of a construction system. Some constraints are general architectural principles; others pertain only to the design at hand. We select and position building elements so as to meet all these constraints. We realize many constraints operationally, as rules-of-thumb for selecting and positioning material and space elements. The daylight in the hall will be good if there are windows on one side. The building will stand if beams span no more than twelve times their depth. For ordinary practice, these rules-of-thumb suffice; a simple position or dimension rule can often subsume a detailed understanding of structural or daylighting behavior.

Constraints form the boundaries of a multi-dimensional region where each dimension represents an independent design attribute; each point represents a variant, or alternative solution. For example, we can describe the size of a room using three variables: height, width, and length. To describe the room's color or its area we must introduce additional variables. These may relate to variables already in use, or they may be entirely independent. Area, for example, is the product of width and length; whereas color is an independent variable.

We explore the region of alternatives by trying different values for variables and comparing the results. Designers may have different exploration strategies. Trying extreme settings of values is one strategy. Fixing positions before dimensions is another. To select among alternatives we must have preferences, or objectives. We may prefer long rooms, square rooms, or rooms whose width-to-length ratio is the golden mean. We almost always, however, have more than one objective. For example, we may want both the largest and the least expensive alternative. Usually our objectives do not coincide; therefore we must compromise.

We can compromise among competing objectives by partitioning, or decomposing, the design into pieces and optimizing each piece for a different objective. In designing the foundation of a house, for example, we might optimize for strength, while in designing the wood frame we might optimize for light penetration. We must make some decisions together because they interrelate, while other decisions are easily separated. Based on constraint connectivity, this structuring of decisions can only be performed after the constraints have been stated. Seldom can we partition a design perfectly, but often we can partition a design in different ways.

Rules are essential to design; without them we have only free-expression. The concept of design rule is therefore central to the present enterprise. Architectural design rules specify the allowable building elements--both material elements such as columns, walls, and beams, as well as spaces such as gardens, halls, and rooms--and their proper positions relative to one another and to their built context, or site. How are rules adopted, adapted, invented, and explored in designing buildings and places? We can obviously use rules to check already executed designs. But rules also play an integral part in the process of defining and exploring designs. We sometimes abstract rules from a traditional building type, then use the rules to generate additional instances of that type. The ability to see patterns in, and abstract rules from a given set of designs is certainly important for the architect, but we shall not address it here.

Many traditional built environments have been studied and described in this way. Examples include work on traditional Japanese houses; San Francisco Victorian house form, siting, and facades; Spanish hilltowns; and Pompeii courtyard houses [Engel 64; Vernez-Moudon 85; Hille 82; Smith, Hille, and Mignucci 82; Habraken, Akbar, and Liu 82]. All these examples identify a set of elements and the rules to assemble them into coherent configurations that belong obviously to a certain building type. In his masters thesis, for example, Hille shows that we can understand San Francisco Victorian facades as discrete horizontal and vertical zones with minimum and maximum dimensions. He also shows that the combination of facade elements in these zones is strictly governed by rules. Hille presents these rules and shows how they can be used to generate new variations on the theme [Hille 82].

Design, however, is more than following rules; it is making rules as well. Design concerns inventing and adapting systems of form-organization as well as generating specific forms within a given rule-system. By making new rules and combining and modifying existing ones designers invent new styles and occasionally even new building types. Moreover, the rules are not all decided before the designing begins; rather they are adopted and invented throughout the design process. Rule-making may even continue into the process of building. For example, where the architect has simply said "there shall be bricks", the mason may impose a pattern.

Architects seldom express rules explicitly. Even when abstracting rules from a built reference, we draw until we come to understand the building's theme, or system of rules. Then we can invent variations on that theme. Usually the understanding remains implicit--we do not articulate the rules. One obstacle to explicitness is the lack of a way to express, or notate, architectural rules. As drawings (along with scale models) have been the traditional medium for communicating about the design of buildings since at least the rennaisance, one might reasonably suggest drawings as an appropriate medium for notating design rules. Drawings are useful for indicating specific design solutions, but not for indicating ranges of possible decisions. For example, in a single drawing one cannot easily express a range of alternative facade arrangements. Nor can one draw a room known only to have an area of one-hundred-fifty square feet. A drawing can illustrate a rule by showing a typical or extreme variant. Diagrams and sketches (drawings without dimensions), on the other hand, show position relations between elements and can more effectively convey the essence of a rule than can an exact drawing. The advantage of a diagram is its ambiguity, its ability to stand for a range of alternatives.

A typical design involves thousands of rules, or constraints, at many different levels. The theory requires that we express the rules explicitly using a formal notation that will be introduced in chapter 3. Without a computer program to manage the constraints, experiments would be too unwieldy. The computer becomes therefore a laboratory instrument that supports the investigation of the theory. The computer helps in several ways: it calculates the consequences of changes to the constraints and variables, it maintains a history of the design process and enables the designer to retract decisions in any sequence; it provides explanations for automatic inferences; it assists with the partitioning of large designs into smaller pieces; and it allows a set of constraints and variables to be partitioned in different ways, affording multiple views of the design.

The dissertation is organized as follows. Chapter 2 presents a general statement of the theory of design as exploring constraints. Architectural examples are used to illustrate the theory. Chapter 3 introduces the constraint explorer, a computer assistant based on the theory. We look first at an interactive session with the constraint explorer where we design a simple configuration of two columns and a lintel, then we look behind the scenes at the same session, observing the structures that the constraint explorer constructs to represent the simple design. Finally we look at two classes of constraints that are especially important in architectural design: position constraints and dimension constraints. In chapter 4 we examine other structures that make up the constraint explorer's model of a design: a hierarchy of elements and configurations, a hierarchy of general prototypes and specific instances, and the dependencies between elements, both inherent and induced. Chapter 5 shows how the constraint explorer would assist the designer in making basic-variants, a particular space-planning task in the S.A.R. design method. Chapter 6 explains how the constraint explorer might be implemented, showing the parts of the program and the data structures the constraint explorer uses. Finally, chapter 7 presents a review of other related work. I have placed the discussion of related work at the end of the dissertation in order to come more quickly to the main idea. Chapter 7 also sketches further work to be done in developing and testing the present theory: design as exploring constraints.

CHAPTER 2

Design as Exploring Constraints.

We now present and discuss the theory of design as exploring constraints*. In order to present the theory, however, we must first introduce its basic vocabulary, a set of terms used throughout the dissertation. The terms are explained more fully in the paragraphs following this introduction, but, for convenience, here is a brief summary. We describe a design as a set of constraints, or relations on a set of variables. Each variable has a value, that the designer may set, or fix. Some variables the designer may fix directly; others are calculated as consequences of those fixed. Every collection of constraints, variables, and values may constitute a partial design context, or package. The constraints in any context bound a region of alternative solutions, or variants. Each independent variable represesents one dimension of the region, or a degree of freedom. The degrees of freedom in a design fluctuate, increasing as the designer introduces new variables, and decreasing as variable values are fixed. The region's boundaries also fluctuate as the designer adds and changes constraints. Points in the region represent particular variants, or completely specified alternatives with definite values assigned to each variable. We explore, examine, and rank variants according to various objectives, or preferences.

We adopt this somewhat sparse vocabulary because we want to formulate the theory using a small number of precisely defined terms. Though the terms may sound technical, they refer to concepts already largely familar to designers. For example, adding constraints and

*One may notice similarities between the present theory and that presented by Herbert Simon in the Science of Design [Simon 69]. In Chapter 7 we compare and contrast Simon's theory with the present one.

fixing variable values correspond to what designers call 'moves' and 'decisions'. Likewise, designers know a region of variants as a set of options, possibilities, solutions, or alternatives. We shall refer often to these more familiar terms in the ensuing exposition of the theory.

Constraints are the rules, requirements, relations, conventions, and principles that define the context of designing. There are many constraints on a design and they come from different sources. Constraints are imposed by nature, culture, convention, and the marketplace. Some are imposed externally, while others are imposed by the designer. Some are site-specific, others not. Some are the result of higher-level design decisions; some are universal, a part of every design. Gravity, for example, is universal. Other constraints apply only in certain design contexts. The general position rules on windows and other facade elements that characterize facades in the Back Bay section of Boston are less universal constraints than gravity, but they are more universal than the additional constraints that operate in any particular Back Bay facade.

We can describe a design problem or task as a collection of constraints and relations on attributes of the object to be designed. That is, to design is to describe and identify constraints and to specify an object that satisfies all these constraints. For example, constraints on the design of a pencil are that it must leave an erasable mark on paper, that it be lightweight and comfortable to hold. But we could design many pencils that satisfy all these constraints: hard pencils, soft pencils, red pencils, thin pencils. Thus design problems are atypical problems in that they have many solutions. We do not find the solution to a set of design specifications; we find one solution out of many alternatives. Although we may prefer some alternatives to others, all are solutions to the initial constraints. At each step in a design we can distinguish among alternatives by adding constraints. The adding of constraints is as much a part of design as the searching for solutions. The design process consists of adopting constraints and then exploring for "good" alternatives in the region the constraints bound.

Examples of constraints are:

1. "Bearing walls occur on 5', 8', or 11' guide lines."
2. "X must be offset from Y by at least its own thickness."
3. "Kitchens must be at least 6' x 8' and occur only in beta zones."
4. "This window must admit at least 1 hour of direct sun."
5. "All material elements must be supported."

Constraints are always relations between variables that represent the attributes of the object being designed. In constraint 1 above, the variables represent positions of bearing walls and guide lines. In constraint 2, the variables are the positions of X and Y, and the thickness of X. In constraint 4, the amount of sun that enters the window is a variable (as, presumably, are some properties of the window itself: its position, dimension, orientation). These variables are simple; they stand for attributes of the design that can be represented by a single number. It is sometimes useful to aggregate variables to describe more complex attributes of a design. For example, a window can be a variable, its value to be chosen out of a set of possible windows. The window variable would then be an aggregate or compound variable consisting of the many simpler variables that describe attributes of windows: shape, size, material, transparency, method of opening, and manufacturer, for example. Constraint 5 expresses a universal constraint, gravity.

It is useful to distinguish between variables directly set by the designer and variables the designer controls only indirectly. In the design of our pencil, for example, we directly control the color, length, radius, and shape of the pencil but its mass is a function of its length, radius, and materials. Architectural designers control the selection, position, and dimensions of material and space elements. Other variables in the design are determined consequentially. For example, a variable representing the privacy of a room might be related to the number of turns needed to enter the room from the nearest major access, and whether one can see in from nearby locations. Though a designer may determine to provide a certain degree of privacy, it is ultimately by adjusting the positions and dimensions of building elements that the architect affects the privacy of a place.

The amount of daylight entering a room is another example. The designer controls daylight indirectly, by directly controlling the positions, dimensions, and orientations of openings in the room's exterior surfaces. (S/he may do it in other ways as well.) The amount of daylight admitted is related to the size, orientation, and position of each opening. This natural law about light and openings conveys a small piece of design knowledge. Architects know this relation though offhand they may not know its mathematical expression.

Consider the difference between a design specification and a set of construction drawings for a building. The former consists of performance constraints--on variables that the designer can control only indirectly, such as daylight and privacy. The latter consists only of constraints on variables the designer controls directly--the relative placement of material and space elements within certain tolerances. It takes an expert designer to transform a set of constraints on variables such as privacy, outlook, and daylight into a set of constraints on the positions and dimensions of material and space elements. For example, the constraint, "the room must be at least moderately light in the afternoon" might be alternately and equivalently expressed as constraints on the positions, dimensions, and orientations of windows -- "window sills must be at least three feet from the floor and the room must have have at least thirty square feet of window area on the west side".

Constraints describe a <u>region</u> in space, not in physical space but in an n-dimensional mathematical one. Let us call the space an "n-space" to distinguish it from the architectural meaning of the word "space". The number "n" stands for the number of dimensions, degrees of freedom, or independent qualities in the design; n may be large and it changes throughout the design process as variables are introduced and eliminated. Each dimension in the n-space represents one independent variable in the design. Each point in the n-space describes a complete set of variable values. A point in the region describes a complete set of variable values that meet all the present constraints. Hence each point in the region represents an alternative solution, or variant. Typically the region contains many such points. The region need not have a simple shape. It may be large in some dimensions and small in others. It may be all together in one place or in many small "islands". Both the dimensions of the n-space itself and the region within it change throughout the design process. In this change there are two overall tendencies. One tendency introduces new independent variables--new decisions to be made--throughout the design; the other tendency fixes values for variables that have been already introduced.

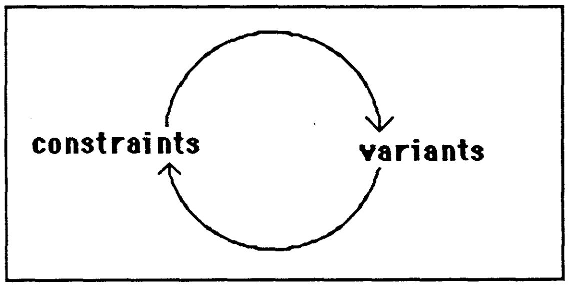
In most designs the initial constraints describe a large region of alternatives. That is, the solution to the design problem as first stated is grossly underconstrained; a great deal of freedom remains in the design. Novice designers experience a difficulty associated with this--with so much freedom it is hard to choose a course of action. Not only are there many degrees of freedom--many variables are unfixed, but also for each unfixed variable there is a large range of possible values. Suppose in our design of a pencil we have chosen values for all variables except length and color. We say then that two degrees of freedom remain in the design. Within each of those two degrees, however, there may be a large or a small range of options. For example, constraints may strictly limit the pencil's length while allowing a large range of colors. When all variables are fixed, the design is complete, having zero degrees of freedom.

The design is "complete", however, only relative to the constraints already adopted. New constraints may be added at any time. Instead of narrowing the range of possible alternatives, the new constraints may introduce new decisions. Consider designing a window in a wall. We may begin with constraints on the window's dimensions and position pertaining to the view, the amount of daylight, etc. At some point, not necessarily after fixing the position and dimension variables, we may begin to design the parts of the window itself. This entails new constraints and new variables. We may introduce constraints on the size of the panes and mullions, on the window's moving parts. These constraints introduce new variables to the design. They may be independent from the earlier decisions about the window's overall dimensions and positions but more likely they are not. For example, the window's overall dimension is related to the number, size, and arrangement of its panes and mullions.

At the outset there may be relatively few constraints and variables. Design proceeds from a general set of specifications to a set of specific solutions. At completion, many more constraints and variables have been introduced, and all variables have values. Thus each set of constraints and associated variable values represents only one instant of a design process. Each such instant we call a design <u>state</u> or <u>context</u>. In any state, some variables have particular values, others are unspecified. The more values defined in a state, the more specific the design. A region of alternatives exists only if the context is <u>consistent</u>; that is, no constraints or values conflict. The region at all times consists of the set of alternatives satisfying the present context of constraints. Often there are many alternatives and the region is very large.

Constraints, variables, and regions of alternatives, or variants make up the basic terms of the model. We now use these terms to discuss the process of designing. A design begins with a set of initial variables and constraints; it proceeds with the designer changing and adding constraints and fixing and unfixing variable values; it ends with a single complete variant, or sometimes a small, well-understood region of variants*. The path from specification to solution is usually not direct. Rather, many options are explored and rejected. Though the general tendency is to fix variable values, on occasion they are also unfixed, or retracted.

Design can also be understood as a process of successive refinement. Refinement proceeds in two alternating steps: describing constraints and exploring alternatives, or variants. The describing-step adds new constraints to the design; the exploring-step examines variants in the constrained region. These variants suggest changes and additions to the constraints. The cycle then repeats; the new context of constraints is explored, generating a new set of variants (figure 2.1 below). This process of refining constraints and exploring alternatives is repeated until it converges on a small region of acceptable variants, or alternatives.

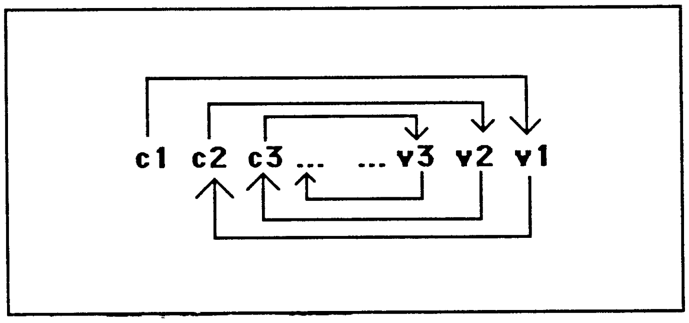


Refinement Cycle.
Figure 2.1.

Figure 2.2 illustrates the same process, but now we see that each iteration of the cycle produces a new set of constraints and a new set of variants. From the initial constraint context

* Designers sometimes leave a design intentionally unfinished leaving room for the user, the builder, or more generally, for the next level of designer to complete the design..

C1, a set of variants V1 is generated. Examination of the set V1 suggests a new set of constraints C2. The process iterates.



Each cycle produces new constraints and variants.
Figure 2.2

After describing a context of constraints and before making many firm decisions the designer must learn what variants the region contains. This the designer does by exploring. What are the extreme variants in the region described by the constraints? What are the degrees of freedom in the design? How large a bedroom can be made in this floorplan? Suppose this dimension were increased? Can these two rooms be moved independently? In general, what variants are in the region and what variants are not? Constraints from different sources may interact to describe a complex region of variants. The boundaries may be neither apparent nor intuitive. Some exploration may be required in order to understand the boundary of the region in detail. Therefore the designer may at first explore the region with only the goal of understanding what variants the region contains.

Design involves various modes, or aspects of expertise. Among others, design involves expertise in preference, inference, resolving conflicts, and on occasion, backtracking. Preference among alternatives is a vital piece of the designer's expertise and it pervades the design process. Design proceeds within constraints as a sequence of fixes, or decisions. At each decision we may work out several alternatives that follow from it. Then we rank the alternatives, choose the one we think best, and proceed to develop it. The choice of an alternative is not arbitrary; rather it is a skillful choice. Sometimes, however, we cannot decide immediately on an alternative so we develop two or three alternatives in parallel until the choice becomes more clear. Of course we cannot develop more than one alternative for long. It is too much extra work. Soon we must choose one and discard the rest. It is not that one alternative is right and the others are wrong; rather, we prefer one alternative to the rest. In exercising preference we apply a kind of expertise that is unique to design.

Design also involves <u>the ability to infer chains of logical consequences</u> of a decision. Each design decision has or may turn out to have logical consequences. For example, choosing a location for one room in a floorplan may determine the locations of other rooms. The location of a stairway may depend at least partially on its horizontal run-length, which depends in turn on its height and the ratio of its risers to treads. Following chains of consequences in reverse is equally important. When a design decision is retracted, the designer needs to know what other decisions depended on it and what else needs to be undone.

Determining a <u>sequence of fixes</u>, or <u>settings</u>, is another sort design expertise. The sequence of settings affects the design outcome. Each decision changes the region, and the alternatives available at any stage in a design process depend on the previous decisions. Within one set of constraints a different sequence of fixes often reveals a different result. For example, consider placing two rooms in a floorplan. If the livingroom is first placed in the sunniest spot, it may preclude some locations for the kitchen. But if the kitchen is placed first, then the sunniest remaining spot for the livingroom may be different. Some kinds of decisions are usually made before others. One usually (but not always) decides the relative positions of elements prior to fixing their dimensions precisely. One usually decides a beam span before selecting its depth and width. There is not, however, only one possible sequence of decisions in a design. Different architects begin in different places, some with construction details, others with a site plan. The different starting points, sequences of decisions, and preferences bring the design in each case to a different conclusion.

Constraints on the design are described by the designer in the beginning, but they are also added to and changed throughout the design process. Some constraints are working assumptions. For example, the designer may try to design a building for a certain cost and using certain materials. But that may turn out to be impossible, or perhaps possible but undesirable. In that case, the designer may change the initial assumptions and explore again. Both inconsistency among constraints and dissatisfaction with alternatives are occasions for changing the constraints.

Constraints on a design come from a variety of sources. Many of the constraints on a design are standard constraints, not specific to the design at hand, but shared across a variety of design contexts. The selection of a building technology, wood frame construction, for example, is a source of many constraints on a building's design, but these constraints are not specific to that particular building. Likewise, all buildings share constraints on daylighting, thermal performance, and structural stability. Many of these constraints are listed in the building code. Others are added to the design context not only by functional requirements of the building (as given in an architectural program) but also by systems of form organization imposed by the architect. For example, a designer may choose to work within the conventions of an architectural style, or adopt or adapt rules from neighboring buildings. Thus many of the constraints on the design are chosen by the designer and not imposed from outside.

As constraints are added to the design for many reasons and from many sources, the constraints on a design may be inconsistent, conflicting with one another. Resolving the conflicts, or inconsistencies, then becomes another important component of design expertise. A constraint concerning the support of a beam may locate a column in the center of the room, a condition perhaps forbidden by another constraint related to furniture arrangement possibilities. Or one constraint may require that the livingroom be located near the entry; a second may demand that the livingroom be on the sunny, or south side; a third insists that the entry be on the north side. The nature of design is to balance, resolve, and sometimes even exploit conflicting requirements.

For the same reasons that design constraints may be inconsistent they may also be redundant, or mutually reinforcing. Thus another aspect of design expertise involves managing redundant constraints. Constraints are redundant when they require the same thing. This may mean that the constraint is especially important. For example, both pedestrian access and visual continuity constraints might require an opening at a certain place in the edge of a public space. Or the size of a livingroom may be governed simultaneously by use constraints that limit the room size to 20' x 40', and by site constraints, allowing a maximum size of only 18' x 25'. In this case site constraints supercede use-dimension constraints. But we would not forget the use constraints, if later we change the site constraints, say by moving a wall. The site constraints might then no longer bound the livingroom size. The use constraints, however, still would.

What is the best design? That is a difficult question because there are always many competing objectives. The house should be sunny, but it should also be large, inexpensive, and easy to insulate. And there may be hundreds if not thousands of other objectives. We can optimize only one objective at a time. The sunniest house may not be the least expensive house. The largest house will not be the sunniest house. The question is difficult, not because we do not know our objectives, but because we cannot optimize the design for all of them simultaneously. The "best" house will not be the sunniest, nor the largest, nor the least expensive. There are trade-offs between these three qualities. The best house will be sufficiently sunny, sufficiently large, and sufficiently inexpensive. The best design will turn out to be a compromise or collage among our many objectives.

Design problems are complex, involving hundreds of constraints and decisions that require skillful management. Because we have many conflicting objectives we can not optimize over the entire design at once. Design expertise involves breaking or partitioning the design into workable-sized pieces, or fragments, working the pieces separately and then reassembling them. We try to minimize connections between the pieces so that we may work each piece independently. The pieces can be worked in parallel by different designers, or sequentially by one designer. We consider each piece as a separate design problem in which we may optimize a different objective. For example, we may optimize sunlight in the livingroom and size in the bathroom, if we choose to work the two rooms separately. We may optimize the foundation for strength, and the wood frame screens for light penetration. We may have to make adjustments where the pieces of the design do come together, but this can be kept to a minimum by cleverly partitioning the design. Later we reassemble the worked pieces of design, and the overall result shall not be a global optimization of one objective, but instead a piecing together of local optimizations.

Now let's compare the method of an expert and of a novice designer. The expert designer has explored extensively in previous sessions and no longer needs to try out many different alternatives. The expert is confident of immediately choosing a good one, based on experience. This partly explains the stylistic consistency of expert designers. Wright's houses, for example, are relatively similar to one other, and different from houses by another architect. Each of Wright's houses, particularly if we consider a short time period, represents only a slightly different variation on the same theme. Experts do not much vary the constraints. They tend to explore a limited and local region of alternatives, a region that they have come to know well through experience. Because expert designers proceed rapidly with minimum diversion towards their final alternative, their method might seem to be more problem-solving than exploration. But their method must be understood in terms of their previous experience in exploring. The inexperienced designer, on the other hand, must explore a great deal to learn which alternatives are more likely candidates. Upon gaining experience, the novice begins to build a set of preferences. In constrast, the expert designer has already built up a large set of default constraints and preferences that invariably result in satisfactory designs, and at least sometimes in excellent ones, thereby reducing the need to explore. However, when expert designers attempt to operate outside their familiar context of constraints, as in a foreign culture or in designing an unfamiliar building type, the expert becomes a novice once again.

CHAPTER 3

Use of The Constraint Explorer

We now illustrate the processes of describing and working with constraints. We begin with a simple scenario showing a dialogue between designer and constraint explorer in which the designer describes dimension and distribution constraints of a very simple configuration. In sections 3.1 and 3.2 we examine this process first from the designer's point of view, and then from the machine's. In sections 3.3 and 3.4 we look at examples of various dimension and position constraints. We see how to construct a vocabulary for describing the relative positions of material and space elements using combinations of simple arithmetic inequalities. In section 3.5 we take a closer look at a single constraint concerning the relative positioning of columns in a building.

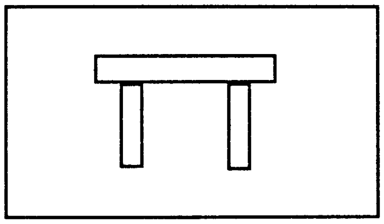
Although the theory that designing is exploring constraints and alternative solutions does not depend on a computer, the computer is the only practical way to explore such a theory. The computer makes the theory easier to test and apply. We need the computer because there are so many constraints to organize and so many alternatives to consider. The computer will be asked to do only mundane tasks: remember design alternatives, keep track of constraints, perform symbolic and numeric mathematical operations, and rank alternatives according to objectives.

The constraint explorer acts as an assistant, keeping track of the constraints as they are added, adopted, and removed from the design. It remembers the constraints on the design and where they came from. It can report which constraints are satisfied and which are not. It can also indicate what additional decisions are needed to specify the design, what degrees of freedom remain, the constraints on each degree of freedom, and issue warnings when it identifies inconsistent constraints or a decision that violates a constraint. The constraint explorer can calculate consequences of decisions, and it can exercise a preference you have specified for choosing an alternative from a constrained set of values. It remembers and can recall all the previous states in an exploration, and it can combine parts of one state with parts of another. It can remember what sequence of operations was performed on a previous occasion, and perform that sequence again upon request.

3.1 A Brief Scenario.

A brief scenario demonstrates some reasoning the explorer can do. A simple session with the constraint explorer is presented and discussed. Interaction is presented here as textual dialogue; that is, the designer issues commands and queries by typing at the keyboard (or by selecting items from menus), and the constraint explorer responds by typing back answers. Instead of interacting textually, designers may want to indicate constraints and settings by drawing and sketching. Here however, the topic is design reasoning, not user-interface. We shall concern ourselves with the design of a simple configuration of three elements--two columns and a lintel--and with three position relations between the elements: the two columns each support the lintel, and the columns are some minimum distance apart. The configuration we call for simplicity an "arch"* (figure 3.1). (This name is used in the sense of the basic structural principle of load-transferance rather than in the typical sense of a curved structure.)

* Technically "flat arch" or "portal" would be a more correct name.



"arch" configuration
Figure 3.1

To describe the "arch", we enter the following constraints:

```
>> Column-1 supports Lintel.
>> Column-2 supports Lintel.
>> 4' minimum-distance Column-1 Column-2.
> >
```

Figure 3.2

The ">>" in each of these lines is typed by the constraint explorer; it prompts the designer to enter a command or query. Throughout this scenario, text entered by the designer is preceded by the ">>", to distinguish it from the constraint explorer's responses. Having entered some constraints, we may wish to check that the constraint explorer remembers them. In figure 3.3 we ask to see the constraints:

```
>> What-constraints?
   A. (given) Column-1 supports Lintel
   B. (given) Column-2 supports Lintel
   C. (given) 4' minimum-distance Column-1 Column-2
   D. (deduced) Lintel length > distance-centers Column-1 Column-2.
>>
```

Figure 3.3

The constraint explorer lists the three constraints we entered and shows us the source of each. By printing "(given)" in front of each of constraints A, B, and C, it reminds us that we typed them in ourselves. Constraint D is a new constraint deduced by the explorer from the given constraints. The explorer has figured out that the lintel may be no shorter than the distance between columns! In figure 3.4 we ask the constraint explorer to account for that deduction. It responds with the basis for its deduction: the two "support" constraints, A and

B. Notice that the minimum distance constraint is NOT a basis for the deduction of the maximum-distance-between-columns constraint.

```
>> why D ?
   The deduced constraint:
   (Lintel Dimension) > (distance-centers Column-1 Column-2)
   was reached by reasoning from given constraints:
   A) Column-1 supports Lintel
   B) Column-2 supports Lintel.
>>
```

Figure 3.4

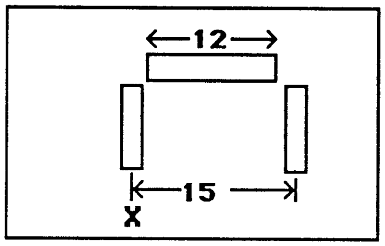
The constraint explorer stores all the design constraints and we can query it about the constraints on a particular variable. For example, we may ask "what are the constraints on the position of Column-2?". The explorer shows all constraints presently in the design that refer to the position of Column-2 (figure 3.5).

```
>> What-constraints on (Column-2 position) ?
   B) Column-2 supports Lintel,
   C) Column-2 is at least 4' from Column-1,
   D) Column-2 center is at most (Lintel length) from (Column-1 center).
>>
```

Figure 3.5

These are the same constraints as in figure 3.3 (except for A, that has nothing directly to do with Column-2) but here they are all expressed from the 'point-of-view' of Column-2. In this example it is apparent by inspection what constraints reference any particular variable. But in a design with hundreds of variables and constraints, one cannot tell by inspection all the constraints that control or might control a variable or variables. Hence this cross-referencing facilty is especially useful with larger designs, and perhaps even useful with small ones. The constraint explorer will also report constraint violations. If we position the columns fifteen feet apart, and specify a twelve foot lintel, the explorer reports a violation.

```
>> Set (Column-1 Position) X
   OK

>> Set (Column-2 Position) X + 15'
   OK

>> Set (Lintel length) 12'
```



```
     The move ...Set (Lintel length) 12'... conflicts with constraint:
    D. Maximum Lintel length
        (Lintel length) > distance (Column-1 center) (Column-2 center)
        (Lintel length) > 15,

    which was deduced from these three relations:
        ----------------------------------------
        LL. Lintel is rectangular.
                (Lintel length) = distance (Lintel left) (Lintel right)
         ----------------------------------------
        A. Column-1 supports Lintel,
            (Column-1 center) is between
               (Lintel left) and (Lintel right).
         ----------------------------------------
       B. Column-2 supports Lintel,
            (Column-2 center) is between
               (Lintel left) and (Lintel right).
        ----------------------------------------
>>
```

Figure 3.6

In figure 3.6 we set the columns fifteen feet apart and then called for a twelve foot lintel. The explorer noticed the inconsistency and reported it. At the end of the sequence both the conflicting constraint and the setting still remain in the design. We may 1) reposition one or both columns; 2) unset or change the lintel length; 3) remove one of the support constraints (this may cause other problems), or 4) let the inconsistency ride. In figure 3.7 we unposition Column-2.

```
>> UnSet Position Column-2.
   OK
>>
```

Figure 3.7

Next we perform a local optimization, setting the distance between columns.

```
>> Maximize distance between Column-1 Column-2.
    OK.  Distance between Column-1 and Column-2 is now 12'.

>> What is the position of column-1 and column-2?
    A. (given) The center of column-1 is at position X, and
    B. (deduced) the center of column-2 is at position X + 12'.

>> Why B?
    B1. maximize distance column-2 center and column-1 center (12).
    B2. center of column-1 is at X.

>> What settings?
    Lintel length is 12',
    Column-1 center is at X.
```

Figure 3.8.

In figure 3.8, only the position of one column and the length of the lintel remain set. Then we asked the explorer to choose the largest possible distance between columns, and we asked it for the positions of the columns. It reminds us that one of the positions was given, and the other is an extreme value, the result of maximizing the distance between columns. Finally, we ask for a list of the settings that hold in this context.

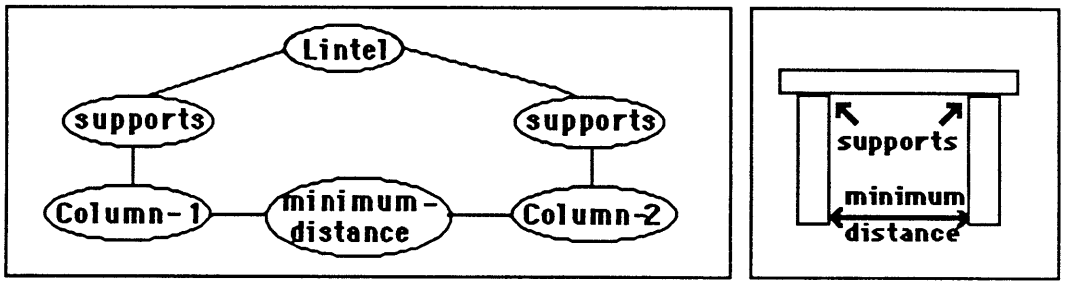
This concludes the brief scenario. Let us summarize the preceding sequences of interaction with the constraint explorer. First we described three constraints on the relative positions of three elements (figure 3.2). From these, and using built-in mathematical expertise, the constraint explorer deduced a fourth constraint, a maximum distance between the column centers (figure 3.3). We asked for an account of the reasoning that led to the deduced constraint (figure 3.4), and we asked for a list of constraints on a particular variable (figure 3.5). Then we placed two elements so as to violate the deduced constraint (figure 3.6); the explorer reported a violation and by showing the bases for the constraint that was violated, suggested alternate ways to resolve the conflict. We choose to resolve the conflict (figure 3.7) by retracting one of the placements and ask the constraint explorer to re-position the element by maximizing the distance between columns (figure 3.8).

We saw that the explorer can reason about given combinations of constraints and deduce new constraints as consequences. We saw also that the explorer can calculate the effect of one setting on other, connected parts of the design, and that it can recognize an inconsistent state and trace its possible sources. We have seen that the constraint explorer remembers the bases for its deductions, and can display the chain of events that has led to any particular state. This can help the designer to understand the steps that have brought the design to any state.

3.2 Behind the Scenes in the Constraint Explorer.

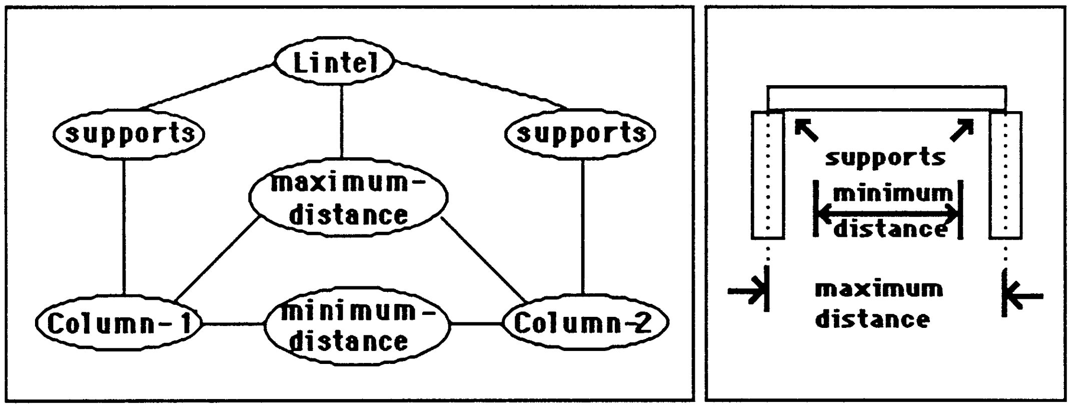
In the previous section we discussed constraints on the relative positions of parts of a simple configuration of two columns and a lintel. We looked at an interactive session with the constraint explorer. Our emphasis was on the role of the user. This section discusses the same interaction from the constraint explorer's point of view rather than the user's. We will examine the representation, or model, of the design that the machine works with. This description consists of the constraints and the values of the variables in the design--both changing throughout the design; some are set by the designer and others are computed by the machine. We can diagram the machine's description as a network of constraints and variables. The figures that follow diagram the design model as it is constructed, maintained, and modified by the designer in connection with the constraint explorer.

In initially describing the design of an "arch", we wrote three position constraints on three elements. Figure 3.9a diagrams the description of the initial design state as the program constructs it in its database. Figure3.9b illustrates the form of the "arch", with annotated constraints. Figure 3.9b represents one variant of many that meet the constraints. The constraint explorer can construct such an illustration using default values for variables in the constraint description.



Initial description of "arch".
Figure 3.9a, b.

In the previous section we saw that after the three initial constraints were given, the machine inferred a new constraint, a maximum distance constraint between the two columns (Section 3.1, Figure 3.3). (We discuss in Chapter 6 how the program makes such inferences). Figure 3.10a below shows the description after the inferred constraint is added. Figure 3.10b shows the form of the configuration with the new constraint annotated.



"Arch" description with deduced constraint
Figure 3.10a, b

Now let's look at the description one level deeper in detail than in Figures 3.9 and 3.10. We see that both elements and relations in Figure 3.10 are packages of more primitive constraints and variables. That is, each of the elements "Column-1", "Column-2", and "Lintel", and each of the relations "supports", and "minimum-distance" in Figure 3.9 stands for a package of constraints and/or attribute variables, the definitions of which have been previously put in by a designer. Packages are indicated in the diagrams as ellipses. Each package has a name and may contain constraints and variables.

We now look more closely at the package of variables that describes the lintel. To keep matters uncomplicated we shall treat the lintel here as a simple element, that is, it has no smaller parts. (In chapter 4 we shall discuss configurations of elements). Figure 3.11a shows the relevant variables in the lintel package. These describe the lintel's qualities, including the positions of its edges: left, right, top, and bottom, and its dimensions: length and height. Each edge may also be described by a package of variables, but here we shall not be looking in that close detail. Therefore in the diagram we need not draw the ellipses around names of edges.

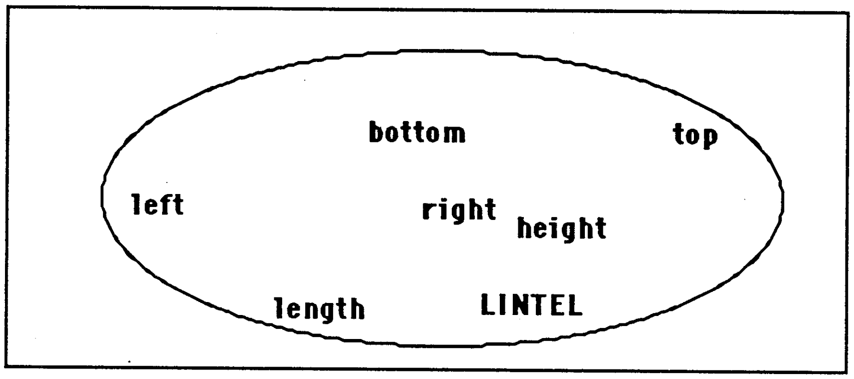


Figure 3.11a: Attribute variables inside the Lintel package.

Figure 3.11b shows the relationships between several variables inside the lintel package. The lintel's length is the distance between its left and right edges. This relation is entirely inside or local to the lintel package; all variables related by the distance relation are contained in the lintel package.

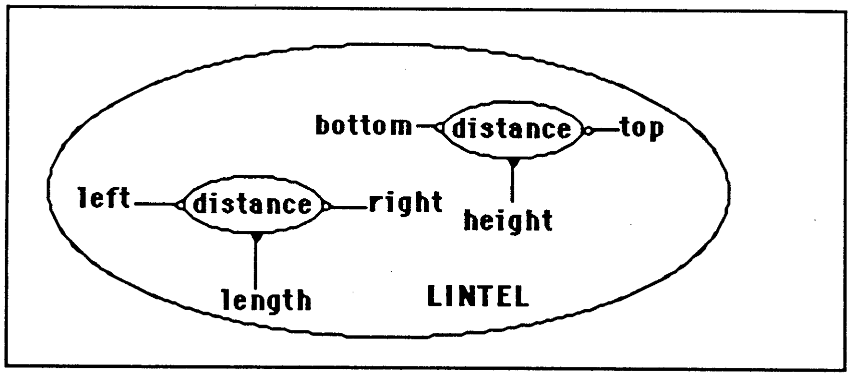
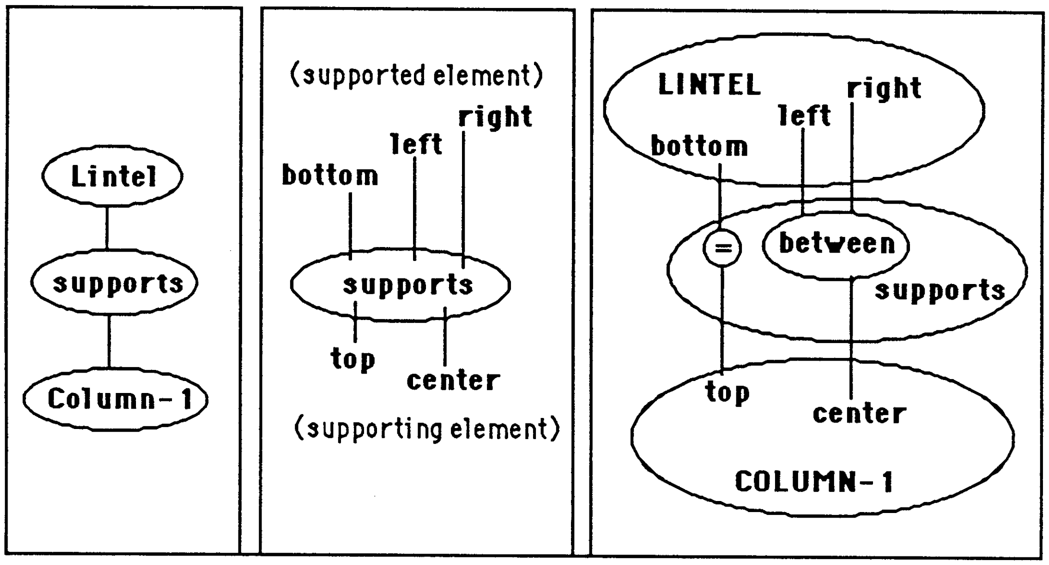


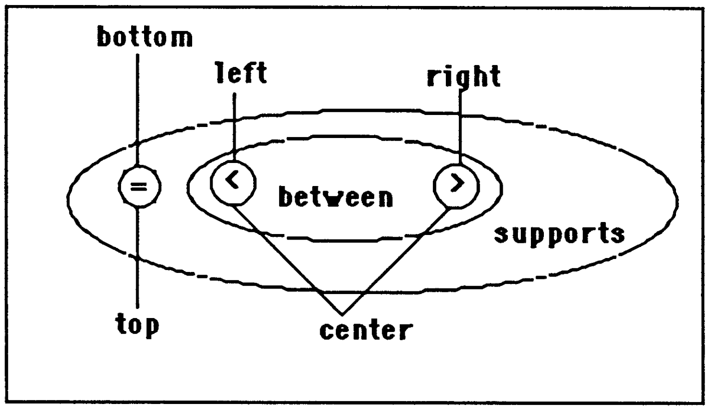
Figure 3.11b: Attribute variables and constraints inside the lintel package.

Like the lintel, each "supports" relation in Figure 3.9 is a package of more primitive components. The "supports" constraint relates two elements, in this case, a column and a lintel (see figure 3.12a). The supports constraint may apply between any elements that have the neccessary attributes for the description, that is, the supported element must have variables representing the left, right, and bottom edges, and the supporting elements must have center and top edge variables (figure 3.12b).



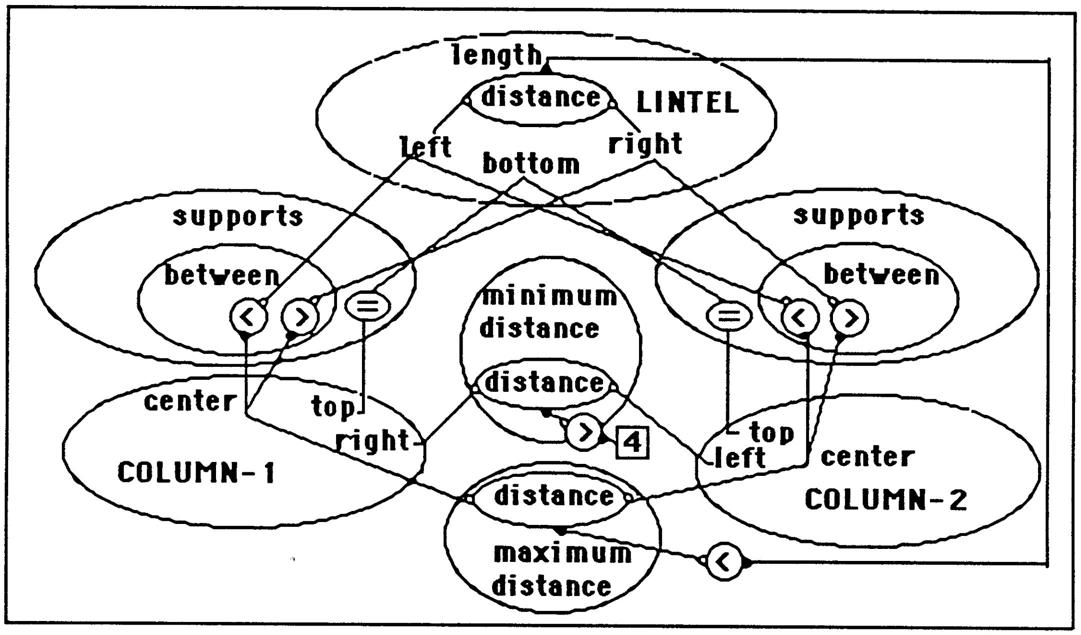
The supports relation in detail.
Figures 3.12a, b, c

Figures 3.12c illustrates the constraint explorer's expanded description of the "supports" package as position relations between variables of the two elements. "Supports" contains two simpler position constraints. The first constraint is that the top edge of the supporting element (the column) and the bottom edge of the supported element (the lintel) must be at the same height, or EQUAL in position. The second constraint is that the center of the supporting element must lie BETWEEN the right and left edges of the supported one. Figure 3.12d shows that the "BETWEEN" constraint, like "SUPPORTS", is a package of more primitive constraints "<", and ">". Here, these constraints may be read "left of", and "right of".



Opening up the "BETWEEN" package inside "SUPPORTS"
Figure 3.12d.

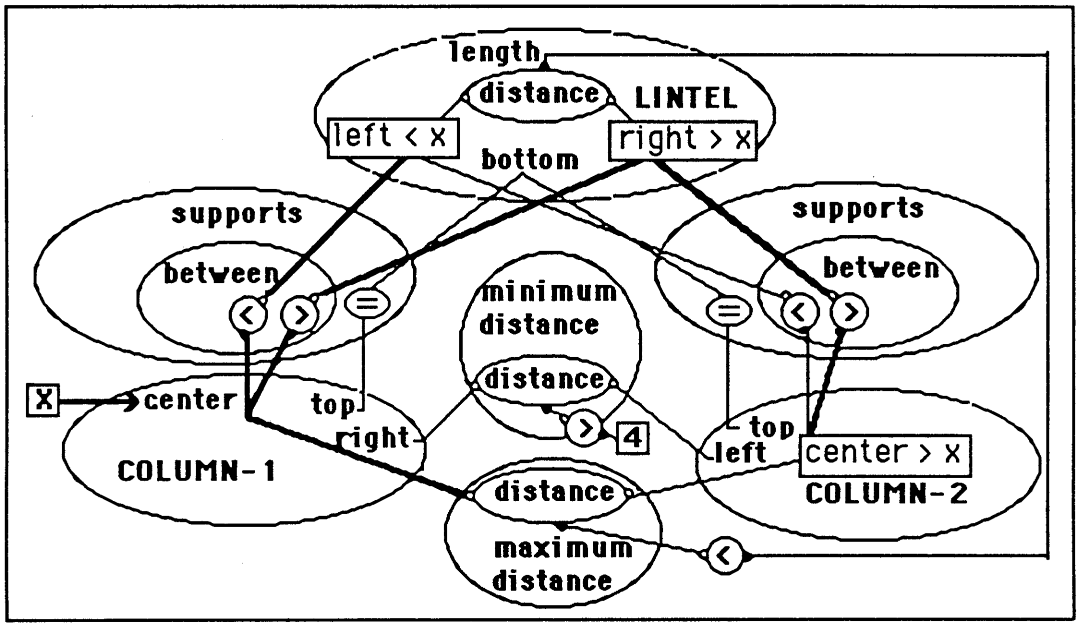
We are now ready to expand the simple model shown in figure 3.10. Figure 3.13 shows how the constraint explorer expands both the element and the constraint packages. Column and Lintel descriptions each have variables representing their edges, the Lintel also has a length variable, and each Column has a variable representing its center. The "supports" relation is expanded into more primitive position relations as in Figure 3.12c and 3.12d, and the "minimum-distance" and "maximum-distance" packages are opened up to reveal a distance computation and an inequality constraint.



Expanded description of the arch.
Figure 3.13

On the basis of these constraint descriptions, the constraint explorer can perform various calculations. If we give the positions of the two columns, then the machine will calculate the effects on the lintel's length. If instead we give the position of one column and the lintel length , then the explorer calculates the other column's position. (In section 4.3 we shall see that under some conditions, the direction of calculation, or propagation of effects, may be restricted). Thus the constraint explorer can express each constraint from the different points of view of each of the variables it constrains. The first support constraint, for example, can be seen as both "column-1 supports lintel" and "lintel is supported-by column-1".

We next make two settings in the design. Figure 3.14a shows the description network of Figure 3.13 with a new setting injected: we set the position of the center of column-1 to X. Assume X names a previously defined position.



Injecting a new value into the network:
>> set (column-1 center) X.
Figure 3.14a

As the new setting is entered, the effects propagate through the design. Figure 3.14a shows the state of the design description after column-1 has been positioned and effects have stopped propagating. Heavy lines trace the effect of the setting outward from its injection into the network. Any new constraints on the value of variables are recorded on the diagram in place of the variable names. (Recall that "< " , and ">" mean "left of ", and "right of" respectively.)

Fixing the center of column-1 at position X has immediate effects on two other variables. The left edge of the lintel must be left of X, the right edge of the lintel must be right of X. Notice that although two variables related by the lintel's length-left-right distance constraint have received effects of the new setting (lintel left and lintel right), still no changes can be computed for the lintel's length. The lintel's length is computed by the distance between its right and left edges, and all that is known about the positions of the edges is that one of them is to the left of position X and the other is to the right. As yet, nothing can be said about the distance between the edges.

After setting the lintel's length, the design state looks like Figure 3.14b.

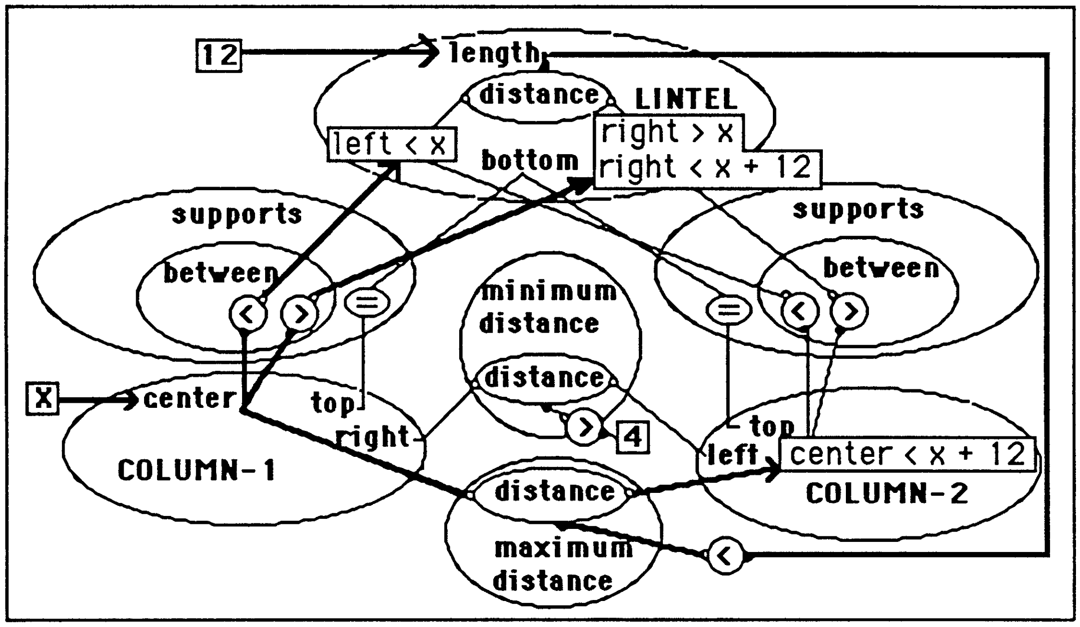


Figure 3.14b

The new value for Lintel length, together with the positioning of column-1's center affects one more variable, the center of column-2. The length of the lintel and the center of column-1 are given, and the explorer now limits the center of column-2 to be within one lintel-length of the center of column-1. That is,

(Column-2 center) $< X + 12$.

Notice that the effects of the change did not reach the tops or bottoms of any elements. Nor is the left edge of column-1 or right edge of column-2 affected. The center of each column is defined as being midway between its left and right edges. These constraints are illustrated in Figure 3.15. The center is equidistant from the left and right edges, and the distance between the edges is equal to the column's width. As we have not specified the column width, no effects of setting the center reach the position of the edges.

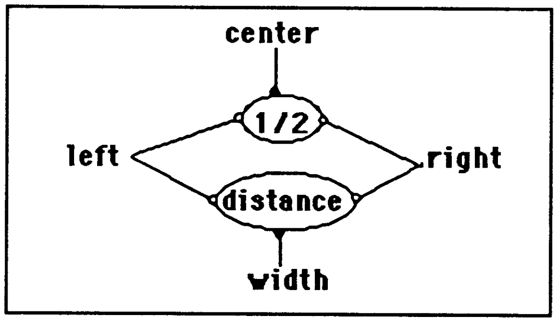


Figure 3.15

Notice also that had we chosen to first dimension the lintel, we would not observe any effects until we also had positioned one of the columns. The Lintel length is bound only by two constraints, the internal "lintel dimension" constraint that relates the lintel's left, right, and length variables, and the "maximum distance" constraint between the columns. Each of these requires another variable value in order to compute and transmit a change. Had we positioned column-2 instead of placing column-1, then a different though symmetric set of calculations would have occurred, as the reader may confirm.

So far the constraints in this model do not <u>fix</u> the values of variables; they only limit them. Even when we set the position of column-1 and the length of the lintel, the position of column-2 was still free to vary within a range of 4' to 12' from column-1. To determine the positions of elements, we might constrain the distance between the column centers to equal three times the distance from column centers to the lintel edges. Then setting the lintel's dimension will immediately determine both column positions.

We have seen that constraints and variables may be packaged together under one name to describe more complex constraints and variables. The elements and position relations used by the designer to describe the design are actually packages of more primitive relations between the most primitive relations are "+", ">", "=", "*". We have also seen how the machine uses the network description to determine the effects of changes in one part of the design on other parts. Effects of changes may propagate through the network in various directions outward from their origin. We saw also that a change only propagates throughout the design when a certain threshold of settings have been made.

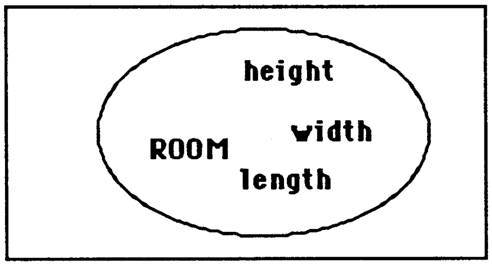
Two questions remain. How does the machine deduce new constraints, for example the maximum-distance constraint in Figure 3.10? How does the machine tracks dependencies between values in the design and explain the basis of its reasoning? These topics are treated in chapter 6, but a brief comment here provides a general idea of the mechanisms. Deduced constraints are added to the model by the program's symbolic mathematics routines (the solver), and dependencies among values are recorded with every setting and calculation. The constraint explorer uses the chain of dependencies that link design moves to provide explanations of the source of particular values and constraints, and also to minimize destructive effects of undoing design decisions.

3.3 Dimension Constraints.

We discussed in the previous section how the designer can package together collections of constraints and variables to describe an element or a relation and we introduced network diagrams as a way to understand the constraint explorer's representation of design rules. The following two sections present two classes of constraints of particular importance to the designer -- dimension and position relations.

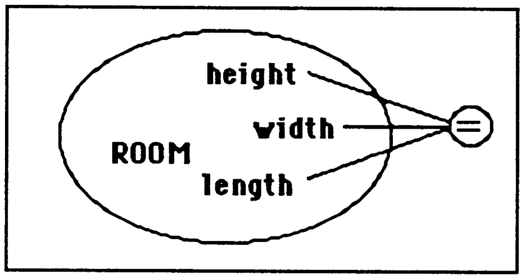
In the examples following we shall consider constraints on the dimensions of a room. Each of the following network diagrams represents a package of constraints, variables, and their connections that the designer might assemble to describe a design rule to the constraint explorer. Each package in these examples describes a different region of alternative room sizes.

Let us begin with a package that has no constraints, only variables. Figure 3.16 describes a room with three independent variables: length, width, and height. That is, the three variable values may be set independently and there are no limits on any of their values.



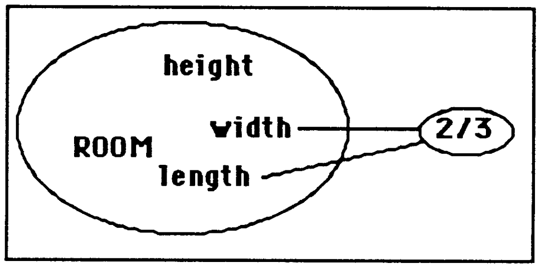
Room variables packaged; no constraints.
Figure 3.16

In Figure 3.17 we see a room now constrained to be a cube. Length, width, and height variables are all constrained to be equal. There is thus only one degree of freedom in this constraint; as soon as one variable's value is set, the design is completely specified. That value, however, may be freely chosen. The room must be a cube, but it may be a cube of any size.



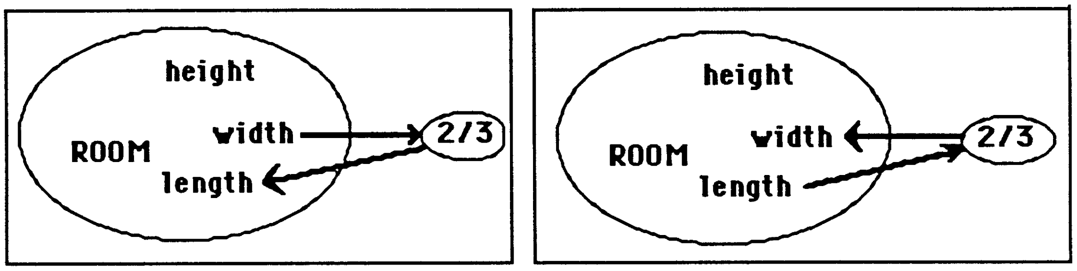
Cubical room.
Figure 3.17

Look at Figure 3.18a. It shows a room constrained to be exactly two thirds as wide as it is long. This package has two degrees of freedom: length and width are related by the two-thirds rule and height may be set independently.



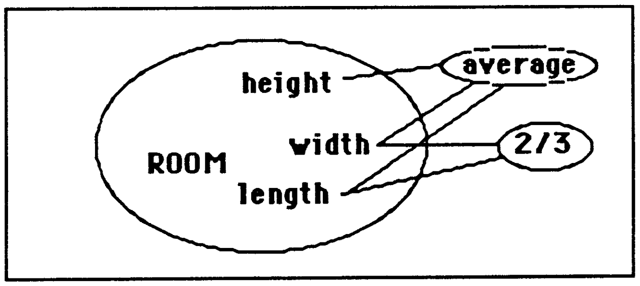
Two thirds proportion rule.
Figure 3.18a

Let us now see what happens when we fix, or set, a value that is connected through a constraint to other values in the design. When we set a value for the width variable, the constraint explorer calculates a value for the length. Alternatively when we fix a value for length, the constraint explorer calculates a value for width. Figure 3.18b shows the two different possible calculations with arrows indicating the direction of propagation. In addition, if either variable--length or width--in the proportion rule is connected also to other constraints, then any change may cause other calculations to be made, inducing a chain of consequences propagating outward from the originally set variable. Here we have considered only one constraint, but in Figure 3.19 we shall see an example of propagation through a network of constraints.

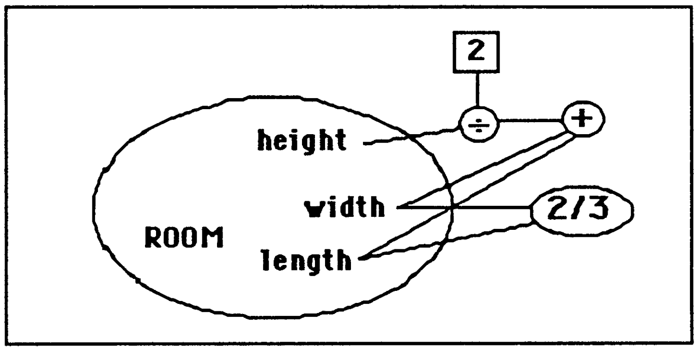


Two-thirds constraint calculates both ways.
Figure 3.18b

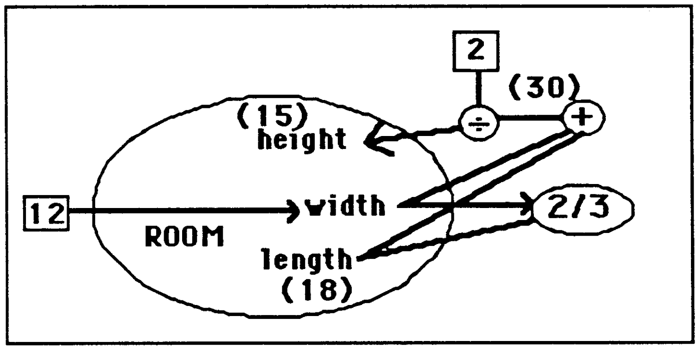
Our next diagram (Figure 3.19a) shows a room constrained to be two thirds as wide as it is long with its height constrained to be the average of its length and width. These constraints also have only one degree of freedom: fixing a value for any one of the variables will determine values for the other two. Figure 3.19b shows the "average" constraint expanded into more primitive components. For example, if we set the width to 12 feet, then the length is 18 feet (by the 2/3 proportion rule), and the height is 15, half the sum of width and height. This propagation of values is shown in Figure 3.19c. As with the cubical room, this room's dimensions are not limited to any numerical range; rather they are constrained to have a certain relation to one another.



proportional room with average height
Figure 3.19a

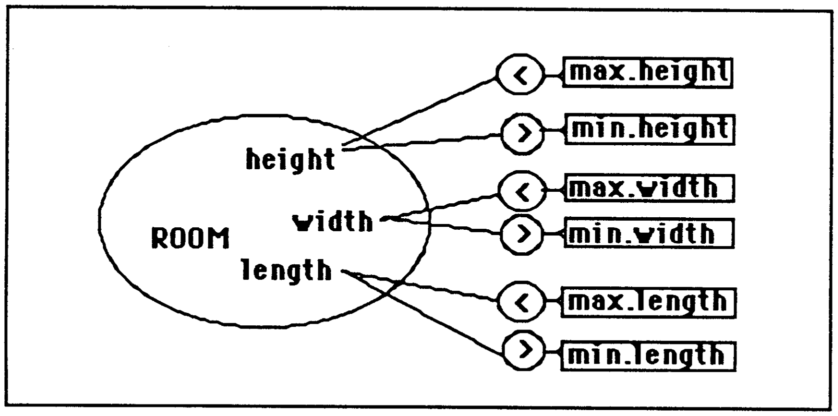


"average" constraint expanded to its primitive components.
Figure 3.19b.



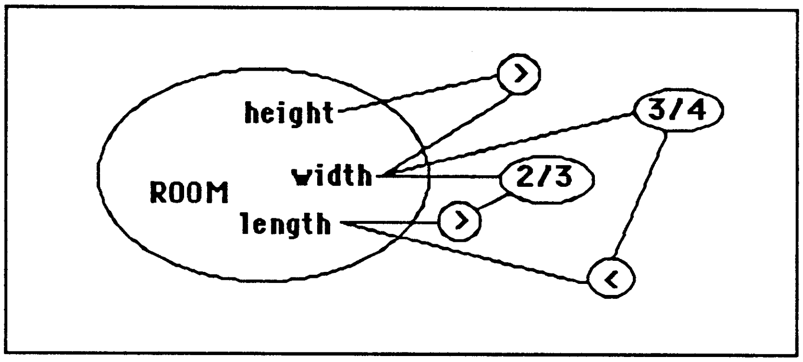
Propagating a value.
Figure 3.19c

Figure 3.20 shows a room constrained; not by proportion relations, but by maximum and minimum dimensions that are constants. There are three degrees of freedom; each variable is unrelated to the others and may be set independently. However, each variable is limited to values in a certain range.



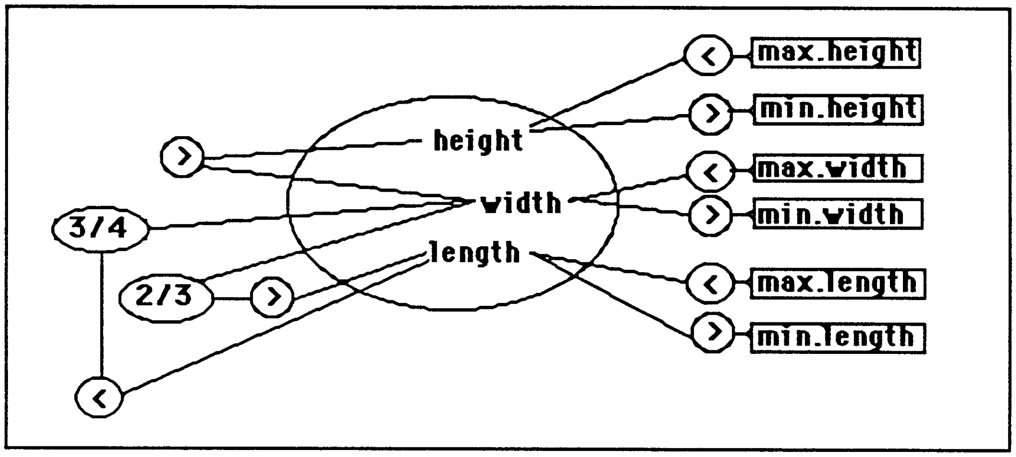
minimum and maximum dimensions.
Figure 3.20

In Figure 3.21, a room is constrained to be at least two thirds and no more than three quarters as wide as it is long, and at least as high as it is wide. Even though the variables are related, they may be chosen somewhat independently. These relations limit the range of values but within that range there are still three degrees of freedom.



range of proportions constraint.
Figure 3.21

Next, in Figure 3.22 we see a combination of the packages in Figures 3.21 and 3.20, representing the range-of-proportion constraint from Figure 3.21 and the minimum and maximum dimension constraint from Figure 3.20.



Combined range-of-proportion and range-of-dimension constraints.
Figure 3.22

3.4 Position Constraints.

Form, or the arrangement of physical elements in space, is the primary concern of architectural design. Position relations and rules are therefore of utmost importance to the architect. This section shows how position relations can be constructed by combining simple arithmetic constraints and relations. We shall consider a number of examples describing basic position relations increasing in complexity. Dimensions and positions are often interrelated and we shall see that combinations of position constraints may have dimensional implications. Later in this chapter, to illustrate the use of position constraints in design, we shall consider the effect of applying a particular constraint--a rule of thumb for placing columns one above the other in a column-and-beam type building.

We shall look at packages of constraints describing the relative positions of two elements, A and B. We will say nothing here about their dimensions; we are concerned only with describing their relative position in space. We shall approximate A and B as rectangular elements confined to a plane. The terms "above", "below", "left of" and "right of" name the four primitive position relations of the two elements. (In a three-dimensional version of the model we would also use relations "in front of" and "behind".) Each package contains simple combinations of the primitive position relations ("above", "below", etc.) between the elements' edges. Each example package shows for comparison: (a) a network of constraints and variables, (b) a representative variant or variants, (c) the same constraints and variables arrayed in a matrix, and (d) a symbolic (Lisp-like) representation. We begin with an extremely simple relation and gradually add and alter constraints.

We see in Figure 3.23a,b,c,d the position constraint, "A's right to the left of B's left". This relation is often abbreviated, "A entirely left of B". Five representative variants are indicated. A can be (1) "entirely above", (2) "(passing) above", (3) "enclosing", (4) "(passing) below", or (5) "entirely below" B in the vertical direction. Observe no dimensions are given; element A can be anywhere so long as it is to the left of B, A can be immediately or far to the left of B; it can be above or below, as figure 3.23b indicates. In figure 3.23c the black dot indicates a relation between the variable in the column and the constraint in the row.

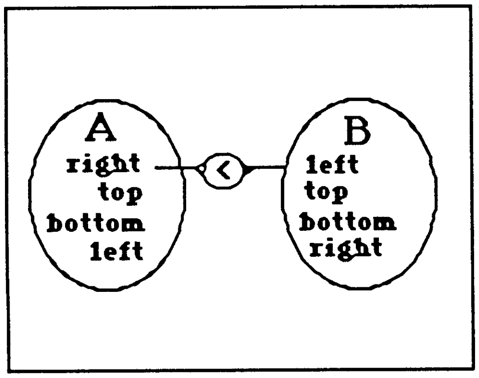


Figure 3.23a. network diagram.

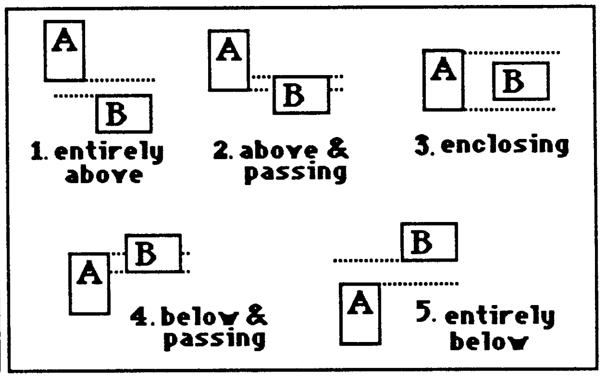


Figure 3.23b. representative variants.

A				B		
	t	b	r	t	b	l
<			●			●

Figure 3.23c. constraint/variable matrix.

(< (A right) (B left))

Figure 3.23d. Symbolic form.

A entirely to left of B
Figure 3.23

Remember the spatial uses of the "<", "=", and ">" symbols (section 3.2). Between right and left edges "<" means "left of", and between top and bottom edges "<" means " below". When the symbol "<" indicates a relation between two numbers, we read it "less than". Here, relating two element edges, it means one edge is "less far along" in some direction than the other edge. The symbol has a different meaning depending on context.

Here "<" measures distance along a direction. Likewise, ">" may mean "right of" or "above", and "=" may mean that two edges are at the same (horizontal or vertical) position.

The constraint packages in Figures 3.24, 3.25, 3.26, and 3.27 show how to distinguish some of the different classes of variants illustrated in Figure 3.23b. For instance, if we add one constraint to the package in Figure 3.23a, we select the class of variants in Figure 3.23b.5. The new constraint package is shown in Figure 3.24a. Now "A is left of and entirely below B" (Figure 3.24b). Another spelling, or name, for the same constraint is "B right of and entirely above A".

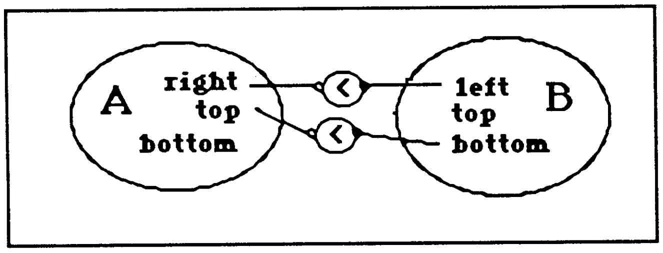


Figure 3.24a. Constraint Network.

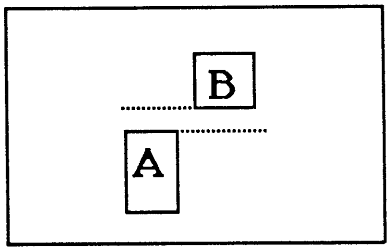


Figure 3.24b. Example Variant.

	A			B		
	t	b	r	t	b	l
<			•			•
<	•				•	

Figure 3.24c. Constraint Matrix.

```
(< (A right) (B left))
( < (A top) (B bottom))
```

Figure 3.24d. Symbolic form.

A entirely left of and entirely below B.
Figure 3.24

By adding two more constraints to the previous package (Figure 3.24) we describe a "passing" relation as illustrated in Figure 3.25. Element A must be below and entirely to the left of B. This corresponds to Figure 3.23b.4.

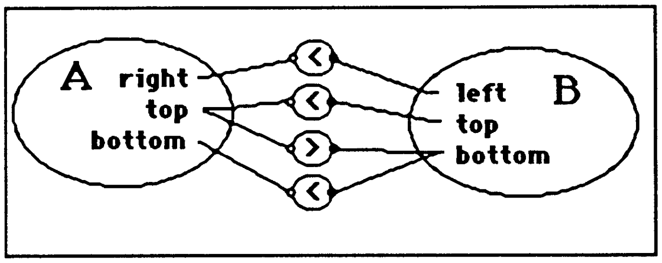


Figure 3.25a. Constraint Network

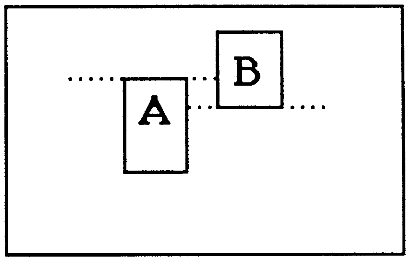
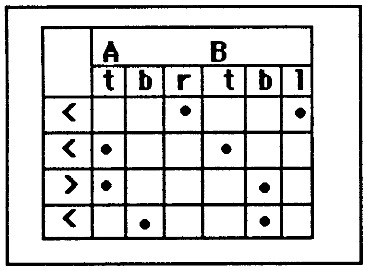


Figure 3.25b. Example Variant



	A			B		
	t	b	r	t	b	l
<			●			●
<	●			●		
>	●				●	
<		●			●	

```
(< (A right)(B left))
(< (A top)(B top))
(> (A top)(B bottom))
(< (A bottom)(B bottom))
```

Figure 3.25c. Constraint Matrix. Figure 3.25d. Symbolic Form.

A entirely left of and passing below B.
Figure 3.25.

A more general passing constraint is demonstrated in Figure 3.26. Two elements pass vertically but unlike the previous example, this constraint does not distinguish between A above B and B below A. This corresponds to the variants in Figure 3.23b.2 and 3.23b.4. Notice that the two cases, A above B, and A below B, are covered separately by two constraints (labelled 'passing above' and 'passing below') OR'ed together. Figure 3.26a' shows the two relations "passing-above" and "passing-below" expanded to reveal their component relations. Only one of the component constraints need be satisfied. In this case, A cannot be both above and below B at the same time. Constraints packaged with no explicit logical relation (such as OR in this example) are implicitly ANDed together--all constraints in the package must be satisfied.

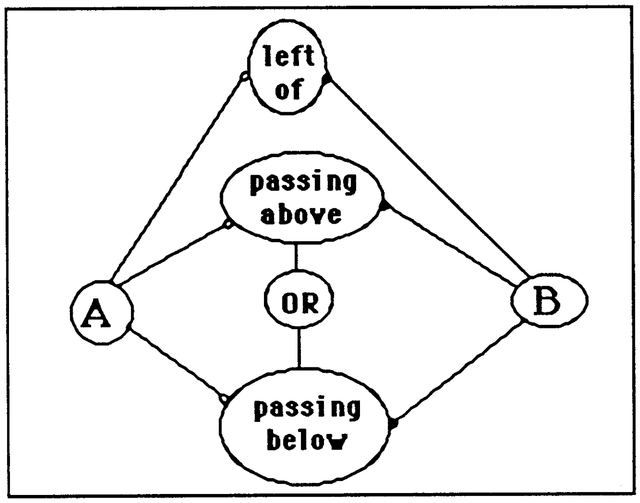


Figure 3.26a. Constraint Network.

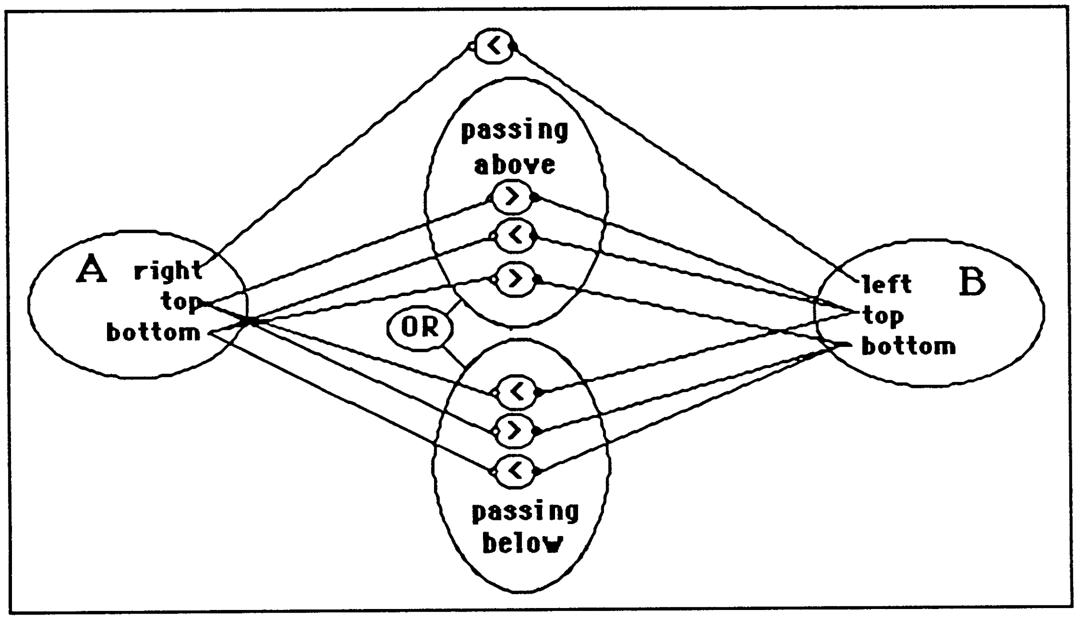


Figure 3.26a'. Constraint Network(expanded)

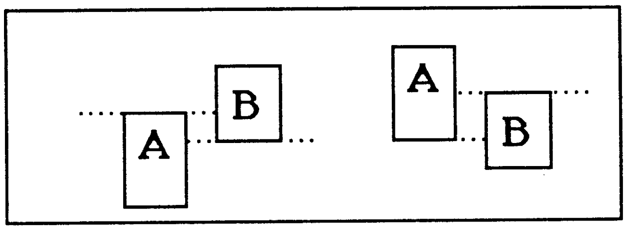
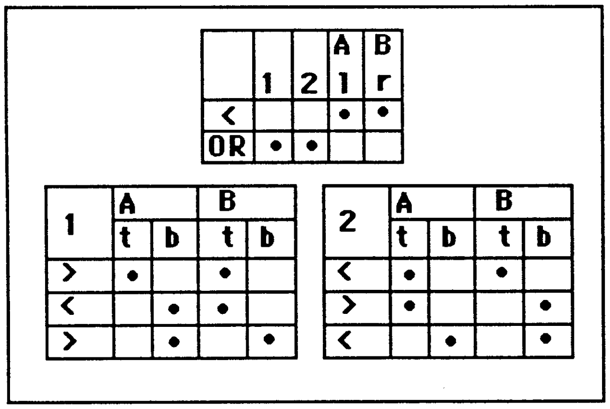


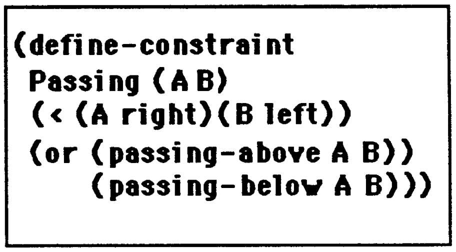
Figure 3.26b. Example Variants.



	1	2	A l	B r
<			•	•
OR	•	•		

1	A		B	
	t	b	t	b
>	•		•	
<		•	•	
>		•		•

2	A		B	
	t	b	t	b
<	•		•	
>	•			•
<		•		•

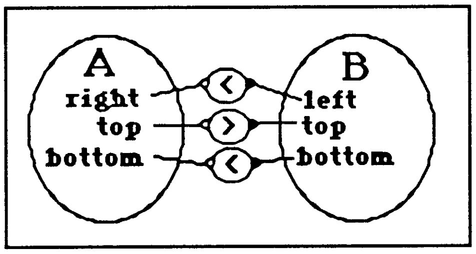


```
(define-constraint
 Passing (A B)
 (< (A right)(B left))
 (or (passing-above A B))
     (passing-below A B)))
```

Figure 3.26c. Constraint Matrix. Figure 3.26d. Symbolic Form

A and B passing.
Figure 3.26

Figure 3.27 shows A to the left of B, A's top above B's top, and A's bottom below B's bottom, corresponding to Figure 3.23b.3. In other words, B is entirely--both top and bottom edges--within the projection of A's right edge. This constraint implies that A's right side must be longer than B's left edge.



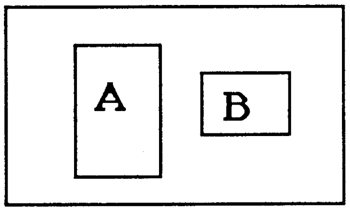
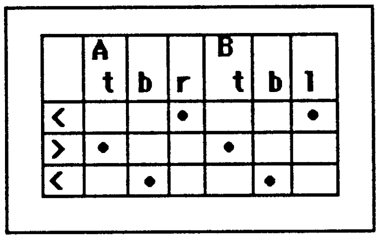
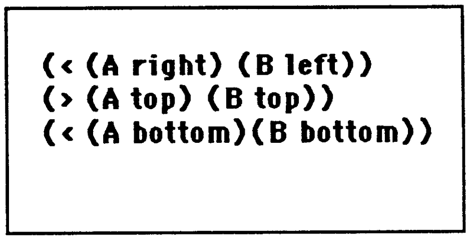


Figure 3.27a. Constraint Network. Figure 3.27b. Example Variant.



	A			B		
	t	b	r	t	b	l
<			•			•
>	•			•		
<		•			•	



```
(< (A right) (B left))
(> (A top) (B top))
(< (A bottom)(B bottom))
```

Figure 3.27c. Constraint Matrix. Figure 3.27d. Symbolic Form.
A to the left of and vertically enclosing B.
Figure 3.27.

In Figure 3.28 we see a description similar to the one in Figure 3.27, except A is offset from B by a minimum distance, or offset. The distance between A's right edge and B's left edge must be greater than the minimum offset. This package involves a dimension as well as a position constraint. The offset dimension may be left as a variable, set to a constant, or, as we shall see in a moment, related to some other quantity in the design.

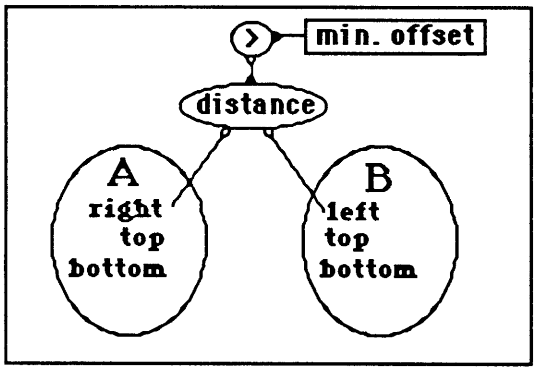


Figure 3.28a. Constraint Network.

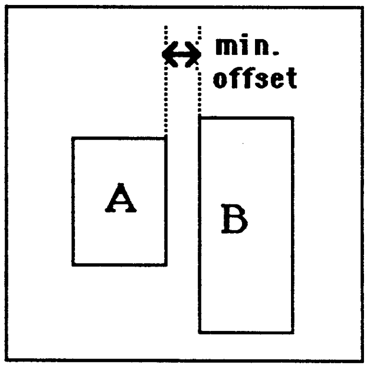


Figure 3.28b. Example Variant.

	A r	B l	min. offset
> dist	•	•	•

Figure 3.28c. Constraint matrix.

```
(> (distance (A right) (B left))
   minimum-offset)
```

Figure 3.28d. Symbolic form.

A left offset from B by minimum dimension.
Figure 3.28.

In the next example, A is offset from B by at exactly its own width dimension (Figure 3.29). The offset that was a free variable in the previous example is now connected to A's width.

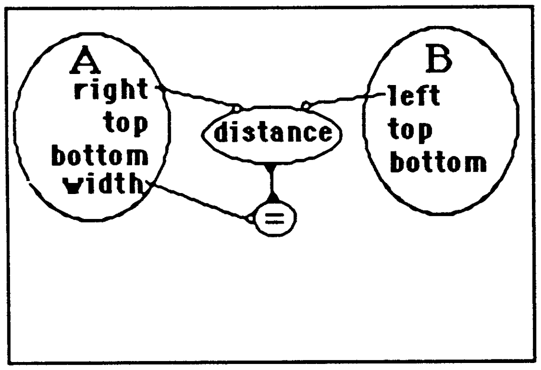


Figure 3.29a. Network

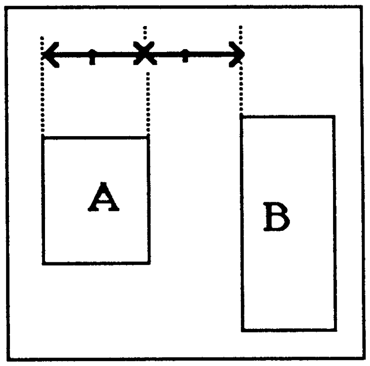


Figure 3.29b. Variant

	A		B
	r	w	l
=dist	•	•	•

Figure 3.29c. Matrix

```
(=(A width) (distance (A right)(B left)))
```

Figure 3.29d. Symbolic Form.

A left offset from B by A's width.
Figure 3.29

Figure 3.30 shows a package of constraints describing the containment relation, A inside B. For this constraint to make sense in two dimensions, B must represent a space element.

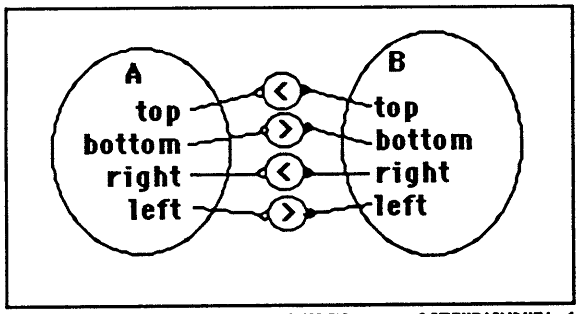


Figure 3.30a. Network.

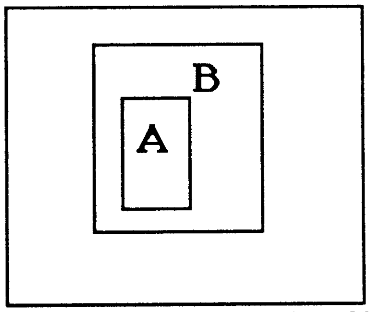
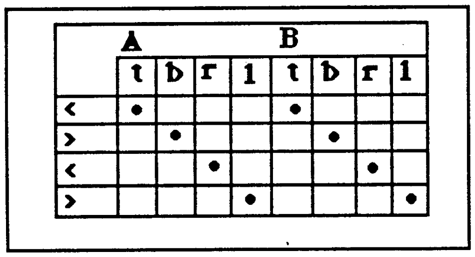
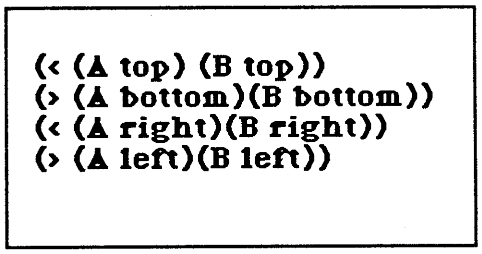


Figure 3.30b. Variant.



	A				B			
	t	b	r	l	t	b	r	l
<	●				●			
>		●				●		
<			●				●	
>				●				●

Figure 3.30c. Matrix.



```
(< (A top) (B top))
(> (A bottom)(B bottom))
(< (A right)(B right))
(> (A left)(B left))
```

Figure 3.30d. Code.

containment - A inside B.
Figure 3.30.

Figure 3.31 shows the constraint, A cross B. Again, either A, or B, or both A and B must be space elements for this package to make sense in two dimensions.

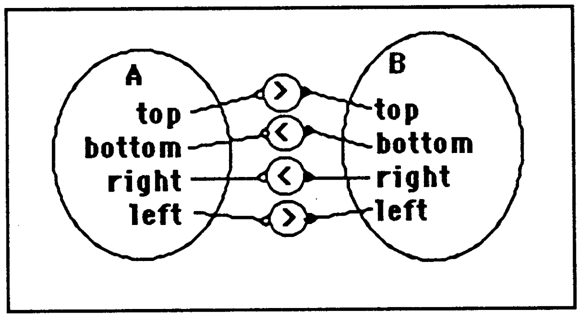


Figure 3.31a. Network.

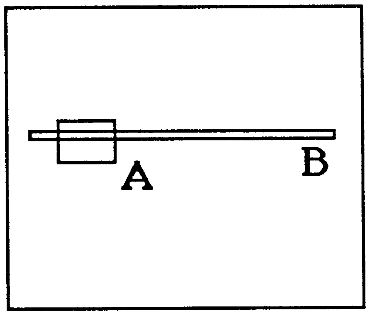
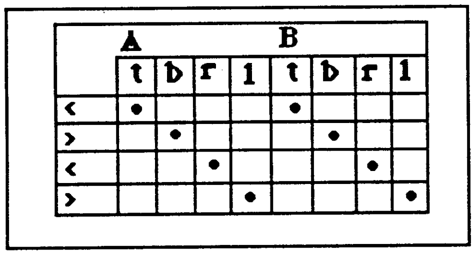
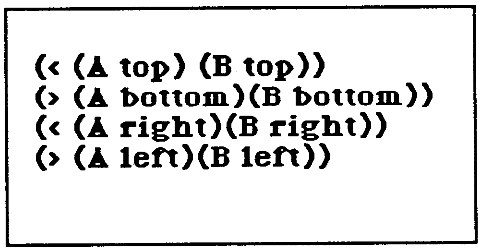


Figure 3.31b. Variant.



	A				B			
	t	b	r	l	t	b	r	l
<	●				●			
>		●				●		
<			●				●	
>				●				●

Figure 3.31c. Matrix.



```
(< (A top) (B top))
(> (A bottom)(B bottom))
(< (A right)(B right))
(> (A left)(B left))
```

Figure 3.31d. Symbolic form.

A cross B
Figure 3.31.

Until now we have not mentioned any constraints on direction, although only orthogonally oriented variants have been illustrated. We have been using absolute names for position relations between elements and for their edges. Our absolute names assume we are looking at a vertical section drawing. We have been using terms such as "top", "above", and "left of" to indicate the relative positions of elements as seen by the reader of the drawing. Thus, in order to simplify the explanation, we have been assuming that all elements face "up". We now introduce a direction variable for each element. An element's direction is represented by a number from 0 to 360, like a compass heading. By convention, in plan an element's direction is along its longer axis, and is indicated in diagram by an arrow. The direction of an element may be constrained in much the same way as the dimension and the position. For example, the constraints in Figure 3.32 allow up to a 20 degree difference (or turn) in the directions of A and B.

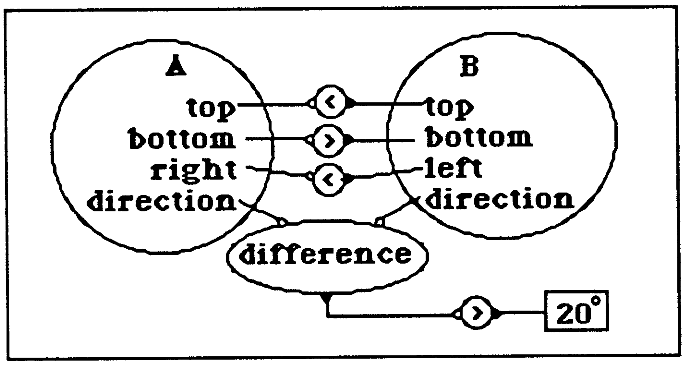


Figure 3.32a. Network.

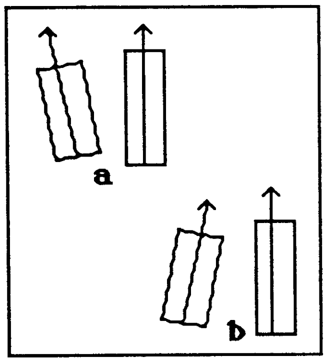
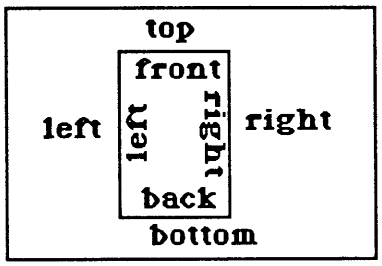


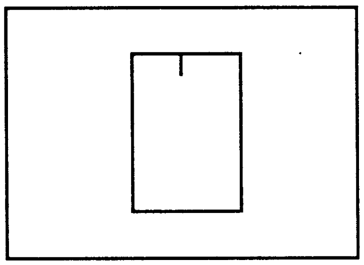
Figure 3.32b. Variants.

Adding an orientation constraint.
Figure 3.32.

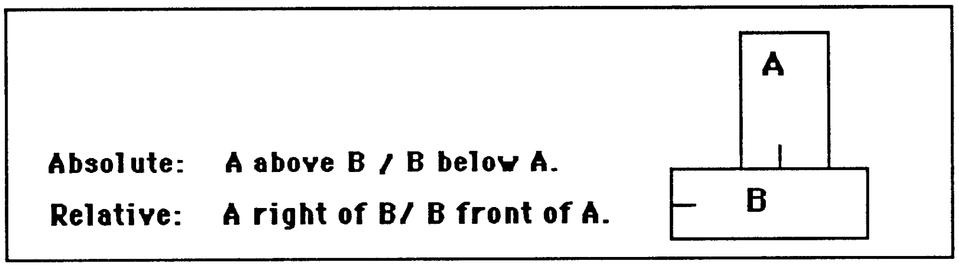
Permitting the directions of elements to vary, we see that our absolute naming system is less than ideal. For example, as element A rotates clockwise, its "top" edge gradually becomes its right edge. Figure 3.33 shows another set of element edge names: front, back, left, and right. These names are not absolute, but relative to the element's direction. The front edge of an element is the edge towards the element's direction. As Figure 3.33a shows, if the direction of an element is "up" (in the absolute naming system) then the absolute edge names we have been using (top, bottom, left, right) map directly to the local names (front, back, left, right). If we wish to use relative names for edges we can use a tick mark to indicate the front edge of each element (figure 3.33b).



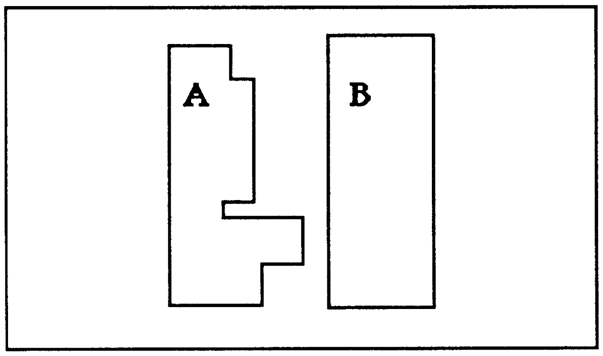
absolute and local names of edges
Figure 3.33a.


tick mark indicates front edge.
Figure 3.33b.

However if we rotate two elements A and B so that their directions are different, we can compare absolute and relative naming systems. The configuration in Figure 3.34 is described using the two reference systems.


Absolute: A above B / B below A.
Relative: A right of B/ B front of A.
A
B
absolute and local names.
Figure 3.34.

This concludes our examples of position constraint models. We have seen how to construct a variety of moderately complex position constraints between two elements from a small set of more primitive relations. A number of simplifications have been made for the purposes of explanation. More than two elements may be related by a single position relation, as we shall see next. Also, we have assumed that elements are rectangles, although we know that often they are not. More sophisticated position rules may relate an element's shape and its position. For example, a position rule might say "orient A's variable edge toward B" (Figure 3.35).

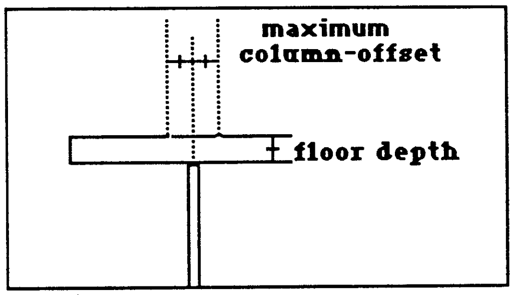


A's variable edge toward B.
Figure 3.35.

3.5 A Position Constraint.

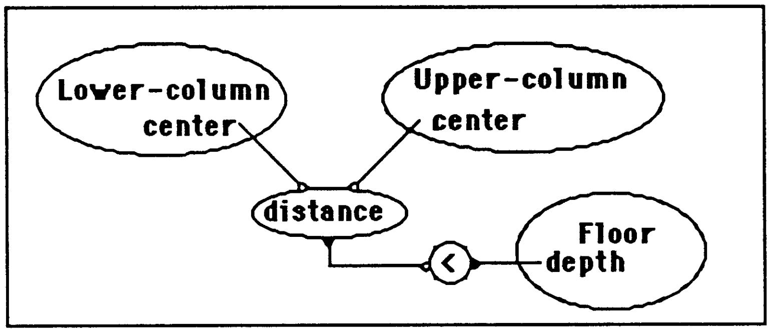
To gain a sense of how the sorts of position constraints described above might function in a design, we next discuss the application of a particular position constraint in some detail. We consider a simple structural constraint, expressed as a rule of thumb for positioning columns. The purpose is not to argue the merits and disadvantages of a particular constraint, but to see how a constraint fits into an architectural context. It is interesting that such arguments can be expressed concisely using a vocabulary of constraints.

When designing with a one-way column-and-beam construction system the columns (that carry the building) must be positioned according to general and particular constraints. Columns that support one floor must themselves be supported by the columns on the floor below. Although every column must be located over a beam, columns need not be stacked directly one atop the other; rather, each may be laterally shifted along the beam upon which it rests, so long as the beam can transfer the load onto the column below. As a rule-of-thumb the columns may be offset along the beam by a maximum distance of the beam depth (figure 3.36). The exact safe column offset depends on the particular beam -- its shape in cross-section and its material -- as well as the load carried by the upper column. The rule-of-thumb is usually, however, a good approximation. In the larger design other constraints also limit the positioning of columns; for example, the maximum beam span and the room layout constraints.



Maximum column-offset equals floor depth.
Figure 3.36.

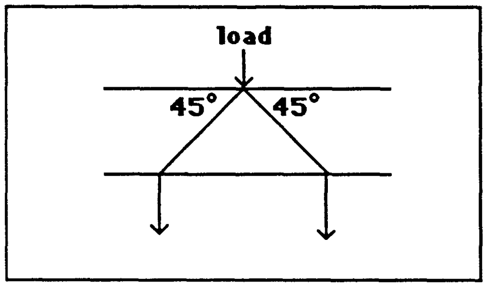
Figure 3.37 shows a model of this constraint. Notice that this constraint concerns variables from three elements: the floor depth, and the positions of two columns.



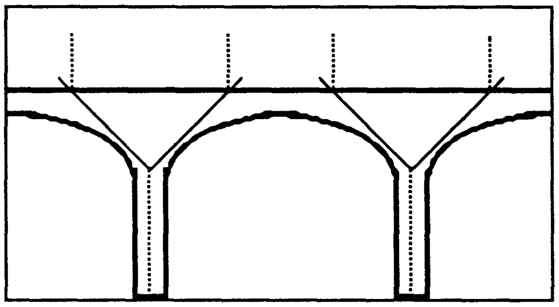
Column-offset constraint - network diagram.
Figure 3.37.

Most modern column-and-beam designs do not exploit this column-offset constraint. Instead, columns are usually placed one atop the other even though they might be offset without violating any structural constraints. Although "stacked atop" is only occasionally necessary where an exceptionally heavy load must be carried, many designers use the more restrictive "stacked atop" constraint as a default. The column-offset constraint is perhaps a more reasonable default than the stacked-atop constraint, as the former permits a wider choice of column position than the latter, without significantly reducing structural integrity.

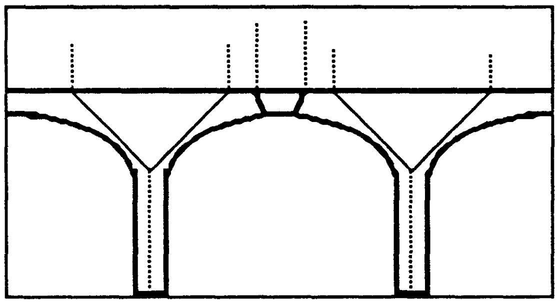
Alexander recommends an architectural rule or "pattern" similar to the column-offset constraint, applied to locations of columns when a construction system of vaults is employed [Alexander, Ishikawa, & Silverstein 77]. In both vaults and column-and-beam construction the applicability of the constraint is based on a structural principle that the beam can safely support a load through an angle as low as 45 degrees from the vertical (figure 3.38). Thus in a vault or true arch the maximum column-offset distance could be greater than with the column-and-beam system; that distance would be limited by the intersection of the 45 degree tangent to the curve of the arch with the top surface of the floor (Figure 3.39). Bracing can also be used to extend the limit in a column-and-beam system. A true arch or vault, unlike a column-and-beam system, can also support a column at its center, or key (Figure 3.40).



Beam can support loads through a 45 degree angle.
Figure 3.38

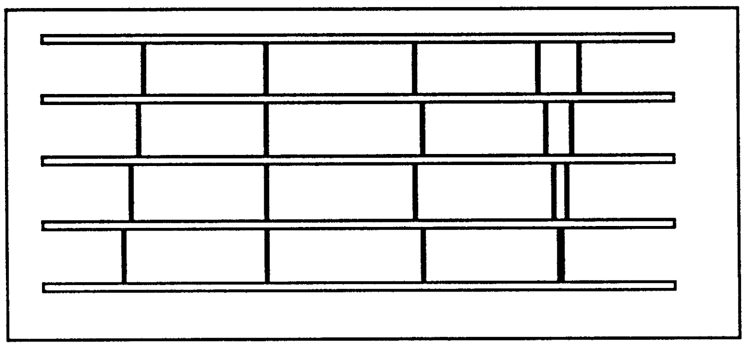


Column-offset in a vault.
Figure 3.39



Arch can also support at key.
Figure 3.40

The column-offset rule is an example of a <u>local position constraint</u>. It is local because each application or instance of the constraint concerns the positions of two particular columns and a beam. As it applies to every pair of columns where one supports the other this local constraint can, however, have far-reaching implications on the design of a building. The constraint allows a variety of beam spans and column configurations; see for example figure 3.41.



Configurations of vertical support
Figure 3.41.

In these examples we have seen only local position rules. The same sorts of constraints can be used to describe more global constraints. For example, all elements in a field may be constrained to have approximately the same direction. Grids and reference-lines can be used to establish a global position rule. Certain elements may be limited to certain grid intersections, or to lie in the zones between certain reference lines.

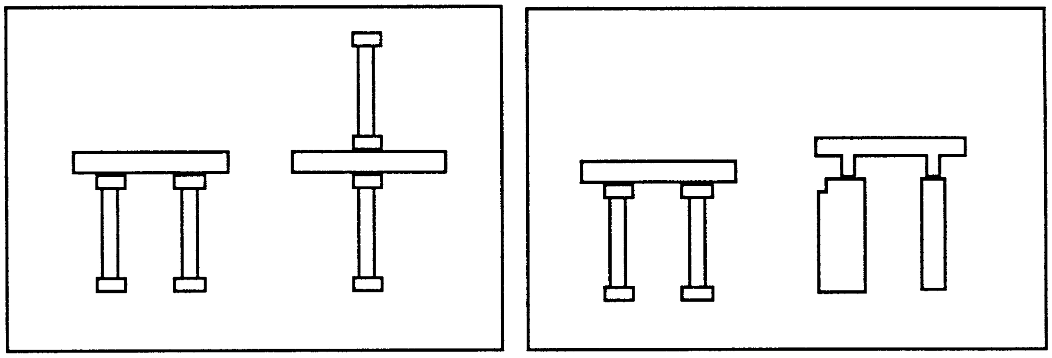
CHAPTER 4

Parts, Prototypes, and Dependencies

4.1 Elements and Configurations.

So far we have been looking at descriptions of designs in terms of position and dimension constraints on elements. These constraint descriptions, however, are not the whole story. For not only do elements have position and dimension constraints, they are also organized into a hierarchy of parts and configurations. In the description of the "arch" we saw the position constraints between its parts, the two columns and lintel, but we did not move up in the part/whole hierarchy to view the arch as a part of a larger configuration, or down to view each of the arch's parts as a configuration of elements. The hierarchy of elements and configurations is another aspect to the description of a design, distinct from the constraints on the properties of elements. The two descriptions are, as we shall see, related. We shall also see that a set of constraints may describe a range of different configurations, and that a configuration of elements may be described, or read, in several different ways.

A configuration is a set of elements with certain position relations. An element may be either a space, or virtual element, or it may be a physical, or material one. In any case we call the "parts list" of a configuration its "selection" and we call the set of position relations between its parts its "distribution" [Habraken 83]. For example, two different configurations may have the same selection of elements but different distributions, as Figure 4.1a illustrates, or as in Figure 4.1b, they may have the same distribution but different selections.

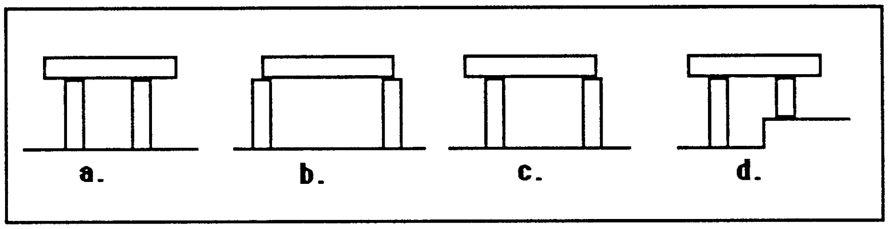


4.1a: Same selection, different distribution.
4.1b: Different selection, same distribution
Figure 4.1a, b

Most elements are themselves in turn configurations of smaller elements. We are all familiar with the idea of a hierarchy of parts and wholes, in which a complex element--a house for example--is decomposed into parts and parts of parts. Ultimately the complex configuration "house" is seen to be composed of a large number of simple, undecomposable elements, such as columns, beams, bricks, sticks, and nails. Though all decompositions of a configuration comprise the same set of most basic elements, the decomposition of a complex configuration into parts and sub-parts is not always unique. Conventionally, however, one decomposition of configurations and elements is preferred. An element can also belong to more than one configuration. For example, a light switch may belong to the configuration "built elements in the living-room" as well as to the configuration "parts connected electrically to the main power-line".

A configuration is described by its selection and its distribution. The selection identifies the elements in the configuration. The distribution consists of position constraints on the elements named in the selection. Each element in the selection may itself be a configuration of smaller elements. Constraints describe the positions of the parts relative to one another, but <u>not</u> relative to the configuration. A configuration and its parts are really the same entity;

therefore they cannot enter into position relations with each other. A set of constraints need not describe a single and unique configuration; it may describe a range of possible alternative configurations, or variants. For example, in section 3.1 we described a set of position constraints on three elements, two columns and a lintel. Those position constraints described the configuration, "arch". As the constraints permit freedom in both the dimensions and positions of the parts, they do not describe one single arch but a range or region of possible arches. Figure 4.2 shows four different extreme arches described (that is, not excluded) by those constraints.



A constraint description may permit variants of a configuration.
Figure 4.2

Although earlier we described the position constraints between the columns and the lintel elements (section 3.1), we did not view the columns and the lintel as configurations themselves. Now we shall look at these elements in greater detail. We may describe a column, for example, as a configuration of three parts: a base, shaft, and capital, centered and stacked one above the other. Figure 4.3a shows the parts of a column in the hierarchic description of an arch, and Figure 4.3b shows the constraints that describe the position relations between the three parts. In Figure 4.3b the small numbers next to the "stacked" constraint indicate the order of stacking. Figure 4.3c shows the position constraints between the parts of the column in greater detail.

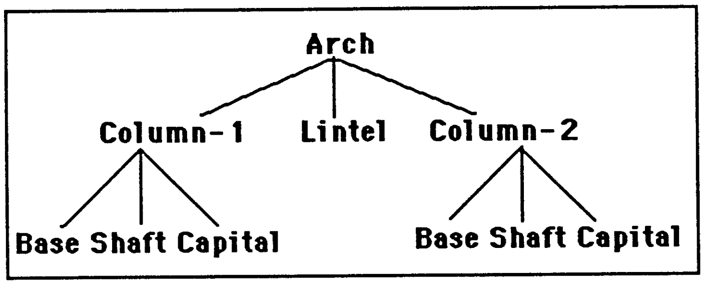


Figure 4.3a. Selection of parts in "Arch".

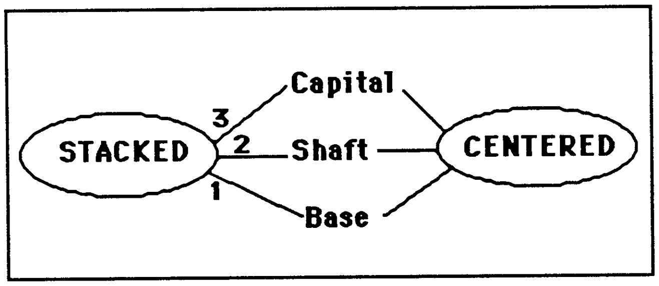


Figure 4.3b. Distribution of parts in "Column".

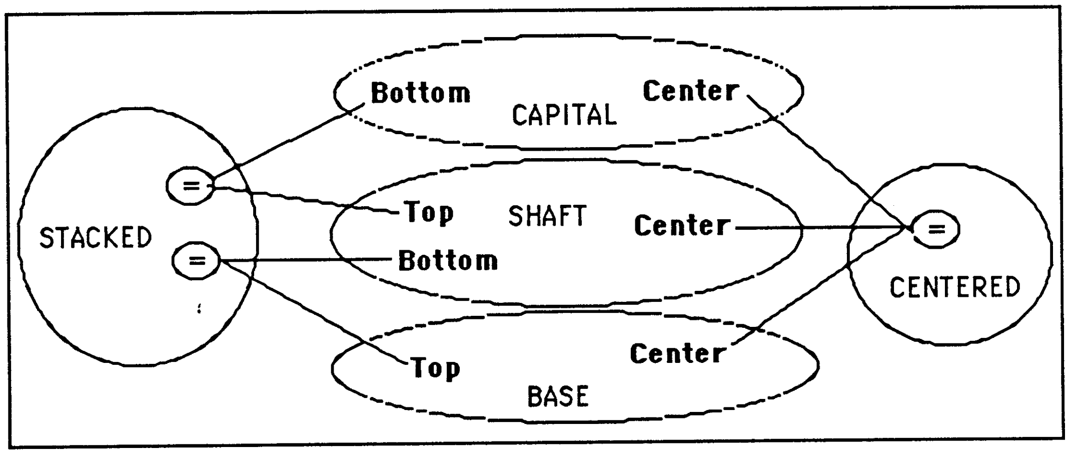


Figure 4.3c. Detailed distribution constraints of "Column".

Parts of a column and their distribution.
Figure 4.3a, b, c

There may also be dimension and proportion constraints that define each of these parts. In Figure 4.4, position constraints are shown at the left and dimension constraints are shown to the right. Figure 4.4 illustrates, in addition to the "stacked" and "centered" constraints of Figure 4.3, the following dimension and proportion rules:

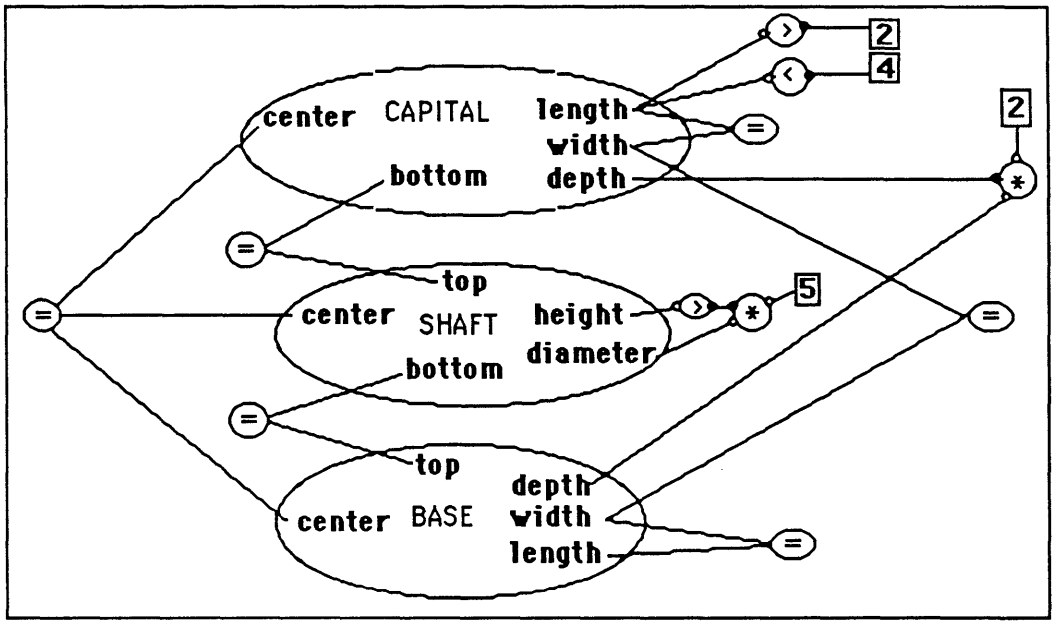
A) base and capital must be square in plan (their width and length dimensions must be equal),

B) the base and the capital have the same dimension in plan (here their widths are constrained to be equal),

C) the base and capital length must be between 2 and 4 feet,

D) the depth of the capital is one half the depth of the base, and

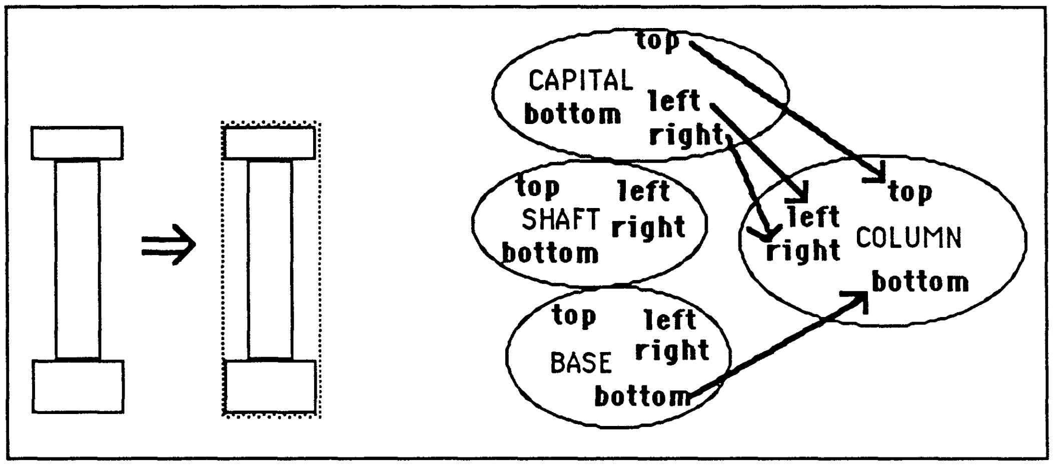
E) the column's shaft is at least five times higher than its diameter.



Constraints between parts of a column.
Figure 4.4

Previously we described a column as though it were simply a rectangle, having top, bottom, left, and right edges (see section 3.2). We were only interested in the column as a part of the arch; we were content to see it as simply an element. Now we have described the column in greater detail, seeing it as a configuration of parts.

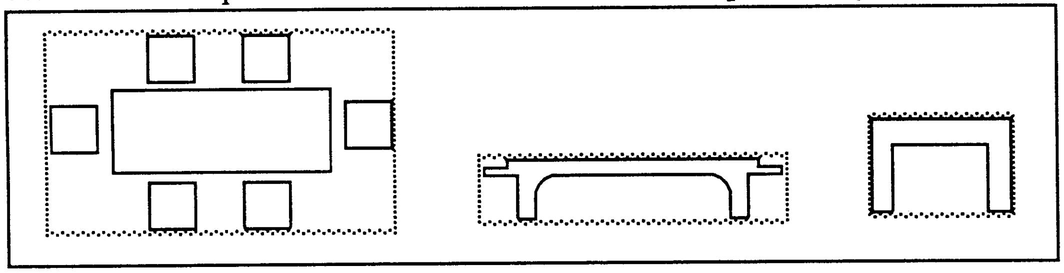
Our next step is to connect, or merge these two descriptions; that is, to integrate the description of "column-as-part-of-arch" with the more detailed description of "column-as-configuration-of-parts". We would like the constraint explorer to understand that the two descriptions correspond to the same element. We need to specify, for example, that the top edge of the column in the first description is the same as the top edge of its capital in the detailed description. Our integration also shows that the column's left and right edges are the same as the left and right edges of the base and capital (figure 4.5).



Integrating the detailed description of the column.
Figure 4.5

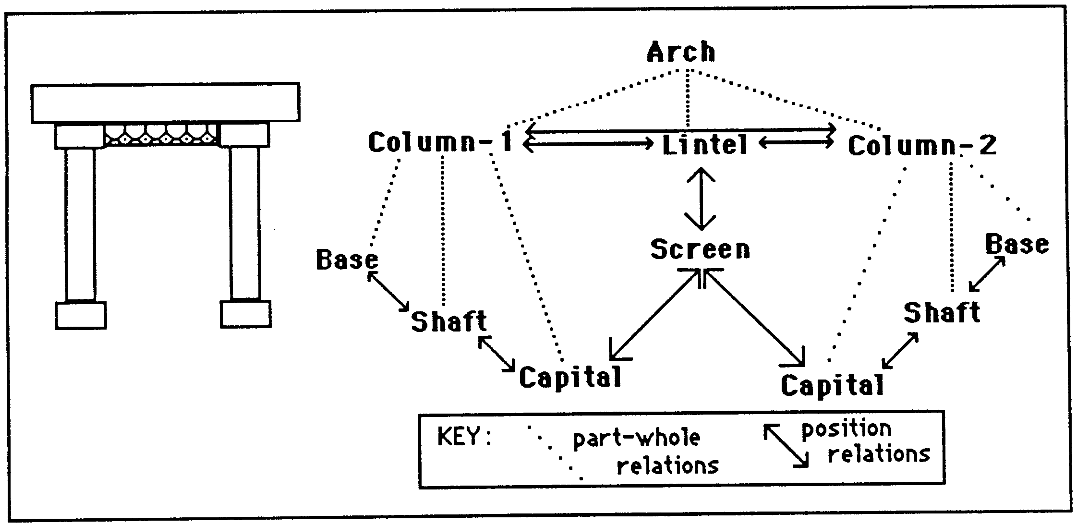
In general we will always work with descriptions of a configuration at two different levels of detail: one describing the parts and their position relations and another describing the configuration as a part in some other, larger configuration. The latter description will usually only concern the outside edges of the configuration, and not position relations between its parts. We would like to automatically generate simple descriptions from detailed ones. We might use a 'bounding-box' heuristic: select the outermost edges of the configuration's outermost parts and call these the edges of the configuration in the simple description. Figure

4.6 shows this scheme applied to several configurations. In some cases, of course, we may want to override this procedure and reduce the detailed description using different heuristics.



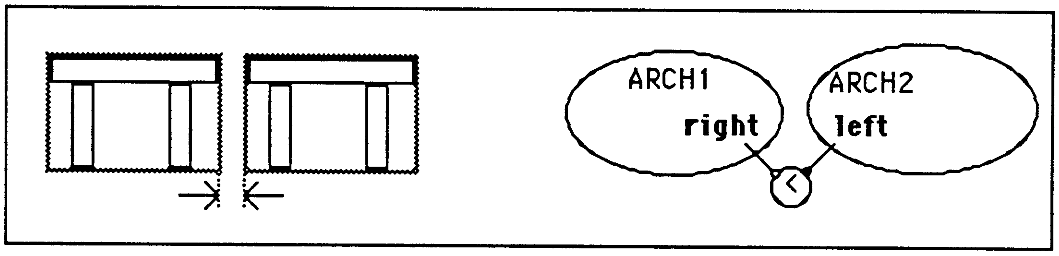
Bounding-box reduces detailed descriptions to outermost edges.
Figure 4.6

Often we shall want to position an element relative to some part of a configuration. For example, we might want to place a screen, or window, between the capitals of the columns of an arch, directly beneath the lintel (Figure 4.7). The position relations here concern the edges of the screen and the edges of the capitals of the columns of the arch. Now we are three deep into the part/whole tree of the arch: the arch itself, the columns of the arch, and the capitals of the columns of the arch. Figure 4.7a shows the form of the configuration and Figure 4.7b diagrams the position relations between its parts. Notice that there are two sorts of relations between elements in this diagram: part-whole relations and position relations.



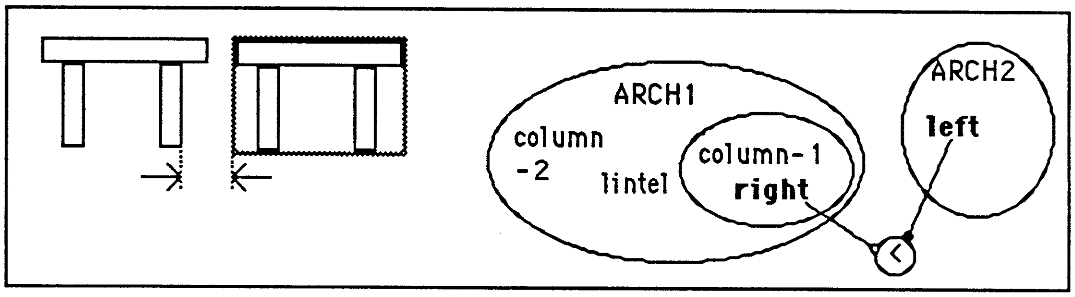
Screen between capitals of columns of arch.
Figure 4.7a, b

The following example shows the same idea--that we shall sometimes want to describe the position of one element or configuration relative to another element that is a part of a configuration. Let us consider two different ways to describe a linear series of arches or "arcade". In Figure 4.8, the distribution constraints of the arcade are position relations between successive arches.



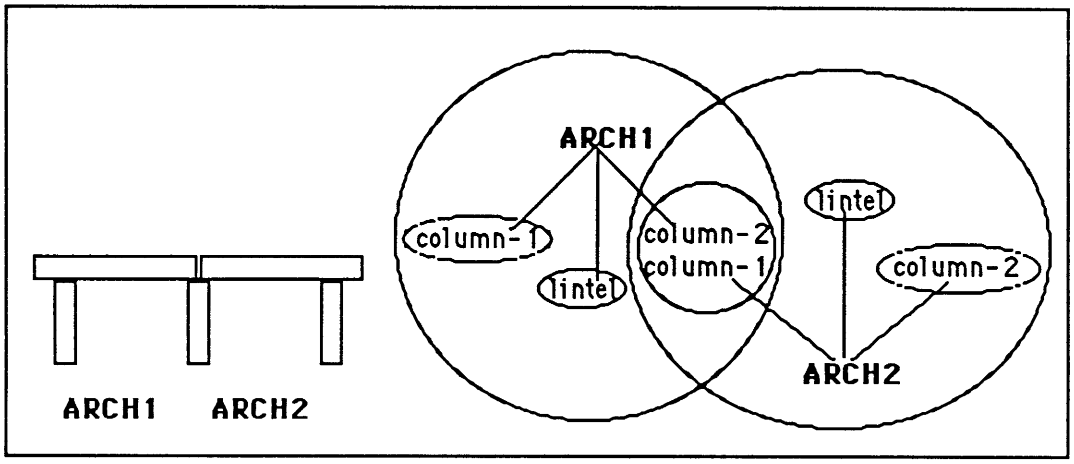
Arcade described as position relations between successive arches.
Figure 4.8

Figure 4.9 describes the same configuration as figure 4.8. Here, however, Arch2 is positioned relative to column-2 in Arch1, and not relative to the configuration Arch1 itself.



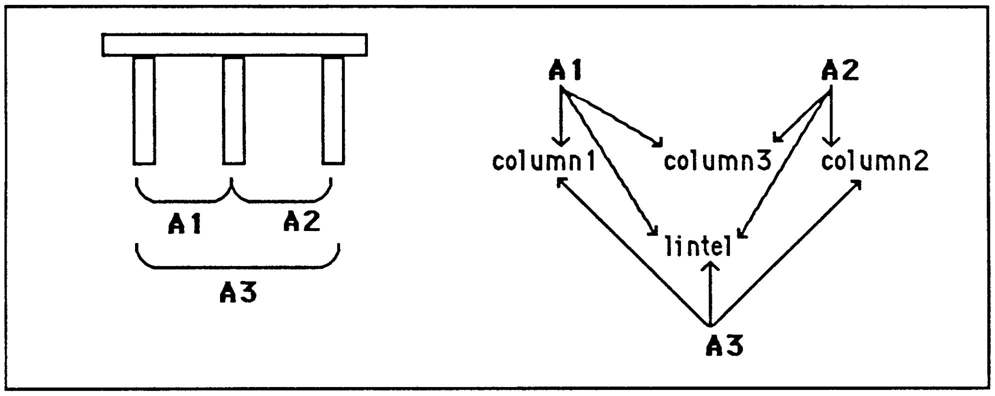
Alternate description of the arcade.
Figure 4.9

The next example shows that different configurations may have common elements. Figure 4.10 shows two arch configurations that share a column. The shared column is known by a different name in each configuration.



Two arches share a column.
Figure 4.10

Figures 4.8 and 4.9 illustrated the idea that there may be several different ways to read, or describe a configuration, all of them equally valid. Figure 4.11a shows another example of this multiple view idea: a lintel with three columns that may be read either as two arches (**A1** and **A2**) sharing a middle column, or as a single arch (**A3**) with a third column placed between its supporting columns. Figure 4.11b shows the different part-whole relations in these three arch configurations.



Three arches sharing a lintel.
Figure 4.11a, b

4.2 Prototypes, Instances, and Individuals.

We have seen how elements and position relations can be described by using packages of constraints and variables and how packages can be organized in a part-whole hierarchy to describe nested configurations of elements. We now examine a different organization of these same packages--a hierarchy based on distinguishing prototypes, instances, and individuals. This structure allows the designer to define new packages by adding distinctions to existing ones.* Descriptions higher in this prototype hierarchy are more general; lower ones are more specific and detailed. All but the most specific packages in the prototype hierarchy refer to classes of elements or relations. For example, having laboriously constructed an "arch" package by describing all the parts and their relations (as in section 3.1), the designer can easily make arch-instances that share the general description, yet each instance may vary in its particular details. Each may also be a prototype for further sets of even more particular arch-instances.

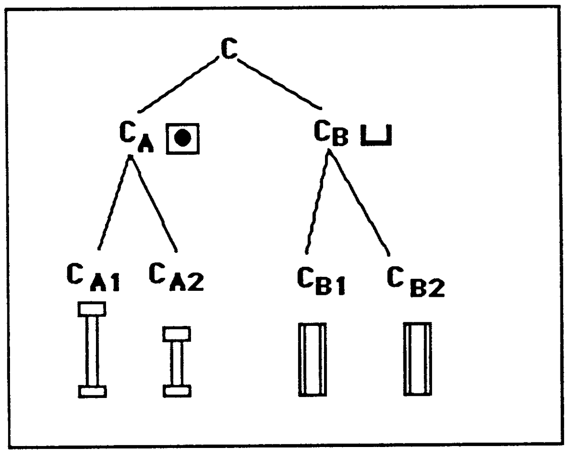
We shall use the terms "prototype", "instance", and "individual"† as follows. Each higher description in the hierarchy is a "prototype" for all its inferiors. The top, or root, description is therefore an ultimate prototype having no prototype of its own. Each lower

* Generic, or classification hierarchy is a central feature in many "knowledge representation languages" and increasingly becoming standard in general purpose programming languages as well. Here the issues are simplified extremely for the sake of brevity. A clear and simple exposition may be found in Bobrow and Winograd's Overview of KRL [Bobrow and Winograd 77]. The interested reader should refer to the programming languages Simula, Smalltalk, and Objectlisp [Dahl and Nygaard 66, Goldberg and Robson 84, Drescher (forthcoming)], and also to the large body of work in knowledge representation. Two good initial references are Representation & Understanding [Bobrow and Collins 77], and the February 1980 special issue of the ACM SIGART newsletter [Brachman and Smith 80]. A different perspective on these issues is afforded in NETL- A System for Representing and Using Real-World Knowledge [Fahlman 79].

† Often the words "class", "subclass", and "instance" are also used for the same concepts, respectively.

description is an "instance" of its superiors. Instances at the bottom, or fringe, have no sub-instances. We call these ultimate instances "individuals".*

If further specifications are not made, the properties of an instance are the same as those of its prototype. We say that each instance "inherits" its prototype's properties. An instance may also carry additional properties that supplement and/or supercede its prototype's properties. If any property is superceded, we say the instance is an exception to its prototype. The entire set of properties of any instance is the union of its private properties and the properties it inherits from its prototype. In case a property is assigned to two instances, the lower value always applies. In figures 4.12 and 4.13, prototype Column C has two instances C_A and C_B. In turn, C_A and C_B are each prototypes for two instances, individuals C_{A1}, C_{A2}, C_{B1}, and C_{B2}.



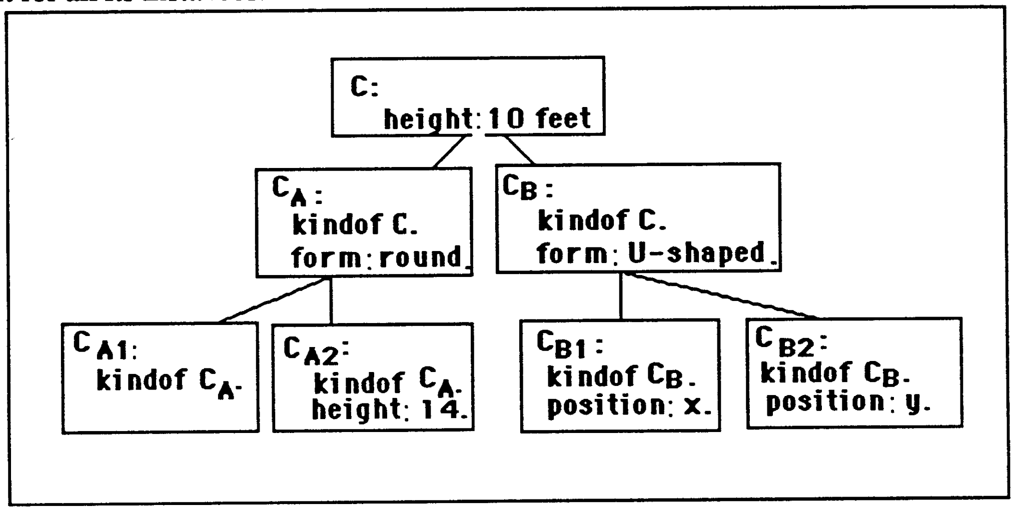
Simple prototype hierarchy.
Links represent the "a-kind-of" relation between element descriptions.
Figure 4.12

Figure 4.13 shows the properties attached to each description in the hierarchy. The properties of each instance can be found by reading up the chain of its prototypes, adding

*Although we shall not discuss it here, instances are not limited to a single prototype. Thus the prototype hierarchy is actually a lattice. The prototype-inheritance mechanisms of the constraint explorer are those of Objectlisp, and are discussed in The Objectlisp Manual [Drescher (forthcoming)].

properties from each description that are not defined in a lower instance. C_A and C_B differ in their form but both inherit their height from C, the shared prototype. C_{A1} and C_{A2} differ in height but share the same form; C_{B1} and C_{B2} are alike except for position.

Look at the individuals in Figures 4.12 and 4.13. C_{B1} and C_{B2} specify different positions but they do not override any properties of their prototypes. C_{A1} inherits C_A's form and C's height; C_{A1} has no private properties (except for its built-in kindof property that establishes its prototype). In contrast, C_{A2}'s private height supercedes the value specified by C. We can think of C_{A2} as an exception to its prototype C* or we can think of C's height as a default for all its instances.



Prototype hierarchy showing properties of elements.
Figure 4.13

A prototype serves as the default description for all its instances. Defaults are useful when we want to begin with a standard description for all instances and (possibly) return later to specify or change descriptions of particular instances. Of course the constraint explorer allows the designer to override the initial defaults.

* Notice it is possible in this scheme to construct a individual that is an exception to all properties of its prototype. Indeed we have one in CA2 with respect to C. We call such an individual a "perfect exception".

The prototype-instance hierarchy pervades the constraint descriptions we have been using all along. For example, Figure 4.13 shows both part-whole relations between elements in an arcade as well as the prototypes shared by the various parts. (Position relations are not shown.) We now see that each arch (A1, A2,...) is an instance of a prototype arch A0, and likewise each column (C1, C2,...) is an instance of a prototype column C0; each lintel (L1, L2,...) is an instance of a prototype lintel L0.

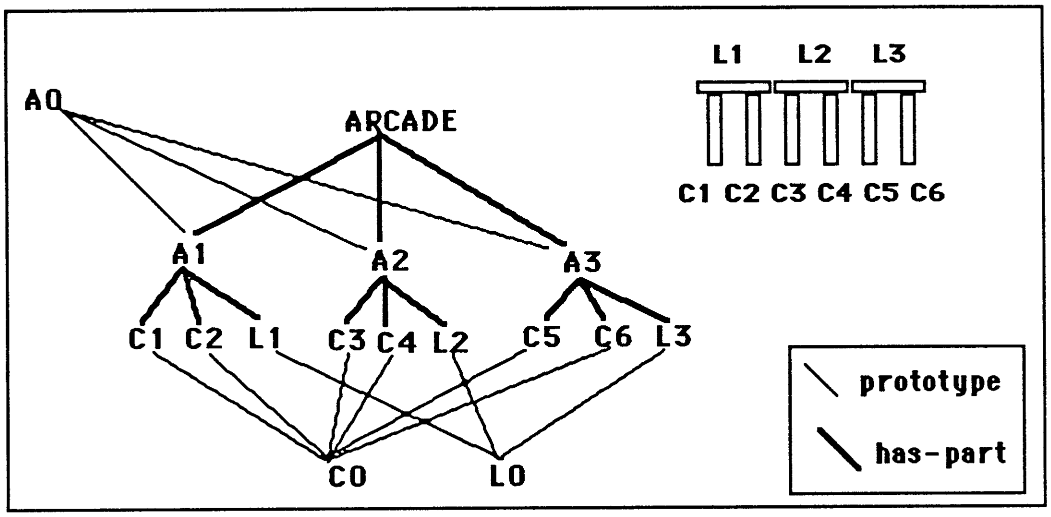


Diagram showing both part-whole and prototype-instance relations in an arcade.
Figure 4.13.

In figure 4.13 the three arches A1, A2, and A3 are identical; in general this need not be the case. Each arch in the arcade may differ, so long as each arch meets the constraints in the prototype A0, and similarly all the columns and lintels meet the constraints in their prototype.

The prototype hierarchy organizes packages of constraints and variables. We have seen how one class of material elements, columns, might be structured. The prototype hierarchy may also be used to specify relations as well as elements. For example, we may begin with a prototype "A to the left of B" relation, and specify different variations as in Section 3.4. Each variation of the relation can be described as an instance, or specification, of its prototype. Structuring packages into a hierarchy or lattice of prototypes and instances is an efficient way for the constraint explorer to store information, and it seems a natural way for the designer to describe the design as well.

4.3 Dependencies in the Built Environment and in Design.

In the built environment, position relations may be ordered by dependencies. If moving one element disturbs the second while the second may be moved freely without disturbing the first, then a dependency relation exists between the two elements. One element supports, contains, and/or supplies another. Such dependencies between elements may be a function of the element pair, or determined by the relative placement in space of the elements. Different actors, or powers may control elements of different inherent dependency levels. For example, one designer may design the major bearing walls and another may design the infil walls and screens. Inherent dependencies of elements thus order the relation between designers that control the elements. These dependencies between elements are especially important in architecture and in any design domain dealing with the arrangement of forms in space.*

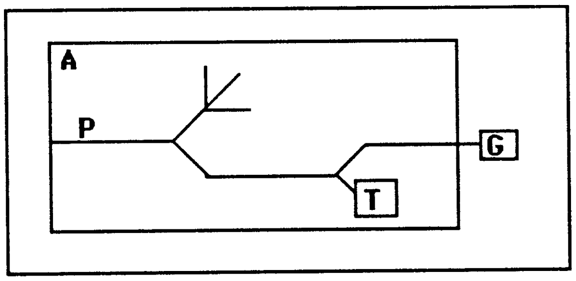
In the course of building, dependencies between elements arise naturally. For example, the roof depends on the bearing walls for support; the door depends on its frame for support. In the supplies relation, the light depends on the power-cord for electricity; the pipes supply the faucet with hot and cold water; the gutter drains the roof. Examples of the enclosure or containment relation are: the room contains the closet; the apartment contains the room; the apartment building contains apartments and a lobby. In each of these examples, one element depends on the other but not the reverse. The roof may be moved without disturbing the bearing walls, the faucet replaced without disturbing the pipes, and the apartments remodelled without changing the volume of the building.

*Dependencies in the built environment are a major topic in Habraken's Transformations of the Site [Habraken 83]. Therefore only a summary appears here.

The "contains" relation is slightly different than "supports" and "supplies" because containing elements must be space elements, not material ones. Space elements exist only by virtue of our convention to recognize them--they are indicated by their bounding elements. A room, for example, is indicated by its walls; a street, by the buildings on either side. Unlike a brick or a beam, a space element cannot be picked up and moved about on the site unless it is part of a prefabricated element. During design however, space elements are manipulated in the same ways as material elements. Containment also has territorial implications, but those are beyond the scope of the present work.

The "support", "supply", and "contains" relations are specific kinds of dependency relations. Each is associated with a position relation. Of course, an element may be in relation with more than one other element, and may at the same time depend on different elements in different ways. For example, figure 4.14 shows power line (P) contained in an apartment (A), supplying a toaster (T) that is also contained in the apartment. Garden light (G) is also supplied by (P). Concisely stated, these relations are:

```
(A contains P)
(A contains T)
(P supplies T)
(P supplies G)
(not (A contains G))
```



Elements may have several dependency relations.
Figure 4.14

Some position relations between elements are not ordered by dependency. Consider the typical position relation between a table and a chair. There is no dependency; each can be moved independently. We can add a constraint that the chair must be at (within a certain distance from) the table, thus introducing a dependency between the two elements. This is dependency by fiat.

We can also introduce a dependency by placing the chair on top of the table. Now the table supports the chair: the position of the chair depends on the position of the table. If we move the table, the chair will move but if we move the chair the table will not. By rearranging the elements we have changed their dependency relation. We can reverse this dependency by placing the table on four chairs (or on a single chair if the table has one leg in its center). Then we can freely move the table without disturbing the supporting chair(s), but the reverse cannot occur. This second kind of dependency between a table and a chair is a matter of the relative positions of the two elements.

A third kind of dependency may be inherent in the selection of elements in a position relation. Unlike the table and chair example, some pairs of elements always have the same dependency relation, unaffected by their relative position. Because of their physical properties certain elements always depend on certain other elements--a dependency that cannot be reversed. We say then that the elements belong to different "levels" of form. A level is a class of elements that have the same inherent dependency behavior. For example, screen elements such as windows and doors all belong to the same level. Levels can be ranked. The more changeable elements are lower-level elements. For example, heavy masonry always supports wood framing elements--never the reverse. These two classes of elements are at different levels. Pipes always supply water faucets and never the reverse. Again the dependence between elements is inherent, indicating their relative permanence or changeability.

Architects often indicate elements of different levels by drawing them in different colors (or different line-weights, or on separate overlays). One color indicates the foundations or building footprint, another indicates the wood frame, a third indicates movable partitions. Although the three systems--foundation, frame, and partitions-- are inherently dependency ordered, each system has its own set of dimension and position constraints. And although one depends on the other, still slack remains in the position relation. For example, although the foundation supports the wood frame, requiring the wood frame to be within a certain distance of the foundation, within this distance the frame may still vary its position relation to the foundation.

We now consider some implications of dependency and levels for the designer. Dependency between elements is part of the physical behavior of the built environment; the designer must understand it in order to operate successfully in it. The designer need only understand dependency in practical terms: "if I move this element, what other elements must move?". Levels of form are physical realities on the site but not on the drawing board. They are especially useful to the designer who recognizes that a design does not often remain as built for its lifetime, but rather is subjected to constant change and adaptation throughout its lifetime. For example, the lot lines in a layout of houses generally remains constant while within each lot a house is built, altered, and demolished. Another example is modern office building design where often the building shell and infrastructure is designed by one architect and the territories occupied by different tenants are designed separately. Some designers (the SAR architects certainly, but also Hertzberger, Kroll and others) use elements of different levels to ensure that their buildings are adaptable for change.* For example, such a designer will often work on

* Design education at M.I.T. has placed especial emphasis on this distinction, both in the "built-form" design attitude exemplified by the teaching and works of M.K. Smith, and in N.J. Habraken's "thematic design" workshops.

configurations of the different levels separately. Other designers are not explicitly concerned with this distinction and do not use it to order their work. The designer need not recognize different levels of elements in order to design, although it can certainly be useful to do so.

In some situations the designer operates only on one level; for example the designer who lays out interior, infil partitions to make apartments within a given building shell operates only at the level of infil partitions. But more often and more interestingly the designer controls elements on at least two levels. For example, the same designer who decides on the positions of bearing walls may later place the infil partitions that divide the space between bearing walls into rooms and apartments. One use of this "levels" concept is in the testing of design alternatives. Design decisions at one level are constraints for the next lower level. A measure of flexibility of a design decision is the freedom it leaves for the next designer, or user, who operates at the next lower level of elements. In general we can rank variants of a higher-level system by the variety of lower-level configurations they permit. For example, we may test configurations of bearing walls to see what different floorplans each can accommodate, and we may test a floorplan to see what furniture arrangements it allows. This measure is not absolute; rather it depends on the selection and distribution rules of the elements in the lower-level configuration. Thus the same configuration of bearing walls would rate differently under different room-size and room-layout constraints.

We have discussed three kinds of dependencies in the built environment: those we declare by fiat, those due to relative positions of elements, and those inherent in the combinations of elements themselves. We have seen that position relations between elements may be ordered by a dependency, that some dependencies are due to the <u>relative placement</u> of the two elements, and that other dependencies are <u>inherent properties</u> of the two related elements. Elements with the same inherent position dependency are said to belong to the same level. The designer must understand position dependencies in order to predict how changes will propagate through the design. Another aspect of dependency in a design concerns the sequence of design decisions. For example, in placing two rooms in a floorplan, the first room placed may limit or even determine the possible positions for the second. We then may say that the position of the second element depends on the position of the first.

We now turn to the implications for the constraint explorer. We want the constraint explorer to know about physical dependence of elements and to be able to distinguish between elements of different levels. It can then be made to simulate the physical dependencies of elements in the built environment. We may want to be warned if we try to move or remove an element on which other elements depend, or even to be prevented from so doing. ("You can't move that column; the house will fall down!"). Sometimes, however, we want to move elements around without regard for physical dependencies. We want to turn the enforcement of dependencies on and off, at our own discretion.

Entering elements of different levels into the design introduces dependencies into the system. An element's level is an inherent property--like its color or shape--and therefore we must indicate the level of every new element we enter. For every element, three level descriptors may be needed, one for each of the dependency relations: support, supply, and containment.

CHAPTER 5

Floorplan Layouts

5.1 The Design of Supports.

How might the constraint explorer be used as part of a design process? To answer the question we look at an interactive session that represents a small portion of a larger space-planning design process* · However, before looking at the session let us take a brief overview of the larger design process discussed here--the "design of supports".

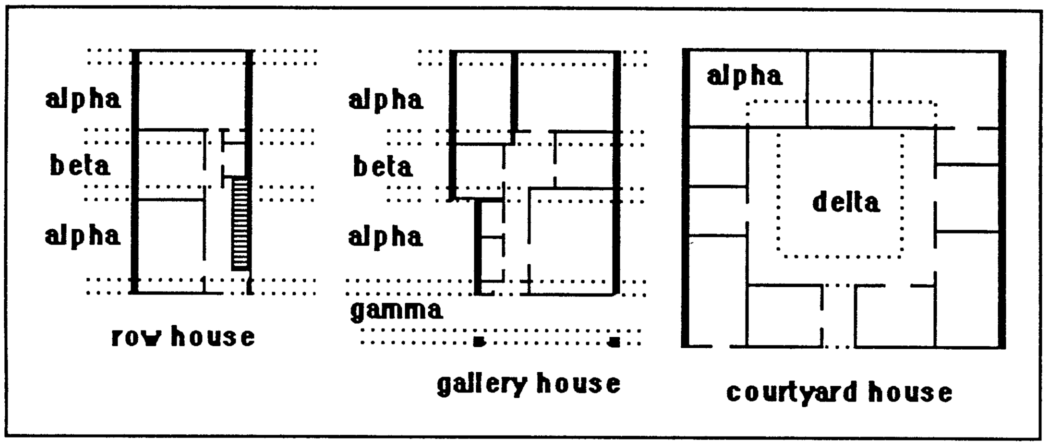
Planning for change is essential to the design of a support. A "support" consists of the parts of a building shared by individual dwellers, for example, bearing walls, floors, and public spaces. The designer's job is to decide on the placement of these shared elements in a given site. This configuration of shared elements is the support, and it then serves as a site for further design interventions carried out by the inhabitants of the building over time. For example, individuals may change interior partitions without disturbing the support. The size and arrangement of spaces indicated by support elements determines a support's capacity, its adaptability to different uses and territorial divisions. Evaluating the relative capacities of different arrangements of spaces is therefore an important operation in the design of supports.

* The design session presented here is based on an excerpt from Habraken's unpublished manuscript, Making Basic Variants with a User-Friendly Machine [Habraken 83]. I have extended some of the queries to better illustrate what the constraint explorer understands and I have also added explanatory notes.

In the interactive session we explore the capacity of a given arrangement of spaces, or site, to support a certain program of activities, or set of "functions". For example, a 1-bedroom dwelling program includes the functions: kitchen, bathroom, storage, bedroom, livingroom, and diningroom. Some functions in a program may be optional, others mandatory. We associate certain ranges of dimensions and certain positions with each function. For example, "in a walkup apartment the entrance must be at least 5' wide, 7' deep, and located in the service zone adjacent to the public stairs". Knowing the dimensions that each function requires and the proper combinations and arrangements of functions, we explore the programs and program layouts possible within a given site. In this way we come to understand the capacity of the site to support lower-level variation.

The S.A.R. method* can help in evaluating capacity but to use it we must describe the design as an arrangement of zones, margins and sectors. These formally represent the spatial organization of the design. Figure 5.1 shows zone distributions and sector groups for several familiar dwelling types: row house, gallery apartment, and courtyard house. Generally, zones run along the major direction of the building (parallel with the street in an urban block situation); sectors are clear spaces between divisions of zones, often delimited by walls running perpendicular to the zones. We can see that a zone distribution is a configuration of zones arranged one after the other, with a margin between each pair of zones.

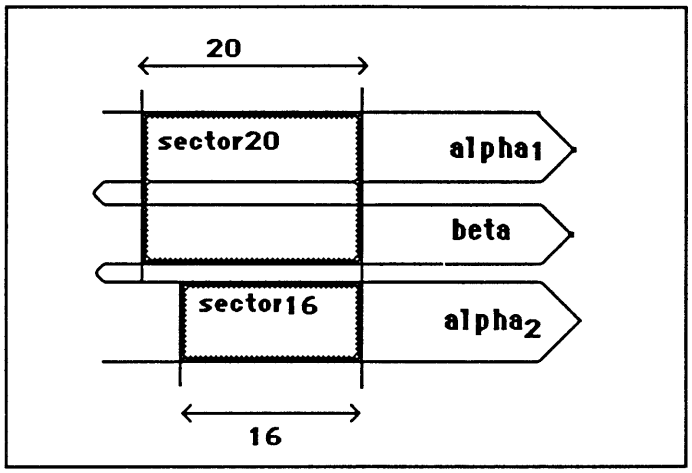
* The design methodology developed by the Stichting Architecten Research (the dutch Foundation for Architecture Research) helps designers evaluate the capacity of proposed designs. This method has been used successfully for over fifteen years by architects in the Netherlands to design housing. A brief summary of the work of the S.A.R. appears in chapter 7. Variations: a Guide to the Systematic Design of Supports by Habraken et al [Habraken 76] contains a complete exposition of the S.A.R. method.



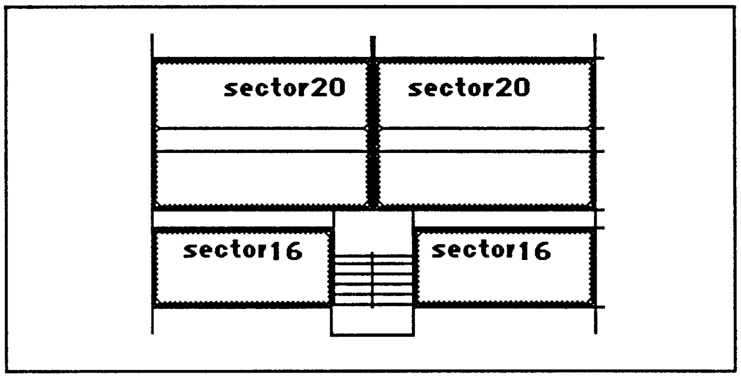
Sector groups and zone distributions for different dwelling types.
Figure 5.1

Figure 5.2 shows a simple sector group consisting of two sectors spanning three zones: Alpha1, Beta, and Alpha2*. In the following session we use the constraint explorer to explore the capacity of this sector group. In a building involving many different sectors we would repeat a similar analysis for each of the different sector groups. The sector group examined here describes the spatial structure of a simple walkup flat. The same sector group might be alternately reversed and repeated so that pairs of adjacent flats share access, as shown in figure 5.3. Zones Alpha1 and Alpha2 represent respectively the back and front zones of the building, adjacent to the exterior. Beta represents the building's interior, or service zone.

* Actually the S.A.R. method recognizes three sectors here, one for each zone. Here we have concatenated sector20 (alpha1) and sector20(beta) into a single sector spanning two zones.



Sector group for simple walkup flat.
Figure 5.2.



Adjacent dwellings share access.
Figure 5.3.

The S.A.R. method recognizes three kinds of constraints: site constraints, function norms, and position rules. Site constraints represent the built context of the design: the zones and sectors. Function norms constrain the dimensions of each function that may appear in the floorplan. Position rules describe where each function may occur in the building. In order to proceed with exploring layout alternatives we assume that these constraints have been previously defined; at the end of this chapter we see how to enter them into the constraint explorer.

5.2 Exploring arrangements of functions in the floorplan.

Exploring a sector group's capacity consists in trying different positions for each function in each potential program. At this stage we only want to identify "basic variants", those alternatives with functions in different positions, regardless of function dimensions. The constraint explorer assists by recording all transactions, issuing warnings when we violate previously entered constraints and, upon request, comparing the capacity of space remaining in the layout with space needed for functions not yet placed. As in chapter 3, the dialogue appears as a textual interaction, although a more sophisticated user-interface involving drawing and gesture can also drive the same operations. For example, we can place functions and set their dimensions using a simple pointing device such as a mouse or touch-sensitive screen or tablet. Constraint selections might be made from a menu or chart, such as those illustrated in figures 5.16 and 5.17 below. Text, though more tedious, shows more explicitly what information is exchanged between designer and constraint explorer.

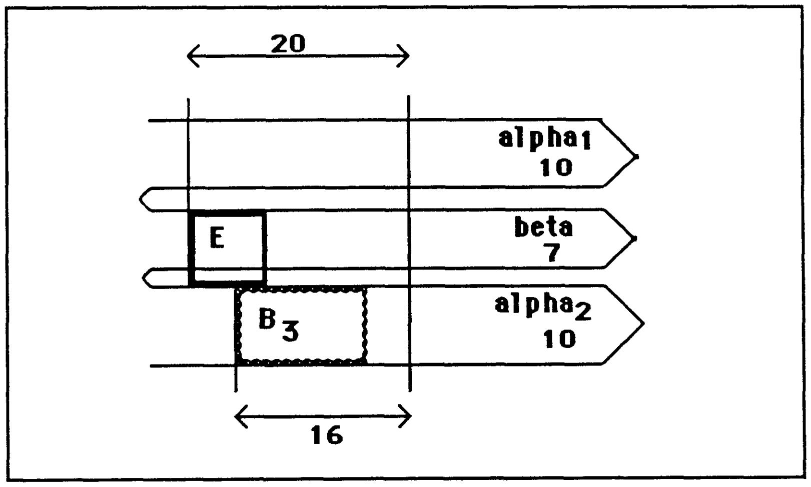
If we tried all arrangements of all functions in the site we would find many do not work due to conflicting dimension constraints or that they violate a position rule. Each arrangement of functions we specify must be tested for feasibility. That is what we do in the following session; we perform a sequence of placement and dimensioning moves resulting in two basic variants. We consider a function placed when we name the zone, or zones, it occupies and fix its left or right edge relative to a sector. As a function's top and bottom edges may vary within margins and its unfixed right or left edge may also vary, placement usually does not determine function dimensions. At this stage of design we minimize function dimensions by default (subject to position constraints) in order to maximize the number of basic variants.

We are now ready to use the constraint explorer to try out layout alternatives. Let us begin by placing two functions, E (Entry), and B3 (Master bedroom), in two moves. We must locate E adjacent to the shared access, either in the Beta zone or in the Alpha2 zone and the Beta-Alpha2 margin. Accordingly, our first move places E in Beta, aligned to the left inside Sector20. By default E's top edge registers at the top of Beta in order to minimize the use of the inner margin. Then we fix E's depth so that it occupies the entire Beta-Alpha2 margin.

```
>> place E in Beta+margin, left aligned in Sector20.
   ok.
   (state 1)

>> set (E depth) 9'.
   ok.
   (state 2)
>>
```

Next, we place B3 (Master Bedroom) in Alpha2.

```
>> place B3, in Alpha2, left aligned in Sector16.
   ok.
   (state 3)
```



Sector group after placing two functions. (State 3)
Figure 5.4.

We request an accounting for B3's dimensions. We ask first about B3's depth. We find that B3's depth, according to its function norm, may be no less than 10' and because we placed B3 in Alpha2, B3's depth may be no more than Alpha2's depth, also 10'. Therefore B3's depth must be 10'.

```
>> why (B3 depth)?
     (B3 depth) = 10; because:
      (B3 depth) ≥ 10   ; function norm.
       (B3 depth)  ≤ 10   ; placement.
            because:
               (B3 depth)  ≤ (Alpha2  depth)    ;placement.
               (Alpha2 depth) = 10.
>>
```

What about B3's width? We did not fix it; yet it is constrained. We saw that B3's depth is 10 feet and according to its function norm its minimum area is 120 square feet. The relation between width, depth, and area and prior knowledge about inequalities allows us to deduce a minimum width, 12. B3's width in Figure 5.4 shows this deduced minimum.

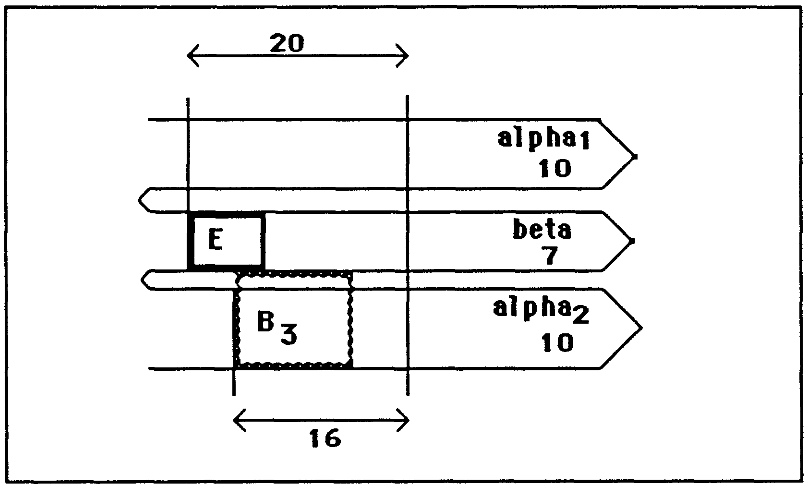
```
>>  why (B3 width)?
     (B3 width) ≥ 12          ;because:
         (B3 depth) = 10.     ; see above.
         (B3 area) ≥ 120      ; function norm.
          (B3 area) = (B3 depth) * (B3 width)  ; rectangle dimensions rule.
>>
```

Now we want to place another function in Alpha2. There may only be sufficent width for two functions in a sector if one or both of them is pushed into the margin(s). We inquire first how much free width remains in Alpha2 and then we request the "capacity" of this free width in Alpha2. The capacity operation returns the names of functions that this free width can contain and that are not forbidden by some position rule to occupy the Alpha2 zone. We see that four feet of Alpha2 is only enough space for storage. Finally we check how much width we need for a bathroom (b).

```
>> free-width Alpha2?
   4' of free-width remain in Alpha2.
   40 sq. ft. at Alpha2's present depth, 10'.

>> capacity Alpha2?
   s (storage) is the only function that fits in 4'
   of Alpha2.

>> show (b width).
   6 ≤ (b width) ≤ 9.

>>
```

We need two more feet of width to make a bathroom in Alpha2. To gain the two feet we adopt a strategy: maximize the free width in Alpha2. First we replace B3, allowing it to occupy the Beta-Alpha2 margin. Then we minimize B3's width, squeezing it up into the margin. This, however, conflicts with our previous placement of E in that same margin. We cannot have it both ways; therefore we unfix E's depth. B3 pushes E back into Beta but no further, because we have limited B3 to margin+Alpha2; that is, B3 may not occupy Beta. E's width remains unchanged.

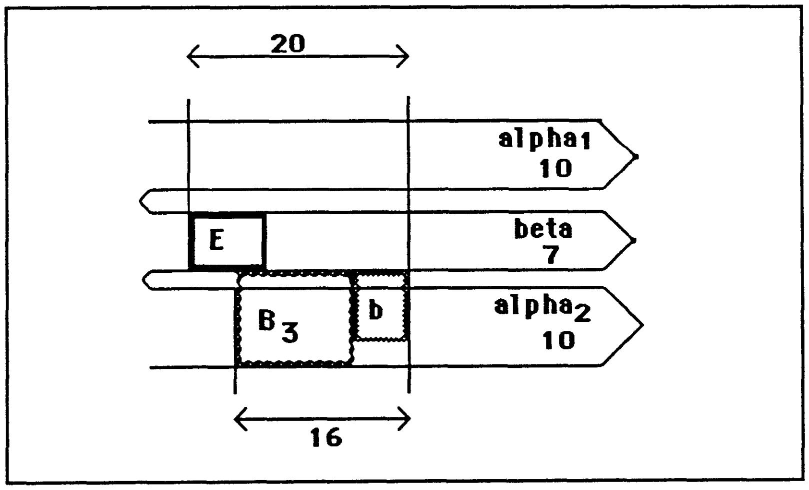
```
>> replace B3 in margin+Alpha2

>> minimize B3 width
   conflict: E, B3 overlap in Beta-Alpha2.
   (state 4)

>> unfix (E depth).
   adjusting (E depth) = 7'.
   ok.
   (state 5)
>>
```



B3 pushes E out of the margin. (State 5).
Figure 5.5

Having gained the required width in Alpha2, we can now add the bathroom. Notice that we place b in the margin although sufficient space remains in Alpha2 alone.

```
>> place b top aligned in Alpha2, Beta-Alpha2, aligned right in Sector16.
   ok.
   (state 6)

>>
```



And now there is room for a 6 x 8 bathrom in Alpha2. (State 6).
Figure 5.6

When we place more than one function in a sector or in a shared margin the constraint explorer automatically enters constraints preventing two functions from occupying the same space. Two such non-overlap constraints now exist: one between B3 and E in the Beta-Alpha2 margin, and another between B3 and b in Sector16:

```
(B3 top)  ≤ (E bottom),
(B3 right) ≤ (b left).
```

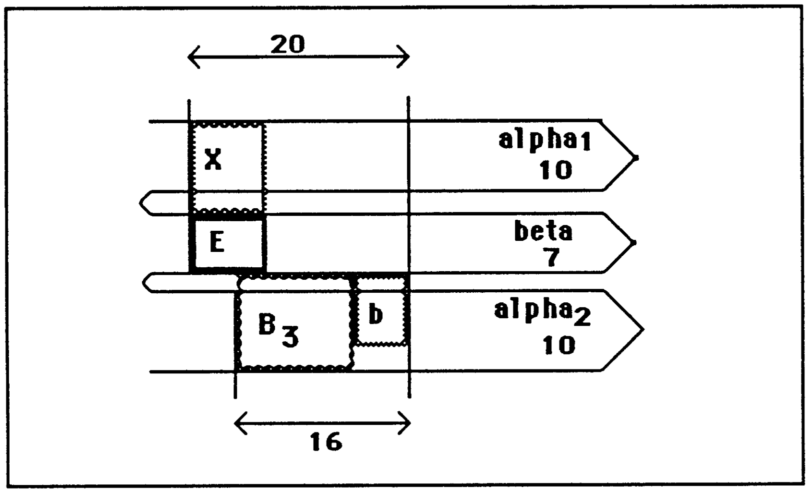
These non-overlap constraints relate the dimensions of these three functions, E, B3, and b. Pushing on b may affect E and vice versa. If we widen b, then B3 must narrow. As B3 narrows it must become deeper, in turn reducing E's depth.

We can also check the distance of shared edge between any two functions. We ensure that we can make a narrow doorway between E and B3.

```
>> shared-edge E B3?
     (E bottom) and (B3 top) share 3 feet.
>>
```

Next we place a "dummy" function, X, in zone Alpha1. X stands for a function to be identified later. It has no default dimension or position constraints. We constrain X's width and E's to be equal.

```
>> place X in Alpha1, aligned left in Sector20
>> constrain (X width) = (E width).
   ok
    (state 8)

>>
```



We place a dummy function in Alpha2. (State 8).
Figure 5.7

Now we consider what functions we might place in the Beta zone. First we ask what width remains in Beta. The answer is 13'. Planning to use 3' of Beta for access, we subtract 3' from the 13' and then request the capacity of the 10 usable feet in Beta both with and without the adjacent Alpha1-Beta margin. We do not yet place a function in Beta.

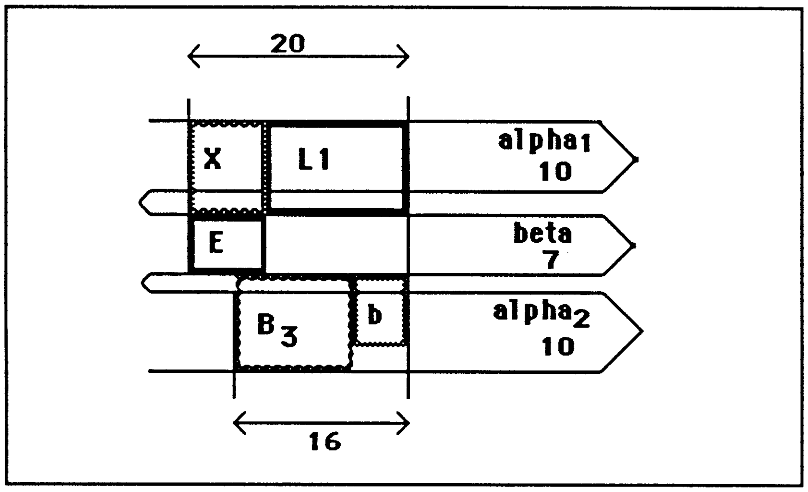
```
>> free-width Beta?
   13'.

>> capacity 10' Beta?
   B1, b, D.

>> capacity 10'  margin+Beta?
   B1, b, K1, D.

>>
```

Instead, we turn to the remaining space in Alpha1. We try to place L1 (a small livingroom) but Alpha1 in Sector20 lacks sufficient width. L1 requires 15 feet width, and X already occupies 7 of Alpha1's 20 feet.

```
>> free-width? Alpha1
   13'.

>> place L1, in Alpha1, align right in sector20.
    conflict: overlap L1 X in Alpha2.
    (state 9)

>> why?
    the following constraint is violated:
         (L1 width) + (X width) ≤ 20.
   because
        (X width) = 7.
        (L1 width) ≥ 15.
```

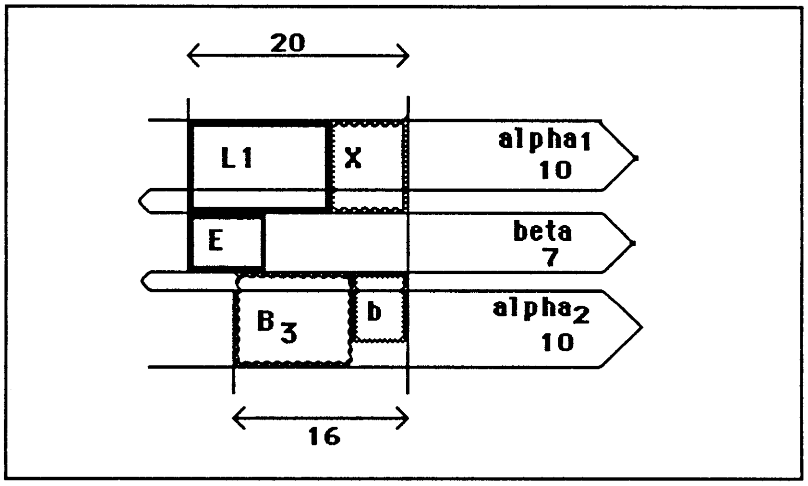
If we can reduce L1's minimum width to 13' then we can place L1 in Alpha1. We inquire why L1 requires 15 ' width and we see that as with B3, L1's minimum width is related to its depth. In placing L1, we restricted it to Alpha2. If we now allow L1 to occupy a margin we find that we can place it in Alpha1.

```
>> why (L1 width)?
     (L1 width) ≥ 15.
   because:
       (L1 depth) ≤ (Alpha1 depth)
            because (L1 in Alpha1)
                  and (Alpha1 depth) = 10;

         (L1 area) = (L1 depth) * (L1 width);
        (L1 area) ≥ 150;

    therefore: (L1 width) ≥ 15.

>> place L1, in Alpha1+margin, align right in sector20.
   ok.
    (state 10)

>>
```



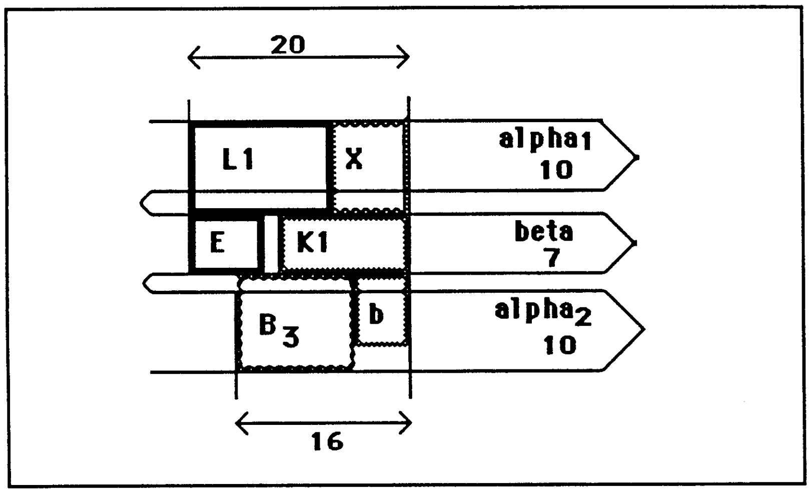
Using a margin we can place a livingroom in Alpha1. (State 10).
Figure 5.8

```
>> switch X, L1.
   ok.
   (state 11)

>>
```



We reverse the positions of L1 and X. (State 11).
Figure 5.9

```
>> place K1 in Beta, aligned right sector-20.
   ok.
   (state 12)

>>
```



The sector group is now filled with functions. (state 12)
Figure 5.10

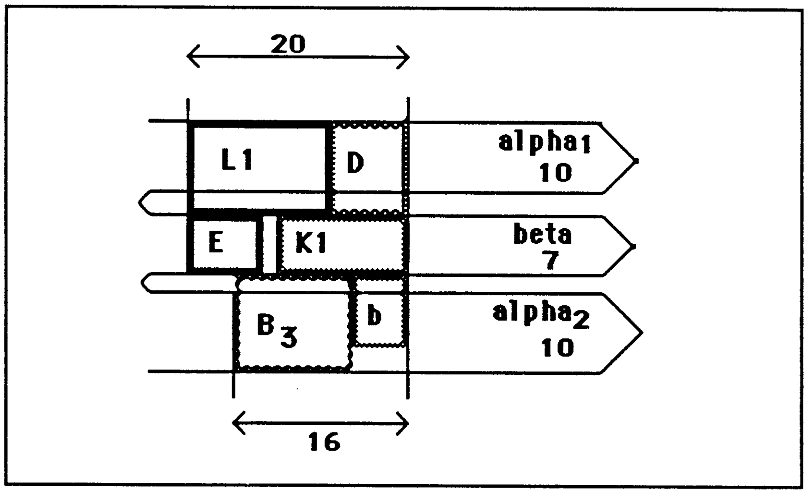
```
>> show basic-variant.
   L1  X
   E  K1
    B3 b

>> shared-edge L1 K1?
    (L1 bottom) (K1 top) share 5'.
```

We now try to make L1 narrower still; but without violating L1's minimum dimension we cannot do it.

```
>> fix (L1 width) = 11.
   conflict: L1 ≥ 13 ;(L1 function norm).
   (state 12)

>> unfix (L1 width).
   adjusting (L1 width) = 13.
   (state 11).

>>
```

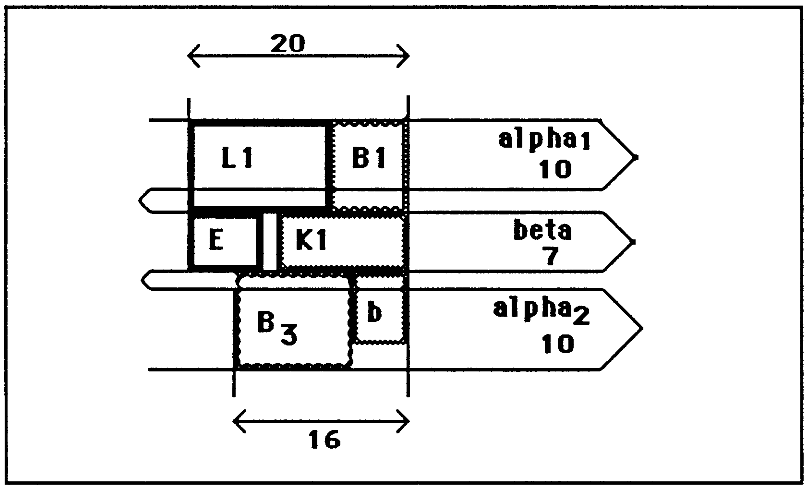
It is finally time to decide the identity of X. We want to know what functions we might substitute for X, given its location (in Alpha1) and dimension (7' x 12'). Three functions meet these constraints: dining (D), 1-person-sleeping (B1), and small-kitchen (K1). One of these (K1) we have already placed in the floorplan. We next successively substitute each of the other two functions (D and B1) for X, and save the resulting configurations as basic variants. Adding B1 changes the program from a roomy 1-bedroom to a cramped 2-bedroom flat.

```
>> capacity X?

   X may be D  (dining room),
                     B1 (1-person bedroom),
                  or K1 (small kitchen).

>>  substitute X = D.
    ok.
           (state 13)

>>
```



Substituting D for X yields a basic variant (state 13)
Figure 5.11

```
>> add to basic-variants.
  L1   D
   E  K1
    B3  b.

>> in state 12, substitute X = B1.
   ok.
    (state 14)
```



And substituting B1 for X yields another basic variant (state 14)
Figure 5.12

```
>> add to  basic-variants.
   L1  B1
   E   K1
     B3  b.
>>
```

Here we shall leave the session. We have seen how we place, move, and reshape functions in a simple site, according to dimension and position constraints, assessing the site's capacity to support a variety of layouts and programs of functions. The exploration we have been doing assumes that some constraints were already entered into the constraint explorer. In particular we have been assuming a predefined site, predefined function norms, and predefined position rules. We now see briefly how to enter these constraints.

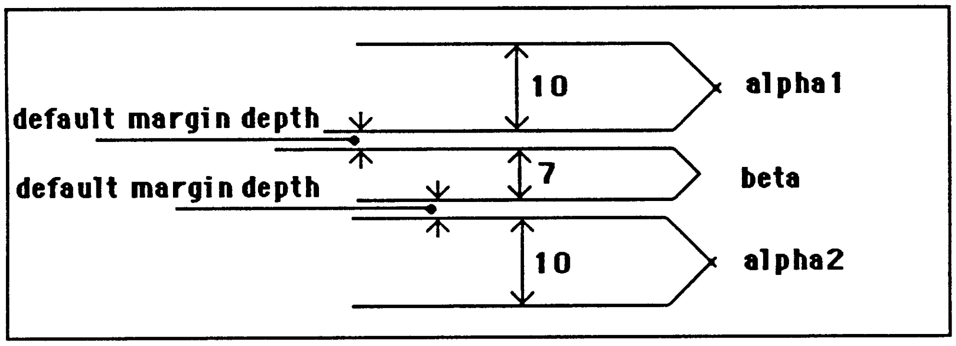
5.3 Describing the Site.

As noted earlier, the site consists of a zone distribution and a sector group. To define a zone distribution we must have zones. We first define three zones, Alpha1, Beta, and Alpha2, assigning a depth dimension to each. Alpha1 and Alpha2 are instances of the predefined prototype Alpha-Zone, similarly Beta1 is an instance of Beta-Zone. If we did not specify zone dimensions then the constraint explorer would use the default values from these prototypes.

```
>> define: Alpha1
    oneof Alpha-Zone
    depth 10.

>> define: Beta
    oneof Beta-Zone.
    depth  7.

>> define Alpha2
    oneof Alpha-Zone.
    depth 10.
```

We classify zones by distinguishing built and open zones, and exterior and interior ones. Alpha and Beta zones are both built zones. An Alpha zone represents the built space immediately inside the building edge, typically containing livingrooms and bedrooms; a Beta zone represents built space deeper inside the building, typically a place for storage and service functions. Alpha zones are exterior built zones; Beta zones are interior built zones. Gamma and Delta zones, not used in this example, represent exterior and interior open zones, locations for porches and stoops, patios and courtyards. We combine our three new zones into a zone distribution, a sequence of zones and margins. The zone distribution describes dimensional stops along the floorplan depth. An optional margin separates each pair of zones. Here we do not specify dimensions; therefore all margins assume default margin depths (2').

```
>> define Z1
oneof zone-distribution
zone-list (Alpha1 margin Beta margin Alpha2).

>>
```

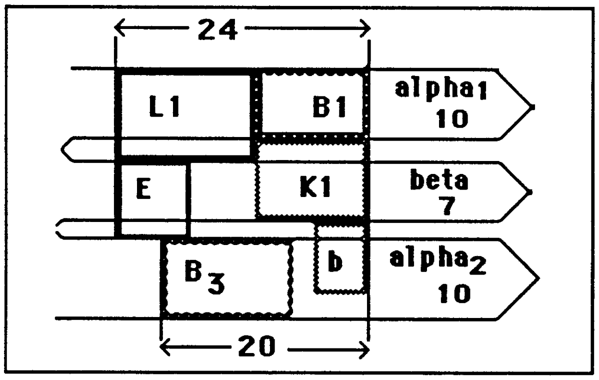


Building a zone distribution from zones and margins.
Figure 5.13

Next we define two sectors and a sector group. For each sector, we describe a width and specify the zone or zones the sector occupies. Then we define sector group SG1, a configuration of these two contiguous sectors. Note that we associate a specific zone-distribution with the sector group, and also the relative positions of the component sectors.

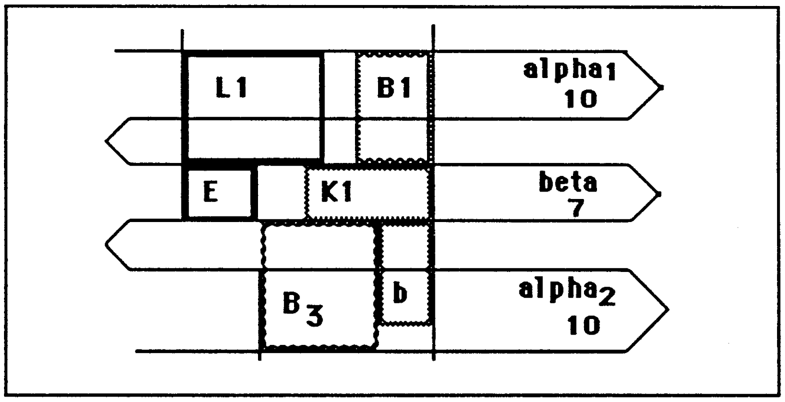
```
>> define sector20
          oneof sector
          zones (alpha1 beta)
          width 20

>> define sector16
          oneof sector
          zones (alpha2)
          width 16

>> define SG1
          oneof sector-group
          zone-distribution Z1
          sectors (sector20 sector16)
          constrain (align
                         (sector20 right) (sector16 right))
```

Each sector's width is fixed; its depth is determined by the zones and margins it spans. Zones are the highest-level elements; functions are the lowest. Functions placed in a sector move when the sector is reshaped. This may allow for some reshaping of functions, as shown in figure 5.14. Of course, stretching the basic variant in this way may result in function size violations.



Changing a sector width in a basic variant.
Figure 5.14

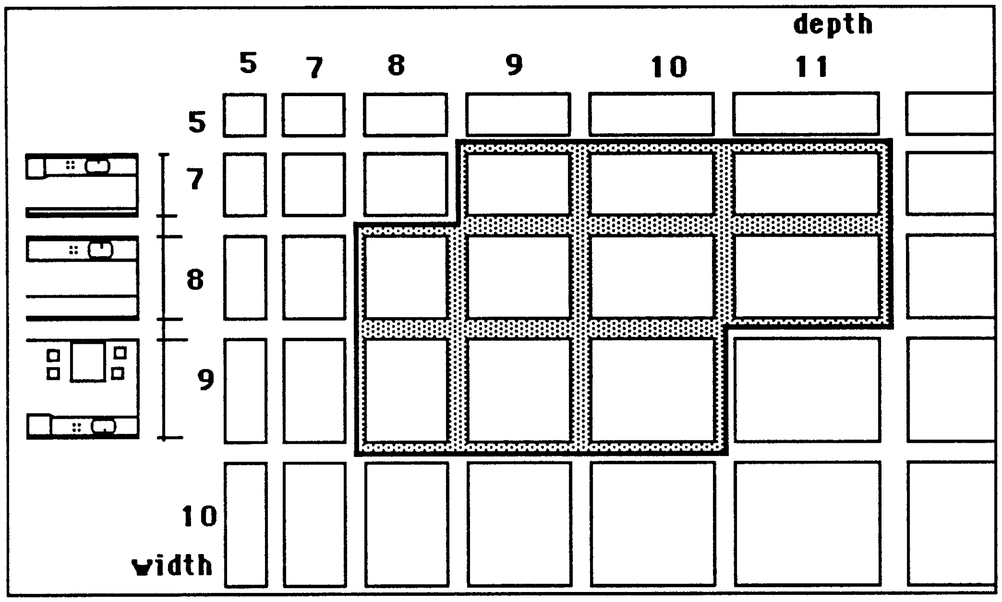
We just saw that functions, because they are placed relative to sectors, change with the sectors. Sectors are themselves placed relative to zone distributions, so changing the zone distribution causes the sectors to change; this in turn affects the functions. Figure 5.15 shows the same basic variant with greatly increased margin dimensions.



Changing the margin dimensions in a basic variant.
Figure 5.15

5.4 Function Norms.

A "function norm" is the set of allowable dimensions for a room, or function. We may specify the function norm by selecting maximum and minimum values from a chart gridded by furniture dimension increments (figure 5.16). The constraint-explorer's parser interprets selections from the chart into the constraints shown below.



Function Norm Chart specifies dimension constraints per function.
Figure 5.16

```
>> define K-function
   oneof function
     (depth ≥ 8)
     (depth ≤11)
     (width ≥ 7)
     (width ≤ 9)
     (area ≥ 60)
     (area ≤ 95)
>>
```

Minimum dimension values ensure that sufficient space is allowed for required furniture and use-space, and maximum values ensure that the space allotted for the function is not unreasonably large. We may additionally constrain the function dimensions, allowing, for example, only certain modular dimensions. Or, we may describe the function norm some other way, for example, by listing all allowed combinations of dimensions. Usually, however, minimum and maximum depth, width, and area constraints suffice to describe the range of desired function sizes. When we define a new function norm, the constraint explorer enters the dimension constraints into a new and empty package. This package serves as a prototype for instances of the function; thus function instances inherit the function norm constraints by default.

We define dimension constraints for each function in the architectural program. We then save a set of function norms together as a package. The next time we want to work with those dimension constraints we simply recall the function-norm-set package.

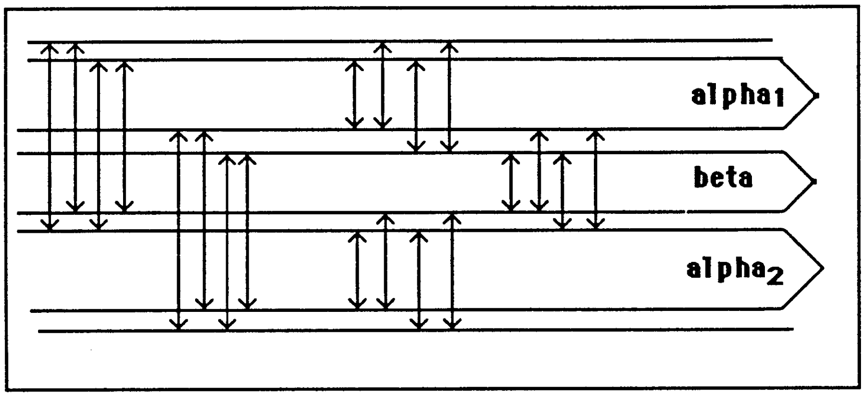
5.5 Position Rules.

In the S.A.R. method, position rules specify the allowed positions of functions in relation to a zone-distribution; for example, "Entrance must be in the Beta zone". We can also state position rules by listing the functions that are allowed/forbidden in each zone; for example: "In Alpha1, no Entrance, bathroom, or Bedroom".

```
>> position-rule: (E in Beta).
   ok.

>> position-rule: (not (in Alpha1 (E B3 b))).
  ok.
```

Instead of typing a position constraint expression for each function, we can set position constraints using a mouse or pointer and the chart of possible position constraints shown in Figure 5.17. From this chart we can select position constraints for each function, or combine simple position constraints using NOT, OR, and AND. For example, we can construct the following position rules:

```
>> position-rule: (L1 in Alpha1) or (L1 in ALpha2).
  ok.

>> position-rule: (or (K1 in Beta) and (L1 in Alpha2)
                      (K1 in Alpha2) and (L1 in Alpha1))
  ok.
>>
```



Position rules chart.
Figure 5.17.

CHAPTER 6

The Parts of the Constraint Explorer

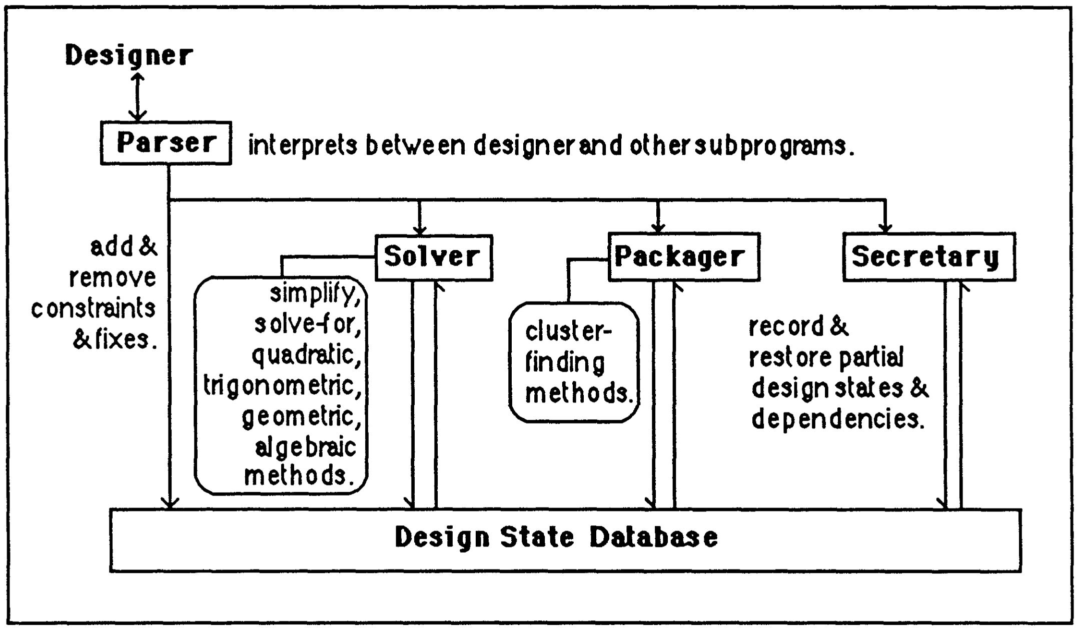
6.1 The overall organization.

We now examine some details of how to implement the constraint explorer. First we shall look at the overall organization of the program, then at the data-structures used to represent constraints, variables, and packages, and finally we see the role of the various parts of the constraint explorer program in a brief example. First let us take an overview of the constraint explorer system. The constraint explorer is to be embedded in an object-oriented dialect of lisp, ObjectLisp. Embedding permits the designer using the constraint explorer to also access and benefit from directly the full functionality of the implementation language*. For example, the inheritance hierarchy of prototypes and instances discussed in chapter 4 is an integral component of the ObjectLisp programming environment. Lisp embedding also makes it easy to interface the constraint explorer with other programs written in Lisp.

The constraint explorer is a collection of programs organized around a few elementary data structures: constraints, variables, and packages. Constraints and variables are the most atomic data structures the constraint explorer knows, and packages are collections of constraints and variables. The programs: the parser, packager, solver, and secretary of state perform the constraint explorer's functions. The programs share a common database that

* The project is presently being developed in Objectlisp, a lexically scoped lisp environment with an extension providing generically scoped closures[Drescher (forthcoming)]. Objectlisp provides a syntactically elegant, semantically powerful implementation of object-oriented programming. For some advantages of this style of programming, see also [Abelson and Sussman 85; Robson and Goldberg 83]. A Scheme implementation is also being considered.

represents the evolving state of the design. Figure 6.1 shows a block diagram of these programs and their shared database, the design state.


Designer
Parser
interprets between designer and other subprograms.
add & remove constraints & fixes.
Solver
simplify, solve-for, quadratic, trigonometric, geometric, algebraic methods.
Packager
cluster-finding methods.
Secretary
record & restore partial design states & dependencies.
Design State Database

Block diagram of the constraint explorer.
Figure 6.1

The design state database is simply the set of all constraints and variables, and their various packagings. The parser interprets the designer's commands and queries, distributing commands to other parts of the constraint explorer and adding and removing constraints and value settings to the design state database. The solver propagates values through the system of constraints, solves symbolically and arithmetically for variable values, and performs local optimizations upon request. The packager identifies clusters of closely interacting constraints in the design state database and groups them together for solving. The secretary of state maintains a history of the design process-- it saves and restores portions and sequences of the design state and records dependencies between design decisions. Notice that no supervisor program controls the execution of the constraint explorer's subprograms. Rather, the constraint explorer is event-driven; each subprogram monitors the design state database and activates itself as needed when new constraints and settings are entered.

6.2 The data-structures.

The main data structure is a set of connected constraint and variable data structures. This set represents the state of the design-in-progress. In chapter 3 we looked at fragments of design states. There we used a network to represent these fragments. We saw how constraint and variable data structures can be packaged together and connected to model complex physical systems. Here we use a matrix to discuss operations on the design state as a whole. Any data representation offers certain advantages and disadvantages. For many solver operations the matrix view seems appropriate. For example, solving a set of simultaneous constraints is most easily accomplished using matrices. The network representation has storage-management advantages, however, for large and sparsely interacting sets of constraints and variables. It may therefore be advantageous to store the design state as a large network, building matrix representations for small pieces as needed for the solver. The matrices could then replace pieces of the network. At any rate, the two representations are fundamentally equivalent, and we need not be concerned here with whether the database is implemented as a network, as a matrix, or as a mixture of both.

Think of the design state as a set of constraints and variables:

$$
\begin{array}{l}
C_1 (x_1, x_2, x_3, \ldots) \\
C_2 (x_1, x_2, x_3, \ldots) \\
C_3 (x_1, x_2, x_3, \ldots) \\
\ldots
\end{array}
$$

That is, each constraint C_i can be written as a functional relation of variables $x_1 \ldots x_j$. Let us suppose further that each C_i can be written as an inquality relation:

$$f_i\ (x_1, x_2, x_3, \ldots) \geq 0^*.$$

(We shall call this the "normal form" of the constraint.) Then each C_i can be reformulated in various ways. That is, by solving the functional relation f_i for each x_j we can generate functions $f_{i1}, f_{i2}, f_{i3}, \ldots$ such that,

$$x_1 \geq f_{i1}\ (x_2, x_3, \ldots)$$
$$x_2 \geq f_{i2}\ (x_1, x_3, \ldots)$$
$$x_3 \geq f_{i3}\ (x_1, x_2, x_4, \ldots)$$

Each of these expressions we call a different formulation of the constraint, and each of the functions f_{ij} we call a formula (following spreadsheet terminology) for the constraint C_i with respect to variable x_j. We can write these formulae in a matrix describing the design state (see figure 6.2). We arrange the variables in columns and the constraints in rows. In each cell corresponding to the interaction of a constraint i and variable j we write the formula f_{ij}, that is, constraint C_i solved for the variable x_j. For example, $a < b$ and $b > a$ are two formulations of the same constraint. Looking across each row we find the same relation f_i between variables expressed in each cell, but with the value of x_j expressed in terms of all the other variables. Looking down each column, we find a list of the different constraints on the variable represented by that column.

	x_1	x_2	x_3	x_4
c_1	f_{11}	f_{12}	f_{13}	f_{14}
c_2	f_{21}	f_{22}	f_{23}	f_{24}
c_3	f_{31}	f_{32}	f_{33}	f_{34}

Form of the design state matrix
Figure 6.2

* Much can be expressed with even only this limited sort of constraint. For example, using 1 and 0 for the boolean values true and false we can write the full complement of first order logic relations (AND, OR, NOT, IMPLIES, etc.). We can also aggregate primitive constraints to describe compound relations. However, we need not limit constraints to *algebraic* inequalities and equations. We may well be interested in other sorts of relations, for example, the relation that the selection of elements in one configuration be a subset of the selection of elements in another, or topological relations. We would have to extend the solver's knowledge of relations, but this requires no major overhaul.

The matrix for an entire design will likely be enormous, having many constraints and variables, but we almost always work with submatrices, or packages of constraints and variables. There are two types of atomic data structures in the database: the constraint and the variable. We shall look at an example of each. These can be thought of as header blocks for the rows and columns of the matrix, respectively, or as nodes and links in a constraint network. In addition to the information displayed in the matrix and network diagrams, these data structures store additional information about each constraint and variable used by the packager and the secretary of state. For example, each constraint and variable data structure knows what packages it is belongs to, as well as what other items in the design state it depends upon. Here is how a constraint prints out:

```
CONSTRAINT
    name: floor-ceiling-height-relation.
    expression: top - (bottom + height) = 0.
    formulae: [ top    = bottom + height;
                height = top - bottom;
                bottom = top - height ]
    variables: top; bottom; height.
    status: indeterminate.
    basis: declared in package "common dimensional relations".
    history: [ ]
    packages: [common dimensional relations; P13].
```

Think of a constraint data structure as a frame with slots* for: the expression of the constraint (its normal form), the list of different formulations of the constraint (entries in this list correspond to the cells across one row of the matrix), the constraint's present status-- (satisfied, indeterminate, or inconsistent), a list of the constrained variables, and a history list that stores the modifications to the expression throughout the design process. (The values of slots in the frame may be precomputed, or they may be computed on demand. For efficiency, once computed, slot values may be cached.) The expression and formulae of a constraint

* Or more simply, a table of properties.

change throughout the design process; as the designer adds constraints and fixes variable values, the solver can begin to simplify constraint expressions. The source of each constraint is indicated by its basis slot. The basis slot records a set of variables and constraints that were used to derive the present constraint. This is useful when the designer wishes to track the chain of deductions and dependencies in a design state. When an earlier decision is retracted, the secretary of state can use this dependency information to determine what decisions and deductions made after the retracted move may be affected. For newly introduced constraints, the basis is simply "declared". Deduced constraints and values may remain even after their basis is retracted. By default, upon retraction of all its bases, the basis slot of a constraint reverts to "unsupported". Or, if we prefer, we can instruct the secretary of state to erase unsupported values automatically. Information in the history slot is used to restore previous states of the constraint. The package slot lists the packages the current constraint belongs to, both user-defined (a-priori) packages, and thosed inferred by the packager (a-posteriori). In this case, the constraint belongs to one user-defined package, named **common dimensional relations**, and one package, **P13**, defined by the packager, that groups together a number of constraints that share several variables. The name **P13** is invented by the packager; the designer may give the inferred package a mnemonic name.

Each variable data structure stores a name, value, a list of constraints that refer to it, a history, or list of previous values, and a list of packages that it belongs to.

```
VARIABLE
name: height
value:  [ = top - bottom; = (2/3) length]
constraints:  floor-ceiling-height-relation
history: [].
packages: [common dimensional relations, P13]
```

Notice that the value of height is a list of two formulae, not a single number. As the designing proceeds, this value becomes increasingly specific. In traditional programming environments, a variable is either bound or unbound. In the constraint explorer, a variable may be partially specified. In the constraint explorer a variable's value is the intersection of all the constraints on its value. Thus we can arrange values in order of constrainedness, or specificity. Proceeding from least to most constrained, a variable value may be: (a) named (declared) but not constrained, (b) constrained, (c) constrained to an interval value, (d) constrained to a fixed number value, (e) overconstrained. Partial specification of variables is a useful device for representing ambiguity in design.

We have seen the two atomic data structures used to represent constraints and variables and how they can be arranged in a matrix. We glanced briefly at some of the bookeeping information that is carried in each data structure. We discussed the range of specificity of variable values in the constraint explorer and we saw that variable values are equivalent to the intersection of all the constraints on that variable, fixed values being a special case. Now we turn to a small example, showing how the parts of the program, especially the solver, operate on the design matrix.

6.3 Example.

Let us consider an example. Suppose we work with the relation between the position of the floor (bottom), the position of the ceiling (top), and the height of a space (height). On a drawing we might indicate the two positions and the distance between them. The parser would enter the following constraint:

$$C_1: f_1 \text{ (top, bottom, height)} = \text{top} - \text{(bottom + height)} = 0.$$

The parser builds a new data structure for this constraint, adding a row to the design matrix. If any of the variables have not been previously mentioned, then the parser also builds data structures for them, adding columns to the matrix. To fill in the cells of the new constraint row, the parser also calls on the solver to calculate the three formulae for the three variables top, bottom, and height. These are:

$$\begin{aligned} \text{top} &= f_{11} \text{ (bottom, height)} = \text{height + bottom.} \\ \text{bottom} &= f_{12} \text{ (top, height)} = \text{top - height.} \\ \text{height} &= f_{13} \text{ (top, bottom)} = \text{top - bottom.} \end{aligned}$$

Next we introduce a new variable length, and describe a desired relationship between height and length.

$$C_2: f_2 \text{ (top, bottom, height, length)} = \text{height} - (2/3)\text{length} = 0.$$

Notice that C_2 does not actually involve variables top and bottom. However, we indicate them in the notation for f_{23} and f_{24} to remind ourselves that each constraint may relate as many as all the variables in the design. Solving for length and height, then, we have:

$$\begin{aligned} \text{height} &= f_{23} \text{ (length)} = (2/3) \text{ length.} \\ \text{length} &= f_{24} \text{ (height)} = (3/2) \text{ height.} \end{aligned}$$

After the parser enters these two formulae in the matrix, we have the state shown in figure 6.3.

	top	bottom	height	length
0 = top - (bottom + height)	= bottom + height	= top - height	= top - bottom	
0 = height - (2/3) length			= (2/3) length	= (3/2) height

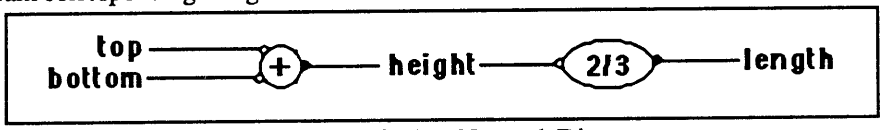
Matrix after entering two constraints.
figure 6.3*

Notice that the matrix of constraints and variables is sparse; that is, many constraints and variables simply do not intersect. Only five of the eight cells in the matrix of figure 6.3 are used; the other three remain blank.

Now that we have seen how the parser arranges constraints in the design matrix as we enter them, we can begin to see how the solver and the other parts of the program support exploration. The solver operates on the matrix in several ways. The simplest is the propagation of changes through the design. In chapter 3 we briefly discussed propagation of changes through constraint networks. Here we see how propagation works from the matrix point of view. To propagate the effects of fixes through the design, the solver operates on the design matrix in much the same way as a spreadsheet engine operates on a spreadsheet [Kay 84]. As we shall see, the constraint explorer's solver operates in other ways as well.

In the matrix view, propagation works as follows. We begin with a matrix as in figure 6.3. Upon fixing a variable value, we write the new value into all of the non-blank cells of the variable's column. For example, if we fix the value of height (in figure 6.3), we would write

* It may help to momentarily recall the network notation of the same set of constraints. Figure 6.3a shows the network diagram corresponding to figure 6.3.



Equivalent Network Diagram.
Figure 6.3a

the new value into two cells, whereas if we fix the value of <u>top</u>, we only would write it into one cell. (Instead of erasing the previous contents, think of each cell as a stack onto which the new fix is pushed. If later we wish to withdraw a prior fix, the cell's previous contents are available.) The solver is automatically invoked after each fix to calculate any consequences. Now suppose we set the <u>length</u> variable to 15. The solver then computes a value for <u>height</u>, using C_2, and writes the computed value, 10, in both cells of the <u>height</u> column. It cannot compute further until we give a value for either <u>top</u> or <u>bottom</u>. Figure 6.4 shows the design state matrix.

	top	bottom	height	15 ↓ length
0 = top - (bottom + height)	=bottom + 10	= top - 10	= 10	
0 = height - (2/3) length			= 10	= 15

Matrix after injecting one fix.
figure 6.4

Notice that the solver has simplified the expressions in the cells for <u>top</u> and <u>bottom</u>, replacing the variable <u>height</u> by its present value, 10. Immediately after fixing another variable (we set <u>top</u> to 14 feet), the solver is automatically invoked again. It computes a numeric value for <u>bottom</u> to reach the state shown in figure 6.5.

$$\begin{aligned} \text{bottom} &= \text{top - height} \\ &= 14 - 10 \\ &= 4, \end{aligned}$$

	14 ↓ top	bottom	height	15 ↓ length
0 = top - (bottom + height)	= 14	= 4	= 10	
0 = height - (2/3) length			= 10	= 15

After injecting a second fix.
Figure 6.5

The solver propagates the consequences of new fixes by copying values down the columns, and by computing functional relations along the rows. In this example, length's fixed value determined height. When subsequently we fixed top's value also, then bottom's value was determined consequentially.

Though propagation works well in the above example, it is a weak method; inherently local it cannot make inferences that require a global understanding of the design state. It requires variable values to be given; from these it computes the values of other (dependent) variables. Essentially it substitutes constants for variables and simplifies arithmetically. It only yields solutions in some of the situations where solutions are possible. For example, when constraints are simultaneous, the solver can compute a solution without any given values. That is, if we describe two constraints

$$\begin{aligned} a + b &= c, \text{ and} \\ b &= c, \end{aligned}$$

then from the constraints alone, the solver can deduce that a's value must be zero. This inference cannot be reached using only propagation of values, however. A simultaneous solution algorithm must be employed.

Solving sets of simultaneous constraints may be easier than identifying them in the first place. Most likely simultaneous constraints come from different sources, were entered at different times by different designers, and are nowhere declared explicitly as a set of simultaneous constraints. It is the job of the packager to notice that constraints 10, 14, and 177 (say) are simultaneous, and to make a new package labelled "system of simultaneous constraints to be solved" and pass it to the solver. It is the solver's job to solve the system of constraints, and fill in any deduced variable values.

Though some relations, such as the relation between the side of a square and its area, are equations, in design we work most often with inequality relations. Inequalities enable us to retain and modulate ambiguity in a design. The constraint explorer can be easily extended to handle inequality constraints as follows. Values, such as 4, 10, and 14, are permitted to take on interval number, and half-line number values. We shall discuss these ideas informally; a rigorous treatment of interval numbers is provided elsewhere [Moore 65]. The interval [8 11], for example, represents the set of values between 8 and 11[*]. If we set the value of height to [8 11], the equivalent constraint is $8 \leq \text{height} \leq 11$[†]. A half-line value is a degenerate case of an interval number, where only one of the interval's bounds are given. For example, the half-line number [∞ 5] represents the set of number values less than 5; think of the ∞ as representing either positive or negative infinity, depending on which position it occupies in the interval number notation. The solver understands the operations +, -, *, and / for interval values as well as for simple number values. For example, the difference of two interval numbers is defined as

$$[a\ b] - [c\ d] = [\ (a-d)\ (b-c)\ ].$$

With this small extension of the solver's arithmetic and algebraic abilities, we can use the same symbolic mathematics machinery to manipulate inequality constraints that we use for equations. For example, suppose we have the constraint,

$$x + y \leq 16.$$

The solver can translate this inequality to an equation of interval values:

[*] An interval can be open or closed on either end, meaning that the end value is included or excluded in the interval, respectively.

[†] We began by distinguishing two different kinds of things: constraints and variables. Now we find that, though useful, the distinction between a constraint and a variable is not absolute. An interval value is equivalent to a pair of inequality constraints and fixing the value of x a constant k is equivalent to writing the constraint $x = k$: a fix is simply an extreme form of constraint.

$x + y = [\infty\ 16]$.

If we set x's value to 4,and solve for y, we have:

$y = [\infty\ 16] - [4\ 4] = [\infty\ 12]$.

Converting back to inequality notation,

$y \leq 12$.

We have seen how, by extending the solver's operations to work on continuous intervals of values, we can manipulate inequality relations as though they are equations. In a similar way, we can extend the solver's operations to work on sets of discontinuous values. We call these sets "choice values". That is, a variable might stand for the choice {2 4 8}, meaning that the variable value is constrained to be one of those numbers. The sum of the two choice values is the set of sums of pairs of numbers picked one from each set. Thus:

$\{2\ 5\} + \{6\ 8\ 12\} =\{8\ 10\ 11\ 13\ 14\ 17\}$.

We can also define choice values by setting constraints. For example, we can define n as the set of odd numbers:

remainder $(n/2) = 0$.

$n = \{...-3\ -1\ 1\ 3\ ...\}$

We have also seen that the solver requires a variety of methods to simplify and solve sets of constraints. The methods for solution depend on what sorts of constraints we want to write. At a minimum, the solver must be able to do linear algebra and linear programming, as well as numeric and algebraic solution of equations and inequalities. We are likely to need also quadratic, trigonometric, and other nonlinear constraints, as well as geometric constraints. We see now that a full-fledged version of the solver must be capable of a range of symbolic mathematical techniques.

6.4 A Closer Look at the Parser and Solver.

How are constraints, values, and objectives entered? How is the design state database browsed and edited? In general, how does the constraint explorer program appear to the designer? The parser interprets the stream(s) of input from the designer. It is the part of the constraint explorer that deals most directly with the designer. We can divide the parsing task into two components: managing input from the designer (reading), and building and modifying the internal data structures accordingly (constructing). Thus we can divide the parser into two parts: a reader that deals with the outside world of the designer, and a constructor that deals with the internal world of the constraint explorer's data structures. The designer enters constraints and values, selecting them from a chart or menu, indicating them gesturally with a mouse or tablet, or typing them in at the keyboard. The reader handles this input. The constructor instantiates data-structures for each new constraint, variable, and package and connects the new data-structures into the rest of the design.

Ideally we would like to instruct and program the constraint explorer using a full complement of interactive media: sketching, drawing, verbal description, gestures. The problem of parsing is difficult, and a good parser must understand a set of representational conventions specific to the design domain and perhaps even idiosyncratic to the designer. Graphics input technology is just now arriving. Early attempts at sketch recognition and graphical inference met with limited success [Negroponte 70]. The parsing task is also undoubtedly informed by contextual knowledge (external to the drawing itself) of what the drawing is about. For example, we are told whether a drawing is a plan or a vertical section, and from this we infer the meanings of graphical symbols; the same size rectangle in plan and section may indicate a bed and a door respectively. Moreover, drawing is not everything;

designers resort to natural language to discuss the elements, relations, objectives, and difficulties of a design. Constraints such as "align", "parallel", "straight, "connected", "inside" may be indicated with a graphic, gestural, or even textual symbol. Reading drawing is largely context dependent; for example, identical rectangles in a plan drawing and in a section drawing may represent entirely different elements.

Designing a user-interface for constraint -based programming is challenging, and little work has been done in this field. We have simply assumed a lisp-like syntax for all interaction; user-interface has not been of concern in this study. Lisp's uniform syntax obviates the need for a separet parsing program. Rather, we discussed using constraints as a representation for architectural knowledge.

How have other constraint-based programs dealt with the parsing problem? A spreadsheet appears to the user as a matrix with text in many cells. TK!-Solver* by Software Arts, the only presently available commercial product approaching a constraint language is also entirely text-oriented--users must type in algebraic expressions and value settings. By contrast, the Sketchpad program enabled the user to construct diagrams of constraint networks-- in effect, to program--using a light pen and a few buttons [Sutherland 63]. In a very sophisticated design environment, the designer could presumably just draw and sketch, making marginal notes in some textual or graphic shorthand. The user-interface of Borning's Thinglab program combined network editing and Smalltalk style browsing [Borning 77]. Gesture-based programming environments have been shown capable of sustaining a small programming language [Minsky 84], and gestural interactions may work well with graphical information. A graphics editor with a strict syntax structures user-input and simplifies the parsing task . Thus

*Originally developed by Seth Steinberg and Milos Konopasek to assist students of textile design calculate the relationships between various weaving parameters.

the user must say "here is a window", "here is a position", explicitly labelling every graphic move. Levitt is also thinking about user-interface issues with respect to a constraint-oriented musical design program [Levitt 85].

The solver is a collection of symbolic and numeric mathematics routines for computing with constraints, relations, and objectives. The solver detects inconsistencies and eliminates redundancies among constraints, and when possible it simpifies and solves for variable values. A simple algebraic solver can be implemented using rule-based technology [Gosling 83]. See also FAMOUS, the Macsyma system, QAS, and, the more recent SMP language [Fenichel 64; Macsyma 82; Konopasek & Papaconstadopoulos 78; Wolfram 84].

In most other constraint-based computing environments relaxation is often used to find solutions to constraint sets that cannot be solved by propagation (see chapter 7). Although often advertised as a satisfaction technique, relaxation seeks local optima in an objective function composed from all the constraints. Relaxation, being a strictly numeric technique, cannot tell how many solutions a given set of constraints may have; relaxation finds always a single solution. For example, given the constraint

$$x^2 = 4,$$

and working from an initial value for x, relaxation will adjust x towards the nearest solution (either +2 or -2 in this case, depending on the initial value) in a series of successive approximations. It cannot find the other solution unless given another initial value on the other side of zero. Therefore the solver uses symbolic methods whenever possible to find the solution(s) to a set of constraints. When they are a finite number, it is often as important to know how many solutions exist as it is to know the solutions themselves. When we set the constraint $x^2 = a$, the solver knows that x has two solutions, even before fixing a's value (if a is zero then both solutions are the same). We may say x has two degrees of freedom: its sign and its numeric value. The sign of x may take any of the three values {+, -, and 0}; the numeric field may take any independent rational number value, except that if the value of one field is zero, the other must also be zero. Thus we can define the constraint "$x^2 = a$" using more primitive constraints:

$x = (\text{Sign})(\text{PositiveRoot})$
$\text{Sign} = \{-1\ +1\}$
$\text{PositiveRoot} = \sqrt{a}$
and $a = (x)\,(x)$.

The solver will also optimize on demand. It operates on small sets of constraints as grouped by the packager. When optimizing locally, the designer must specify what package of constraints to optimize over, the objective to be optimized, and whether a maximum or minimum is sought. The simplex method will optimize the objective function when working with linear constraints; relaxation and/or annealing methods may be used to optimize over regions bounded by nonlinear constraints, or linear approximations to the constraints may be obtained.

The secretary of state maintains a history of events in the design. It can restore previous states or partial states upon request. Dependency directed backtracking techniques [Stallman and Sussman 77] to support efficient retraction of previous fixes. This enables the secretary to minimize side-effects of retractions. Recall the example in the previous section. If after setting the variables length and top (as we did), we were to retract the setting for length, the deduced values for height and bottom should disappear. The value for top may remain, however. The secretary records history locally in the lists associated with each variable and constraint in the design. Previous states can also be restored locally, combining parts of states from different previous stages of a design. (D. McDermott discusses the use of dependency information in reasoning with and about inequality constraints [D.McDermott 83].) A saved design history can later be replayed, generalized, and made into a procedure or routine with variable parameters. The packager identifies clusters of constraints that interact closely, as measured by the number of variables they share. Constraints with all the same variables are the closest; constraints that share no variables are the least close. There may be several packaging algorithms and any piece of the design may be in several packages simultaneously. Packages also change with the designing. Many of the solver operations, (optimization, for example) are expensive, consuming time and/or memory, especially when they are applied to large numbers of constraints and variables. By identifying closely connected constraints and variables the packager improves the solver's performance.

CHAPTER 7

Review of Related Work

7.1 Overview.

The act of design takes place not only in architecture, but also in many other domains. The constraint exploration model of the design process discussed here was suggested by works in three general categories: the study of design itself as a discipline independent of any particular design domain, the study of design process and methodology in the domain of architecture, and the application of computational techniques and methodologies to support design analysis and synthesis.

In the first category, general theories of design, we consider Herbert Simon's outline for a science of design. Simon proposes to treat design theory as a new discipline and he lists some attendant mathematical techniques [Simon 69; Simon 75]. Simon portrays design as optimizing an objective function over a region bounded by constraints. Simon's constraint formulation of design has had a strong influence on the entire field of design theory. Even models cast in Simon's terms that overcome the limitations of Simon's approach are often confused with Simon's original constraint formulation. We review Simon's model and contrast it with the present approach.

In the second category, methods and theories of architectural design, we discuss two works. First we discuss Christopher Alexander's early Notes on the Synthesis of Form. Alexander, like Simon, describes a design problem as a collection of requirements (in our terms, constraints) and shows a systematic method of partitioning, or decomposing large groups of requirements into smaller groups [Alexander 64]. Methods for partitioning designs, discussed by both Alexander and Simon, are also a part of the present model. Then, briefly, we review the design methods of N. John Habraken and the Stichting Architecten Research (S.A.R.). The S.A.R. methods provide architects and urban designers with tools for the systematic analysis and design of built environments [Habraken et al 76; SAR 73].

In the third category, computational techniques for representing design expertise, we consider the Sketchpad program, an early and insightful constraint-based design environment [Sutherland 63], the IMAGE program, an architectural design aid based on Sketchpad [Johnson and Weinzapfel 71], and the simulation kit Thinglab, that combines constraint-based and object-oriented programming techniques [Borning 77]. Then we review some work of Gerald Sussman's group on compute-assisted design and analysis of electrical circuits [Stallman and Sussman 77; Stallman and Steele 80]. Finally, we briefly consider an alternative approach to representing design expertise, the application of production-rule based expert systems to design.

This review assumes that the reader is familiar with the general fields; although in each case a brief explanation is given, the discussion emphasizes the relation of each of the projects to the present work. We focus only on a few efforts in the field of design methods and computational approaches to design that are closest to the present work, omitting much that is important in any context larger than the present thesis. In addition to the work reviewed here, a few different directions are indicated by the DISCOURSE program [Porter, Lloyd, and Fleisher 70], the Design Problem Solver [Pfefferkorn 75], URBAN5 [Negroponte and Groisser 70], and in integrated circuit design, the REDESIGN system [Steinberg and Mitchell 84], the MAGIC system [Taylor and Ousterhout 84] and the Rectangle Placement Language [Roach 84]. Govela's master's thesis discusses a computational environment for space and function analysis based on the S.A.R. principles [Govela 77]. A good source for older references is the bibliography in the collection of papers edited by Charles Eastman [Eastman 75] and the symposium proceedings edited by Gary Moore [Moore 70]. Integrated circuit design is an especially lively area of research now, and is being automated as rapidly as possible*. Partly as a result of work in integrated circuit design, artificial intelligence is turning to the problems of designing and design methodologies. A recent survey article in a popular artificial intelligence quarterly summarizes some directions of current thought [Mostow 85].

*Current work is reported in the proceedings of the annual IEEE Design Automation conferences.

7.2 Simon's Constraint Formulation of Design.

Herbert Simon, working from concepts of mathematical economics and management theory, outlines topics in a new discipline he calls the "science of design". He argues that design problems can be described as sets of constraints or requirements on the object to be designed. He casts design as a problem in optimization--a problem of maximizing or minimizing an objective or goal within a region of alternatives circumscribed by constraints. In other words, the constraints and the objective are initial conditions for design, and optimizing produces the solution. Simon uses the "diet problem" to illustrate the idea. In the diet problem, the objective is to design a diet that minimizes cost while meeting nutritional requirements; constraints also include the relations between cost, nutrition, and different foods. Simon demonstrates the relevance of optimization techniques when design is seen this way. All the relations in the diet problem are linear equations, and given a particular set of constraints and an objective one can compute a solution that optimizes the given objective subject to the given constraints using only the mathematical technique of linear programming. One might well argue, however, that this definition of design is too constraining; that is, situations with well-defined a-priori constraints and objectives are not typical design situations.

Simon concedes that in many cases it may be difficult to find the best solution, for to know that it is the best, it must be evaluated and compared with all other alternatives. In such cases, Simon argues, any alternative meeting all the constraints--though perhaps not the optimium solution--will suffice. That is, if the given constraints truly describe the design requirements, then any alternative within them should be satisfactory. Accordingly, the first two items on Simon's agenda are techniques for finding an optimum solution, and techniques for finding a satisfactory alternative when the optimum is difficult to find. He calls a procedure that generates an alternative that meets the constraints a "satisficing" procedure (satisfice = satisfy + suffice). Constraint satisfying algorithms are at the heart of computer programs such as Sketchpad [Sutherland 63] and Thinglab [Borning 77] discussed later in this chapter. The assumption in using a constraint satisfier is that so long as the constraints are met, we do not care which alternative among the possibilities the satisfier returns. In other words, the use of a constraint satisfier suggests no preference among alternatives.

When finding the best solution is difficult, Simon suggests settling for a satisfactory one. In architectural design however, and undoubtedly in other domains as well, the problem is not merely computational . The problem is to describe the objective function. Architectural design typically has many different objectives only one of which (by the very nature of optimization) may be optimized over a set of constraints. Deciding on an objective function is part of the design process. Also, constraints and objectives are not fixed at the outset of design, but they are introduced and changed by the designer continuously throughout the exploration for satisfactory variants. Paul Freeman and Allen Newell (working on automatic design of software) saw these difficulties, and they offer the following criticism of the simple constraint formulation of design:

> "The generality and utility of this formulation belies the difficulty of specifying problems in its terms.... All aspects of the formulation contribute to the difficulties: defining the space of possibilities; formulating the constraints; obtaining all the constraints in advance, and creating a reasonable objective function" [Freeman & Newell 77, p. 621].

Though Freeman and Newell do not pursue the constraint model of design*, they do offer suggestions for its improvement. In particular they recommend "relaxing the constraint

* They develop instead a model for "functional reasoning in design", which, they argue, is a poor-man's scheme for satisfying constraints. They work with functional descriptions and see design as a process of connecting structures that provide certain functions with other structures that require those functions.

formulation: permitting the space, the objective function, or the constraints to change, or to become progressively defined throughout a design" [p 621].

Following Simon, the present work uses constraints to describe design problems. But the use of constraints is somewhat different than the use Simon proposes. In part, the present work extends Simon's constraint formulation in the way Freeman and Newell suggest. For Simon, constraints are always part of the initial conditions of the design, and they inflexibly limit the space of solutions that may be explored. In the present model, constraints are redefined, added, and changed dynamically throughout the design process. By exploring different sets of constraints, the designer also explores different regions of alternatives. In Simon's model one goal or objective describes the best design; optimization of that one objective is global, carried out over the region defined by all the constraints. The present model, in contrast, allows for multiple objectives. Here we optimize locally (suboptimize), over small pieces of the design. Simon's model of design differs from the present model on two key points: his constraints are fixed, ours change; where he optimizes globally, we suboptimize locally.

7.3 Models and Methods of Architectural Design.

We shall now take up two well known theories of architectural design: the early work of Christopher Alexander, and the work of N. John Habraken and the Stichting Architecten Research. Both are attempts to account explicitly for the process of designing, in short to answer the question, "how to design". Alexander's presents his thesis in the form of a design theory, whereas Habraken and the S.A.R. embodied their theory in a set of procedures, or design tools.

Like Simon, mathematician-turned-architect Christopher Alexander describes a design problem as a list of requirements on the object to be designed. Alexander takes up one of Simon's topics, the decomposition of complex problems into smaller, simpler pieces. In Notes on the Synthesis of Form he argues that partitioning design problems into chunks of workable size is an important and increasingly difficult part of designing; and he describes an algorithm for decomposing design requirements in this way [Alexander 64] . The algorithm finds groups of requirements that can logically be worked together given a table of connections between constraints. Alexander explains that "...two requirements are linked if what you do about one of them in a design necessarily makes it more difficult or easier to do something about the other" [p 107]. Of course, systematic problem decomposition, also known as "divide and conquer" is no new idea. But its systematic application to design, and in particular to architectural design, was Alexander's contribution. Alexander's concern--how to break into pieces a design problem described as a system of constraints--has since been taken up by others who have explored different algorithms for decomposing, or clustering, large collections of constraints [Kernighan and Lin 70; Milne 70; Owen 70].

The design model presented here incorporates Alexander's thesis about decomposition, although different algorithms may be used. Different partitioning algorithms will group the constraints in different ways, thereby affording the designer alternative views of the design. Also, because constraints change continuously, different partitionings of the design may be employed at different stages of the design process. Finally, Alexander required the designer to enter a table of connections of the different design variables. Because in the constraint explorer, all the relations between variables are explicit, that table can be generated automatically from the given set of constraints.

We now turn to the work of N. John Habraken and the Stichting Architecten Research (S.A.R., the dutch Foundation for Architecture Research) in systematic, analytical, design methods for architects and urban designers [Habraken et al 76; SAR 73]. These procedures have been tested for twenty years now, and are gaining widespread acceptance in the Netherlands as well as elsewhere in Europe. The S.A.R. methods provide a way to notate architectural design rules, using a coordinated system of grids, zones, and margins, and with the help of the notation, a way to systematically explore and compare design alternatives, or variants. The S.A.R. methods help a designer evaluate and compare design variants within a system of rules about the placement and dimensions of building components and use-spaces, and especially respect to the options remaining for subsequent design operations. For example, the architect designing the layout of bearing walls of a building can test the capacity of alternative layouts to support alternative floorplan (space-arrangement) layouts. The S.A.R. methods can also be used to measure the variation implied by a given system of rules.

Fundamental to the S.A.R. methods is the distinction of different levels of physical form (discussed briefly in chapter 4). The S.A.R. methods include procedures for analyzing the capacity of a design to support variation at lower levels. For example, the basic variant analysis procedure calculates different possible room arrangements in a major building structure. The designer distinguishes "support" elements from "infil" elements;the latter are those that can be moved without disturbing the former. The recognition of different levels of elements as members of separate systems that may be controlled by different designers is a cornerstone of the S.A.R. methods. Transformations of the Site [Habraken 83] discusses hierarchical organization of physical elements based on their inherent position dependence, or level, in greater detail.

The S.A.R. methods, in essence, are a systematic way to explore systems of constraints. The present work extends the S.A.R. methods in several ways. Most obviously, the present model is implemented in software, which makes it faster than the same operations carried out by hand. In the constraint explorer, all bookkeeping is done automatically; consequently exploration is much less expensive. The S.A.R. analyses emphasize dimensional comparison (fit/no-fit). That is a certainly one critical aspect of design but it is not the only aspect. Other analyses, for example concerning daylight, thermal performance, structural stability, privacy, enclosure, and noise, might be modelled in the same way. The S.A.R. methods do not preclude such analyses, except insofar as systematically performing many of them by hand for many alternatives is tedious and tiresome. Here we are concerned with exploring consequences of design rules systematically, but we would like to work with more general design rules than those of the S.A.R.

Carried out by hand, the S.A.R. methods require hours of tedious work, and rigorous attendance to a precise notational system. By developing a computational environment for explicitly describing both designs and design procedure, the present work aims to extend the sorts of design rules that we can notate, the complexity of design procedures that we can write and follow, and the number of alternatives we can practically explore.

7.4 Computers and Computational Techniques.

Next we consider a number of computational approaches to representing and using design expertise. We begin with Ivan Sutherland's program, Sketchpad [Sutherland 63]. Sketchpad anticipated many modern computing developments including list-processing, interactive graphics and graphical programming languages, objects and generic hierarchies, and constraint based programming. Although Sutherland titled his report, "A Graphical Man-Machine Interface", Sketchpad was really an early constraint-based computer language.

The Sketchpad user described constraints on a set of graphical objects - points, lines, arcs, angles. Sketchpad then produced an arrangement of the objects that satisfied the constraints or at least came close. Sketchpad tried to satisfy the constraints using two techniques. One technique, that Sutherland calls the "one-pass method", is equivalent to propagation, as Sussman and Steele point out [Sussman and Steele 80]. The other technique, "relaxation of constraints", was used when propagation failed. Sketchpad, when given a set of constraints, constructed an objective function that measured the "sum of errors" (an error measures the extent to which an alternative falls short of optimizing a constraint). Given a set of initial variable values, the relaxation algorithm adjusts these values incrementally so as to reduce the sum of errors. Thus Sketchpad sought a "compromise optimum"; it minimized the sum of squares of local dissatisfactions. The Sketchpad user had no control over this objective function, except by altering the constraints themselves. Relaxation, a hill-climbing procedure, searches the region of alternatives for the solution that best satisfies the compromise objective function, by proceeding in the direction of increasing quality (lower sum of errors). It finds local optima but cannnot tell whether there is more globally a better alternative.

The Image program [Johnson and Weinzapfel 71] provided an architectural interface to Sketchpad. The designer's role in the Image program is to provide constraints and monitor solutions. The constraint satisfier inside Sketchpad selects one alternative that meets the constraints and Image draws the alternative. The designer may then change the constraints and try again. The designers of the Image program understood that constraints in design shift and change, thus Image was intended to be used iteratively, in a sort of specify-satisfy cycle. Image provides the designer with a catalog of geometric and very simple architectural constraints--proximity, adjacency, line-of-sight, that can be selected and applied to graphic elements such as lines, points, circles, and rectangles in a field. For example, the designer might constrain two rectangles to be adjacent, and with lines of sight between each of them and also a third rectangle some distance away from each. The Sketchpad satisfier then operates on all the constraints together as described above and produces a set of values for the variables that satisfies--or at least tries to satisfy--all the given constraints. The resulting data-set is then displayed as a drawing.

In the constraint explorer, constraints may change continuously and incrementally; optimizations, when performed, are local, not global. In Image, constraints may only be changed after a complete "run" of the Sketchpad satisfier; Image attempts to satisfy the entire system of constraints simultaneously. Unlike Image, the present work also proposes to partition or decompose the design and work pieces separately, providing the designer with greater control over the process of selecting an alternative.

Alan Borning developed some of Sutherland's ideas in the Sketchpad program in a more modern programming environment [Borning 77, 79]. Borning's simulation kit, Thinglab, incorporates constraint satisfaction in an object-oriented programming environment. The Thinglab user describes constraints on properties of objects; Thinglab's constraint satisfying mechanisms (similar to those of Sketchpad) attempt to maintain the given relations between those properties. For examples, Borning shows Thinglab maintaining the simple metric and geometric relations between sides of a quadrilateral, simulating the stresses in a bridge under load, and maintaining constraints on the layout of text in a document. Thinglab is notable for its elegant integration of constraint-based and object-oriented programming (it was written in the Smalltalk programming language) also Thinglab maintains both class/instance relations between descriptions in its database, but also a part-whole hierarchy. The present work would extend Thinglab's capabilities by keeping track of dependencies in the design, by providing automatic decomposition of designs, and by providing a solver capable of symbolic mathematics.

Gerald Sussman and his group at MIT have worked extensively on the theory and methods of electrical and integrated circuit design. They have developed computational formalisms for design and have suggested ways to structure computer programs to aid in both design analysis and synthesis. (Sussman's Ph.D. thesis described HACKER, a program that designed sequences of robot-arm operations for constructing configurations in a simple world of blocks, was based on a paradigm of design as debugging almost-right plans [Sussman 73]). Steele and Sussman's Constraints paper [Steele and Sussman 80] provides an introduction to constraint-based programming techniques. That paper also discusses the differences between declarative and imperative programming styles. Electronic components such as resistors and capacitors are composed by combining packages of simple arithmetic constraints; for example, they would model a resistor as a package of constraints on the voltage and current at its terminals, the voltage being constrained to equal the product of the resistance and the current across the terminals. A circuit is constructed by connecting the packages of constraints that represent the circuit components. The behavior of the circuit can then be simulated by fixing voltages and/or currents at various points in the circuit. If component values (resistances and capacitances) are left unfixed, then fixing the voltages and currents at enough places in the circuit may determine the component values. Local propagation of currents and voltages in the circuit are sufficient to simulate some but not all circuit behaviors. Some circuit behaviors require a higher, algebraic, level of analysis; for these Steele and Sussman recommend the technique of "slices". This involves substituting an equivalent but easier to analyze sub-circuit (slice) for some difficult-to-analyze part of the circuit; hence the method of slices is similar to algebraic manipulation of the constraints in symbolic form. Search is sometimes needed in circuit behavior analysis, for example to determine which of several plausible operating ranges of a transistor in a circuit is correct. Chronological backtracking in the event of search failure is wasteful. In their programs EL and ARS, Sussman and Stallman introduced the idea of dependency-directed backtracking, a technique for remembering the assumptions and the chain of deductions that lead to a design state and for using this information to avoid repeating the same mistake when searching [Stallman and Sussman 77] . Every item in a dependency network is linked to other items in the database that were used to deduce it. Steele's Ph.D. thesis packages together several techniques for managing constraints in a proposal for a general-purpose constraint language [Steele 80].

In the last phase of integrated circuit design, layout constraints are important. These constraints, involving the location, orientation, and connections of elements and configurations is similar in some ways to many architectural constraints. One program developed in Sussman's group, the Design Procedure Language [Batali and Hartehimer 82], manages constraints on the positions and dimensions of the thousands of rectangles of silicon and metal on a VLSI (Very Large Scale Integrated-circuit) chip* . In addition to layout design, the

* An introduction to the general problems of circuit layout is provided in the overview article Circuit Layout [Soukup 81].

electronic (both electric and digital logic) behavior of integrated circuit components at several levels of abstraction can also be modelled within the constraints paradigm. [Sussman, Holloway, and Knight 79] .

We turn finally from the constraints model to a different computational model of expertise, the use of production-rules to represent expertise. In a rule-based expert system, all knowledge is represented as a structured set of If-Then rules. Each rule represents an inference that an expert might make about the subject. Rule-based expert systems have been used so successfully in various domains that the term "expert system" has come to imply a production-rule system. The technology involved dates back to the General Problem Solver program [Newell and Simon 63]. MYCIN, a rule-based expert-system for determining therapeutic regimes for patients with infectious diseases is a model for a large class of knowledge-based expert systems using this technology [Shortliffe 76], but rule-based expert systems have been developed in many different task domains, for example: configuring the components and cables of VAX computer installations [McDermott 82], selecting the most likely site for mineral exploration based on seismic data [Duda et al 78], and performing checking, routing, and optimal placement in integrated circuit design have also been developed [Williams 77; Kim and J. McDermott 83; Steinberg and Mitchell 84]. It is only a matter of time before architectural applications are built; many such efforts are presently underway. Rule-based expert systems are presently a very active area of research, and some interesting variations on the theme described here abound. We will not attempt a review of the field. An informative review of rule-based expert systems appeared in Science [Duda and Shortliffe 83].

The rule base in this kind of expert system typically contains many hundreds of such If-Then rules, or deductions. Of course, the rules cannot be entered in an ad-hoc way, but must be structured to lead to conclusions from given sets of propositions. MYCIN for example, always begins with a description of symptoms and results of clinical tests, and finshes with a diagnosis. The structuring and preparation of this knowledge base is the main task and the most time consuming one in programming this sort of expert system.

All rule based expert systems contain essentially the same simple mechanism for logical deduction, a so-called "inference engine". Given a set of initial assertions (the problem statement), the inference engine searches the rule base for deductions to make from the given assertions. It then asserts these deductions and iterates. The program again searches its knowledge base, producing a second set of deductions. When the program needs information that was not initially supplied it queries the user. This reasoning cycle continues until the program reaches a goal, in the case of MYCIN, when it has achieved a diagnosis and prescribed a therapeutic regime for the patient, or until repeated iterations of the inference engine reveal no further deductions .

Why not stick with rule-based systems, instead of venturing into constraint-based programming techniques? Given their success, why consider other representations for design expertise. No doubt there are applications of rule-based technology in design. However, rule-based systems have thus far proven best at problem-solving tasks with single best solutions and objectives that are well-defined initially. But we have argued that design expertise is not in solving problems, but in exploring for solutions. Rule-based programming can be seen as a subset of constraint-based programming, in which "implies" is the principal relation. However, not all design conditions and expertise seem easily amenable to expression in the implicational (If-Then) form required by rule-based systems. Design expertise involves informed preference among alternatives as well as logical deduction; it is exploratory as well as goal-oriented. A general-purpose programming environment for design may well incorporate a production-rule language but there it need not stop.

7.5 Discussion and Further Work.

We have viewed designing as the exploration of different sets of constraints and the regions they bound. This idea we have called a theory of designing. The theory is not a manifesto that prescribes good design; rather it is about what we know and how we use what we know in order to design. In the theory there is a place for specific knowledge about a particular design domain. It accounts for various activities in a design process: making decisions, exercising preferences, exploring possibilities, choosing among alternatives,and backtracking. The theory is stated generally, as it might be applied to all design domains and is illustrated with examples from the design of the built environment. Architectural design, perhaps the most archetypal of all design domains seems especially appropriate as a test-domain for any design theory. It is assumed that constraints can be used to represent design expertise and describes how expertise so represented might be exercised. We have discussed a computational model that supports the theory. We discussed the use of the program and sketched how it might be built, and we have seen how a body of well-defined and useful design procedures (the S.A.R. methods) relate to the proposed theoretical framework.

The purpose of building a computational model is to test constraints as a representation for design expertise and to provide designers with a higher-level platform for programming than conventional languages and computer-assisted design programs. The constraint explorer provides a framework for representing and manipulating design expertise. The actual expertise must come from experts and will vary across domains and across experts, and therefore the present work stops short of prescribing it.

The present work makes no claims for completeness or exclusivity. The present model may not account for <u>all</u> of designing, and there may be other ways of looking at the things that the present model does account for. There may well be limitations to what can be expressed using constraints. Therefore, in applying the theory to various design domains, it is only prudent to watch for expertise that <u>can</u> be expressed, but not within the framework of constraints proposed here. At any rate we shall judge the constraint model a success if we are able to use its formalisms to articulate and express a body of design expertise from a small but well-defined subdomain of architectural design. That then is naturally the next goal of the project. The usefulness of the computational model lies in the ability of architects to take over its mechanisms and use them for their own purposes; to extend the built-in primitive relations with more complex architectural ones.

The computer program described here is intended to provide a computational environment for (a) articulating design constraints, (b) developing procedures and strategies to manipulate them, and (c) studying the expertise embodied in (a) and (b). It is intended both as a repository for design knowledge and expertise as well as a tool for designers. The constraint explorer described here is not trivial to construct. Given a reasonable computing environment, however, the task is doable. Although trial versions of almost all the pieces of the constraint explorer program have been built and tested on various computers at various times, the entire program has yet to be assembled together under one operating system. The next step therefore is to build the program, and then to begin to use it to express design expertise.

Abelson, H. and Sussman, G. J. with Sussman, J. 1985. *The Structure and Interpretation of Computer Programs..* Cambridge: M.I.T. Press and New York: McGraw-Hill.

Alexander, C. 1964. *Notes on the Synthesis of Form.* Cambridge Massachusetts: Harvard University Press.

Alexander, C. and Mannheim, M. 1962. *HIDECS-2: A Computer Program for the Hierarchical Decomposition of a Set with an Associated Graph.* M.I.T. Civil Engineering Systems Laboratory Publications No 160.

Batali, J. and Hartheimer A. 1982. *Design Procedure Language Manual.* M.I.T. Artificial Intelligence Laboratory Memo #598.

Bobrow, D. and Winograd, T. 1977. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science* 1:3-46.

Bobrow, D. and Collins, A. (ed.) 1975. Representation and Understanding: Studies in Cognitive Science. Academic Press.

Borning, A. 1977. Thinglab--an Object-oriented System for Building Simulations Using Constraints. In *Proc. Fifth International Joint Conference on Artificial Intelligence* pp. 497-498.

Borning, A. 1979. *Thinglab--A Constraint-Oriented Simulation Laboratory* Xerox Palo Alto Research Center report SSL-79-3, Palo Alto, California.

Brachman, R. and Smith, B.C., eds. 1980. Special issue on knowledge representation. SIGart newsletter no. 70 (February).

Dahl, O. and Nygaard, K. 1966. SIMULA - An Algol-based simulation Language. *CACM* 9(9):671-681.

Drescher, G. L. 1985. *Objectlisp Manual.* M.I.T. Artificial Intelligence Memo (forthcoming).

Duda, R.O., Hart, P.E., Nilsson, N.J., Konolige, K., Reboh, R., Barrett, P., and Slocum, J. 1978. *Development of the PROSPECTOR consultation system for mineral exploration.* Final Reprt, SRI Projects 5821 and 6415, SRI International, Inc. Menlo Park, California.

Duda, R.O. and Shortliffe, E.H. 1983. Expert Systems Research. *Science* 220:261-268.

Eastman, C. M. (ed.) 1975. *Spatial Synthesis in Building Design.* New York: Wiley & Sons.

Engel, H. 1964. *The Japanese House--A Tradition for Contemporary Architecture.* Tokyo, Japan and Rutland, Vermont: Charles E. Tuttle & Co.

Fahlman, S. E. 1979. *NETL: A System for Representing and Using Real-World Knowledge.* Cambridge, Massachusetts: M.I.T. Press.

Fenichel, R. R. An On-Line System for Algebraic Manipulation. Ph.D. dissertation, Harvard University, 1966.

Freeman, P. and Newell, A. 1971. A Model for Functional Reasoning in Design. *Proc. Second International Joint Conference on Artificial Intelligence,* pp. 621-640, British Computer Society: London.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

Goldstein, I. and Bobrow, D. 1981. Layered Networks as a Tool for Software Development. *Proc. Seventh International Joint Conference on Artificial Intelligence.* Vancouver.

Gosling, J. 1983. Algebraic Constraints. Ph.D. dissertation, Carnegie Mellon University (CMU-CS-83-132).

Govela, A. 1977. Space and Function Analysis -- a Computer System for the Generation of Functional Layouts in the S.A.R. Methodology. Master's thesis, Massachusetts Institute of Technology.

Habraken, N.J., Boekholt J.T., Thijssen A.P., and Dinjens, P.J.M. 1976.*V ariations - The Systematic Design of Supports*. Laboratory for Architecture and Planning/MIT Press.

Habraken, N.J., Akbar, J., and Liu, L. 1983. *Thematic Design.* Design Theory and Methods Group Working Paper, Department of Architecture, M.I.T.

Habraken, N.J. 1983. *Transformations of the Site.* Cambridge, Massachusetts: Awater Press.

Hille, R.T. 1982. Understanding and Transforming What's There-- A Look at the Formal Rule Structure of the Residential Facade Zone in Victorian San Francisco. Master's thesis, Massachusetts Institute of Technology.

Johnson, T. and Weinzapfel, G. 1971. Computer Assisted Space Synthesis under Geometric Constraints. *Industrial Forum* 1:17-24.

Johnson, T.; Weinzapfel G.; Perkins, J; Ju, D.; Solo, T.; Morris, D. 1970. *IMAGE: An Interactive Graphics-based Computer System for Multi-Constrained Synthesis.* Department of Architecture, M.I.T.

Kay, A. 1984. Computer Software. *Scientific American.* **251**(3):53-59.

Kernighan, B.W. and Lin, S. 1970. An Efficient Procedure for Partitioning Graphs. *Bell Systems Technical Journal* (Febuary).

Kim, J. and McDermott, J. 1983. TALIB: an IC Layout Design Assistant. In *Proc. National Conference on Artificial Intelligence*, pp 197-201, Los Altos: William Kaufmann.

Konopasek, M. and Papaconstadopoulos, C. 1978. The Question Answering System on Mathematical Models (QAS): Description of the Language. *Computer Lanugages,* v 3:144-155, Pergammon Press, Ltd.

Kowalski, R.A. 1979. *Logic for Problem Solving.* New York: North Holland.

Levitt, D. A. 1985. A Representation for Musical Dialects. Ph.D. dissertation, Massachusetts Institute of Technology.

Macsyma 1982. *MACSYMA Reference Manual.* The Macsyma Group, Laboratory of Computer Science, M.I.T., Cambridge Massachusetts.

McDermott, D. 1983. Data Dependencies on Inequalities. In *Proc. National Conference on Artificial Intelligence*, pp 266-269, Los Altos: William Kaufmann.

McDermott, J. 1982. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence* **19**(9):39-88.

Minsky, M. R. 1984. Manipulating Simulated Objects using a Force and Touch Sensitive Display. *Proceedings ACM SIGraph Conference 84.*

Mitchell, W. J. 1977. *Computer Aided Architectural Design.* New York: Petrocelli/Charter.

Mostow, J. 1985. Toward Better Models of the Design Process. In *The AI Magazine* **2**(1):44-57.

Moore, R. E. 1966. *Interval Analysis.* Englewood Cliffs, New Jersey: Prentice-Hall.

Milne, M. 1970. CLUSTER: A Structure-Finding Algorithm. In *Emerging Methods in Environmental Design and Planning*, ed. G. Moore, pp.126-133, Cambridge: M.I.T. Press.

Negroponte, N. 1975. *Soft Architecture Machines.* Cambridge: M.I.T. Press.

Negroponte, N. and Groisser, L. 1970. URBAN5 - A Machine that Discusses Design. In *Emerging Methods in Environmental Design and Planning*, ed. G. Moore, pp.105-115, Cambridge: M.I.T. Press.

Newell, A. and Simon H.A. 1963. GPS, a Program that Simulates Human Thought. In *Computers and Thought*, ed. Feigenbaum, E. and Feldman, J.. Mc Graw Hill.

Nilsson, J. N. 1984. Change and Continuity in Urban Form: A Case Study of East Oakland's Store-and-Flat Buildings. Master's Thesis, University of California, Berkeley.

Owen, C. 1970. DCMPOS: An Algorithm for the Decomposition of Nondirected Graphs. In *Emerging Methods in Environmental Design and Planning*, ed. G. Moore, pp.133-147, Cambridge: M.I.T. Press.

Pangaro, P.A. 1982. Beyond Menus: The Rats-a-Stratz or the Bahdeens. In *Proceedings Harvard Computer Graphics Week,* Cambridge: Harvard University.

Pfefferkorn, C. E. 1975. The Design Problem Solver: a System for Designing Equipment or Furniture Layouts. In *Spatial Synthesis in Computer-Aided Building Design,* ed. C.M. Eastman, pp. 98-147, New York: Wiley.

Porter, W., Lloyd, K., and Fleisher, A. 1970. DISCOURSE: A Language and System for Computer Assisted City Design. In *Emerging Methods in Environmental Design and Planning*, ed. G. Moore, pp.92-105, Cambridge: M.I.T. Press.

Polya, G. 1945. *How To Solve It; A New Aspect of Mathematical Method.* Princeton: Princeton University Press.

Roach, J. A. 1984. The Rectangle Placement Language. In *Proc. 1984 IEEE Design Automation Conference*, pp 405-411.

SAR 1973. SAR 73: S.A.R. Method for the Development of Urban Environments. Stichting Architecten Research.

Shortliffe, E. H. 1976. *Computer-based Medical Consultations: MYCIN,* American Elsevier.

Simon, H. A. 1969. *The Sciences of the Artificial.* Cambridge, MIT Press.

Simon, H. A. 1975. Style in Design. In *Spatial Synthesis in Computer-Aided Building Design,* ed. C.M. Eastman, pp. 98-147, New York: Wiley.

Smith, M. K., Hille R.T., and Mignucci, A. 1982. Ranges of Continuity: Eleven Towns in Spain and Portugal. Exhibit, Department of Architecture, M.I.T.

Soukup, J. 1981. Circuit Layout. In *Proceedings of the IEEE.* **69**(10):1281-1304.

Stallman, R. and Sussman, G.J. 1977. Forward Reasoning and Dependency-directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*, 9:135-196.

Steele, G.L. 1980. *The Definition and Implementation of a Programming Language Based on Constraints.* M.I.T. Artificial Intelligence Laboratory Technical Report TR-595, Massachusetts Institute of Technology, Cambridge.

Steele, G. and Sussman, G.J. 1980. CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence* 14:1-39.

Steinberg, L. I. and Mitchell, T. M. 1984. A Knowedge Based Approach to VLSI CAD - The REDESIGN System. In *Proceedings of the 21st Design Automation Conference* (pp 412-418).

Sussman, G.J. 1973. A Computational Model of Skill Acquisition. Ph.D. dissertation, Massachusetts Institute of Technology.

Sussman, G.J., Holloway, J., and Knight, T. 1979. *Computer-Aided Evolutionary Design for Digital Integrated Systems*. M.I.T. Artificial Intelligence Laboratory AI Memo 526. Massachusetts Institute of Technology, Cambridge.

Sutherland, I. 1963. *Sketchpad - A Man-Machine Graphical Communication System*, Technical Report No. 296, Lincoln Laboratory, Massachusetts Institute of Technology, Cambridge.

Taylor, G.S. and Ousterhout, John K. 1984. Magic's Incremental Design-Rule Checker. In *Proceedings of the 21st IEEE Design Automation Conference* (pp. 160-165).

Vernez-Moudon, A. 1985. *Built for Change - Urban Architecture in San Francisco* Cambridge, MIT Press 1985 (forthcoming)

Weinzapfel, G. and Handel, S. 1975. Image: Computer Assistant for Architectural Design. In*Spatial Synthesis in Computer-Aided Building Design,* ed. C.M. Eastman, pp. 61-98, New York: Wiley.

Williams, J.D. 1977. Sticks, A New Approach to LSI Design. Master's thesis, Massachusetts Institute of Technology, Cambridge.

Winograd, T. 1975. Breaking the Complexity Barrier, Again. In *ACM SIGPLAN Notices* 1:13-30.

Wolfram, S. 1984. Computer Software in Science and Mathematics. In *Scientific American,* **251**(3):188-203.