# The Architect's Collaborator: Toward Intelligent Tools for Conceptual Design

BARKER

by

Kimberle Koile

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
APR 2 4 2001
LIBRARIES

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2001

© Massachusetts Institute of Technology, 2001. All Rights Reserved.

Author .....
Electrical Engineering and Computer Science
February 5, 2001

Certified by .....
Randall Davis
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

# The Architect's Collaborator: Toward Intelligent Tools for Conceptual Design

by

Kimberle Koile

Submitted to the Department of Electrical Engineering and
Computer Science on February 5, 2001, in partial fulfillment of the
requirements for the degree of Doctor of Philosophy

## Abstract

In early stages of architectural design, as in other design domains, the language used is often very abstract. In architectural design, for example, architects and their clients use experiential terms such as "private" or "open" to describe spaces. If we are to build programs that can help designers during this early-stage design, we must give those programs the capability to deal with concepts on the level of such abstractions. The work reported in this thesis sought to do that, focusing on two key questions: How are abstract terms such as "private" and "open" translated into physical form? How might one build a tool to assist designers with this process?

The Architect's Collaborator (TAC) was built to explore these issues. It is a design assistant that supports iterative design refinement, and that represents and reasons about how experiential qualities are manifested in physical form. Given a starting design and a set of design goals, TAC explores the space of possible designs in search of solutions that satisfy the goals. It employs a strategy we've called dependency-directed redesign: it evaluates a design with respect to a set of goals, then uses an explanation of the evaluation to guide proposal and refinement of repair suggestions; it then carries out the repair suggestions to create new designs.

A series of experiments was run to study TAC's behavior. Issues of control structure, goal set size, goal order, and modification operator capabilities were explored. In addition, TAC's use as a design assistant was studied in an experiment using a house in the process of being redesigned. TAC's use as an analysis tool was studied in an experiment using Frank Lloyd Wright's Prairie houses.

Thesis Supervisor: Randall Davis
Title: Professor

# Acknowledgments

It's about adventure.

— Richard Feynman

Researching and writing this thesis has certainly been an adventure—in AI, in architecture, in remembering to eat and sleep. I am profoundly grateful to those who helped me along the way.

My Committee:

Randy Davis, Aaron Fleisher, Howie Shrobe, Patrick Winston

AI:

Randy Davis, Howie Shrobe, Patrick Winston, John Aspinall, Tomás Lozano-Peréz

Architecture:

Aaron Fleisher, Richard Krauss, Duncan Kincaid, Mark Gross, Stephen Ervin, Bill Mitchell, Bill Porter, Don Schön, The Architects Collaborative

Life:

John and Michael Aspinall, Earl and Carmon Koile, Kristen Wells, Stephen Koile, Scott McKay and Laura Need, Andee Rubin and Liz Bradley, Annie LaCourt and Mark Burstein, Una-May O'Reilly, Lisa Tucker-Kellogg, Sue Felshin, Larry Hunter, Noel Chiappa, Susan Wilson

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The Architect's Collaborator (TAC) is a prototype design support system for early stages of architectural design. It supports iterative design refinement, representing and reasoning about how experiential qualities are manifested in physical form. It assists a designer in specifying goals and in exploring the space of possible designs. It evaluates design goals, proposes and refines repair suggestions for unsatisfied goals, and carries out those suggestions to create new designs.

The Architect's Collaborator was built in order to answer two questions:

- How are experiential qualities translated into physical form?
- How might a tool assist designers with this process?

Before discussing TAC's answers, it's important to know what we mean by experiential qualities and by their being manifested in physical form.

## 1.1 The Problem

By *experiential qualities* we mean qualities such as openness or privacy, concepts that architects and their clients use when describing buildings and spaces. By *physical form* we mean the design elements—walls, windows, doors—that create buildings and spaces. Experiential qualities and physical form are intimately related: The form that creates a space shapes the way in which we experience that space. This shaping of experience is what we mean by experiential qualities being manifested in physical form. Here are some examples.

Imagine sitting at one of the desks shown in Figure 1.1.

**Figure 1.1:** Classroom.

Now imagine sitting at one of the desks shown in Figure 1.2, and ask yourself why your experience would be so different.

**Figure 1.2:** Studio at Taliesin West, by Frank Lloyd Wright

You'll invariably think about the height of the ceiling, the light from above, the natural wood. All these aspects shape your experience, and not accidentally. Frank Lloyd Wright intended and expected for you to be profoundly influenced by this environment.

Looking at another example, imagine how it would feel to sit in the living room shown below.

**Figure 1.3:** Living room.

Now imagine sitting in the living room shown in Figure 1.4. Why does this living room feel different?

**Figure 1.4:** Living room in the Hanna house, by Frank Lloyd Wright

Two living rooms; two very different uses of materials, light, arrangements of walls and windows; and two very different experiences.

As these two examples have illustrated, experiential qualities of a space are manifested in the physical form that creates that space. What's more, those experiential qualities are paramount in architectural design. As Frank Lloyd Wright said of another of his buildings:[1]

> ...the reality of the building did not consist in the walls and in the roof, but in the space within to be lived in.

## 1.2 Our Solution

In the course of building The Architect's Collaborator we discovered how experiential qualities could be translated into physical form, and how a tool could assist designers with this process.

• How are experiential qualities translated into physical form?

TAC's languages and representations formalize abstract concepts such as openness and privacy, and relate those concepts to arrangements of physical form.

• How might a tool assist designers with this process?

TAC enables designers to specify design goals and explore the space of possible designs satisfying those goals. TAC employs what we've called *dependency-directed redesign*: it evaluates

1. Unity Temple in Oak Park, Illinois.

design goals, proposes and refines repair suggestions, and creates new designs by carrying out those suggestions. By means of dependency-directed redesign, TAC is able to deal with issues of a very large search space, noninvertible evaluation functions, complex interactions between design modification operators, and multiple conflicting goals.

## 1.3 An Example

We envision The Architect's Collaborator being used in the following fashion.

> Imagine that you're sketching a design, pen in hand. You tell the computer near you that the design is for a client who wants a house with main living spaces that feel open to one another, but private with respect to the front door. You have ideas about physical forms that manifest feelings of privacy and others that manifest feelings of openness, and you're translating those ideas into lines and annotations on a page. You stop to assess your latest sketch and ask the computer for its comments. It shows you regions defined by your proposed physical forms. It shows you what is visible from each region and from the front door. You notice that a large portion of the living room is visible from the front door, so you add a screen; the computer shows you what is now visible with a screen in place. You also notice that the stair blocks the view of the dining room from the living room. You ask the computer to increase the visual openness of the dining room without sacrificing privacy with respect to the front door or ease of access to the stairs. It suggests two other locations for the stair, showing you how the visual openness, privacy, and access are affected by each location. It shows you several other possible stair locations, but points out that in these cases, while visual openness increases, privacy of the dining room decreases, and the stair is not as easily accessed. You like one of the first two proposed locations, accept it, and continue sketching and consulting with your computer.

As shown in the following example, The Architect's Collaborator is able to perform all the steps in the above scenario, apart from operating on a sketched design. It:

1. displays regions defined by the physical form
2. computes and displays visual openness measurements for all regions
3. computes and displays visual openness measurements for the living region from the front door, with and without a screen[2]
4. proposes repair suggestions for increasing visual openness between dining and living
5. creates new designs by carrying out repair suggestions
6. evaluates all goals to check for solutions
7. displays rejected designs

---

2. By "screen" we mean a perforated structure such as a stair railing or a bookcase without a back. An example of a screen is shown in Figure 8.27.

Consider the design below.

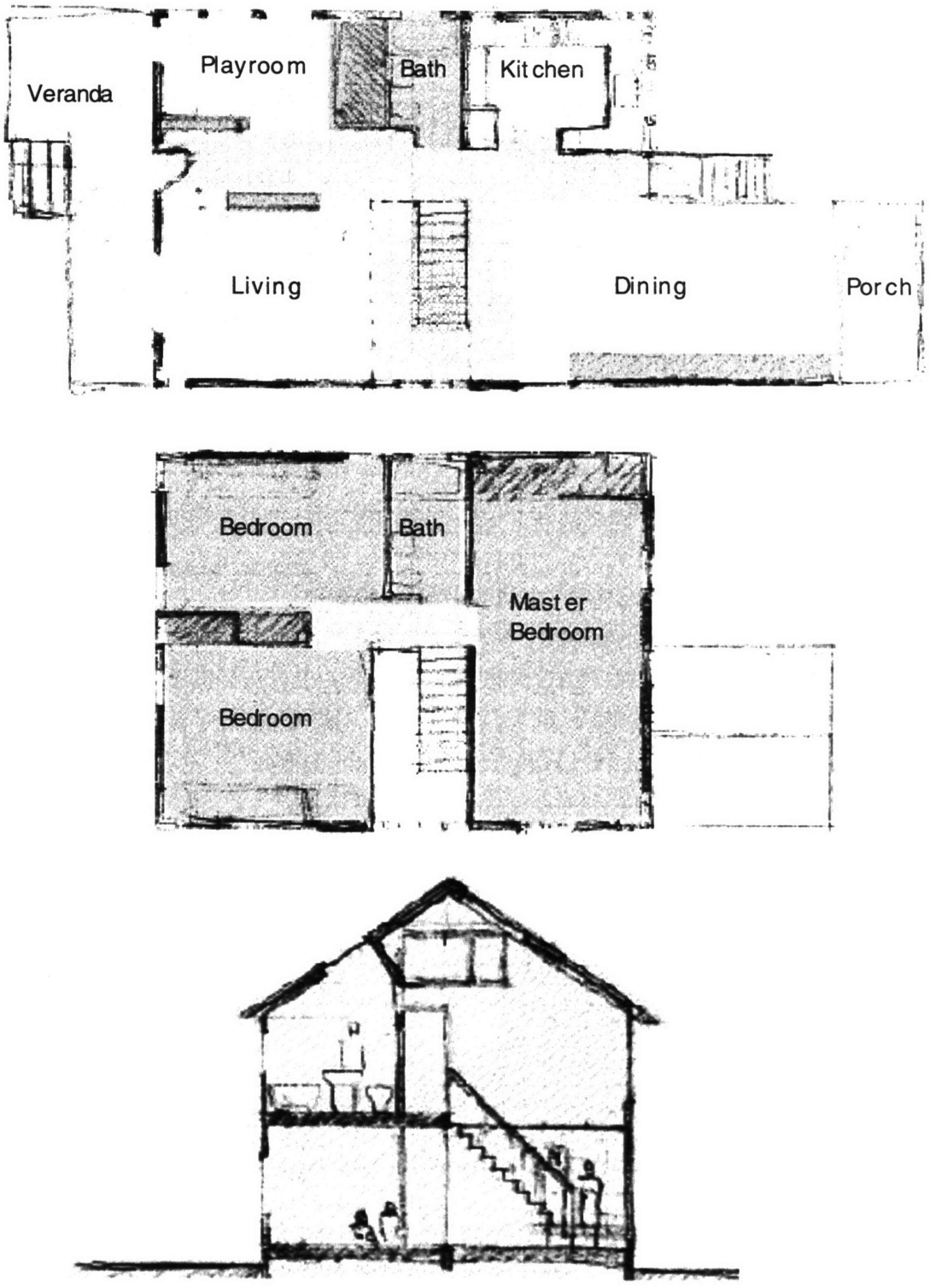



**Figure 1.5:** Design example: first floor, second floor, section through stair.[3]

3. Drawings courtesy of Duncan Kincaid and Daniel Gorini. See Appendix A for a description of diagram conventions used throughout this document.

1. Displaying regions

We consider the first floor of this design. The designer asks TAC to display the regions defined by the physical form,[4] then he supplies region names based on intended use.

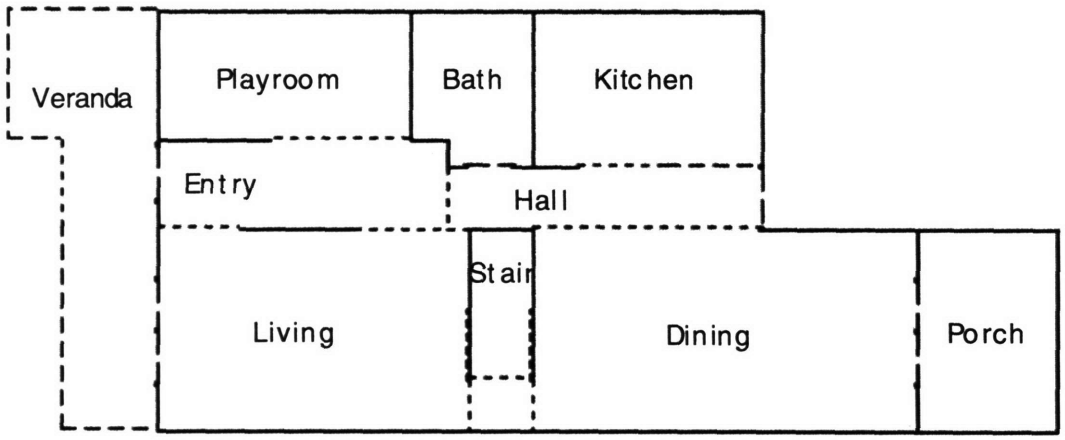



**Figure 1.6:** TAC computes and displays the regions created by the physical form.

2. Computing and displaying visual openness measurements

The designer is interested in visibility issues and asks TAC to display what is visible from each region and from the front door. TAC responds by computing and displaying visual openness measures, which represent the portion of an area that is visible from a particular location. The portion of the design visible from the Living region is shown below.

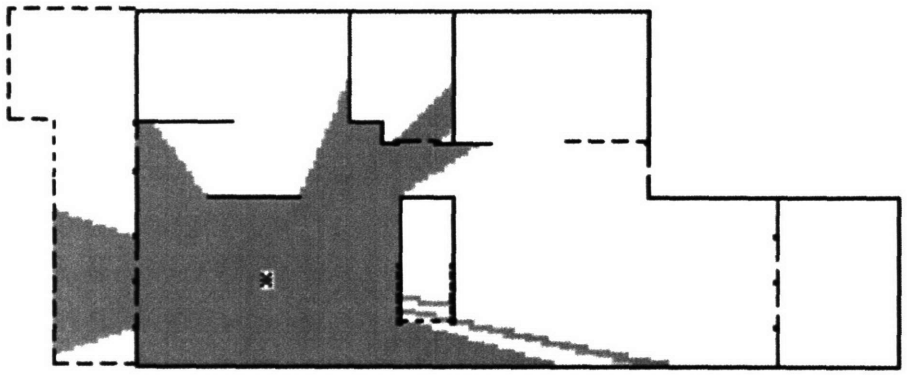



**Figure 1.7:** The shaded area is visible from the Living region; * represents a viewpoint at the center of the region. The first visual openness measure represents the portion of all regions visible from the viewpoint; the second measure represents the portion of the Dining region visible from the viewpoint.

4. A note about TAC's use of 2D floorplans: TAC is interesting because of its design capabilities, not its visualization capabilities. More sophisticated visualization capabilities (e.g. 3D, virtual reality) are possible, but are outside the scope of this research.

The figure below shows the area visible from the front door and gives the visual openness measurements that represent the portion of Living and Dining regions visible from the front door.

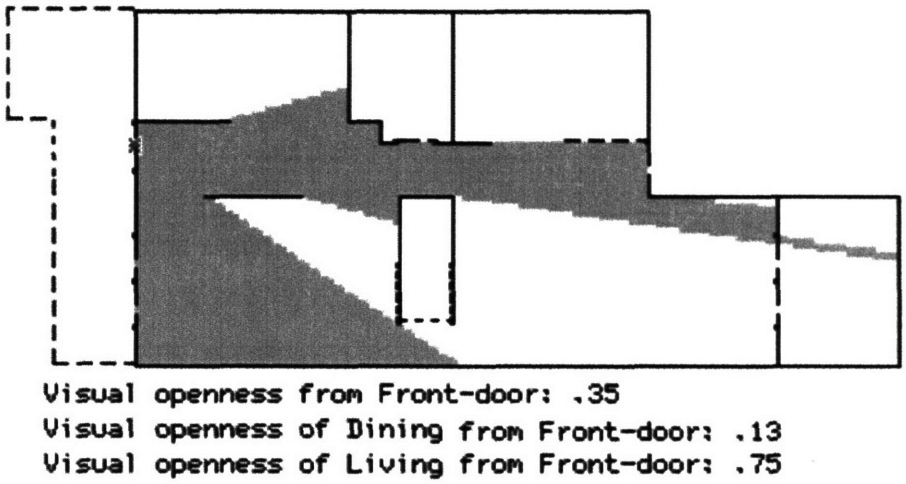



**Figure 1.8:** Shaded area is visible from the front door; * represents a viewpoint at front door.

3. Computing and displaying visual openness measurements for Living from front door with and without an added screen

The designer notices that a large portion of the Living region is visible from the front door and asks TAC to add a screen between the Entry and Living regions and then to calculate the visual openness of the Living region from the front door for the new design. TAC verifies that the visual openness decreases. (Alternatively, the designer could ask TAC to decrease the visual openness between the Living and Entry territories, and TAC would suggest adding the screen.) The original design and the new design, along with their visual openness measurements, are shown in Figure 1.9.

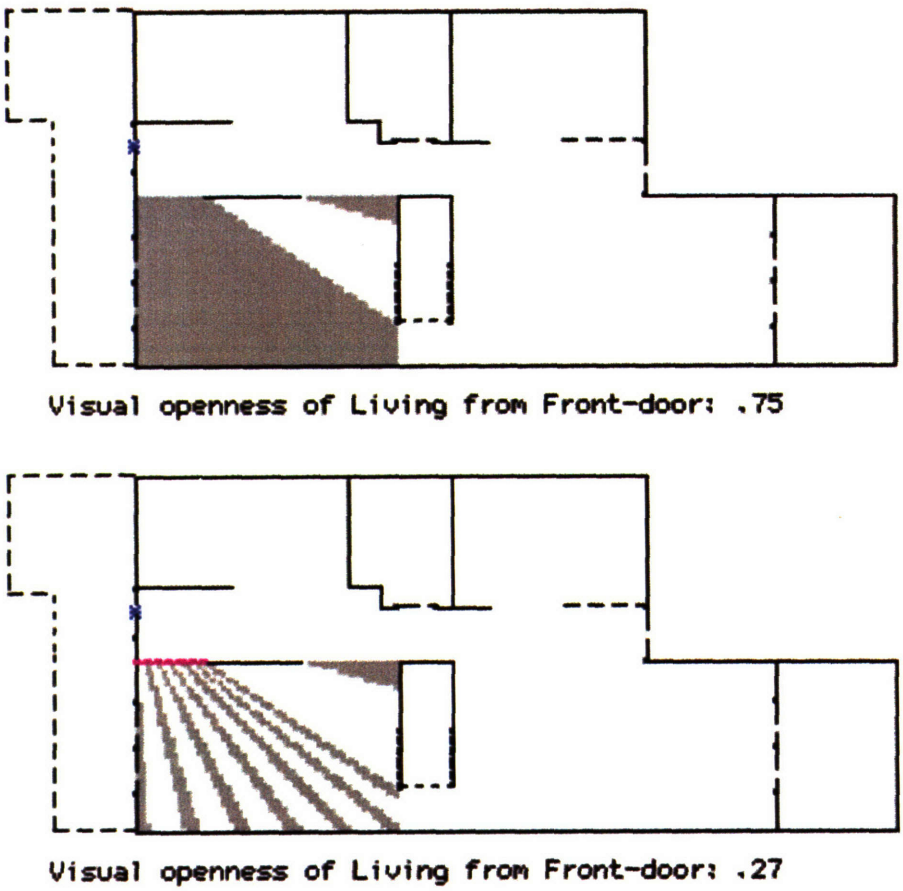



**Figure 1.9:** Verifying that a screen (in lower design) decreases visual openness of the Living region from front door; * represents viewpoint at front door.

4. Proposing repair suggestions for increasing visual openness of Dining from Living

The designer notices that a stair blocks much of the view of the Dining region from the Living region. He asks TAC to suggest ways to make the Dining region visually open from the Living region, while keeping both regions relatively private with respect to the front door and keeping the stair easily accessible. TAC translates "relatively private" into a goal about visual openness with respect to the front door, and it translates "easily accessible" into a goal about length and change in direction along the shortest path between the stair and the exterior doors (front door and back door). TAC now has three goals: have more visual openness between the Dining and Living regions; have no more than 40% of the Dining and Living regions visible from the front door (i.e. have the visual openness values less than 0.40); have the path length and change in direction between the stair and exterior doors less than or equal to current values. Only the first of these goals, the visual openness between the Dining and Living regions, is not satisfied. To satisfy this

goal, TAC proposes rotating the stair 90 degrees or 270 degrees, or moving the stair to any of six exterior edges.

5. Creating new designs

TAC carries out its suggestions, creating two new designs with the stair rotated and six new designs with the stair on an exterior edge. The two designs with rotated stair are shown below. (The other new designs are discussed in step 7.)

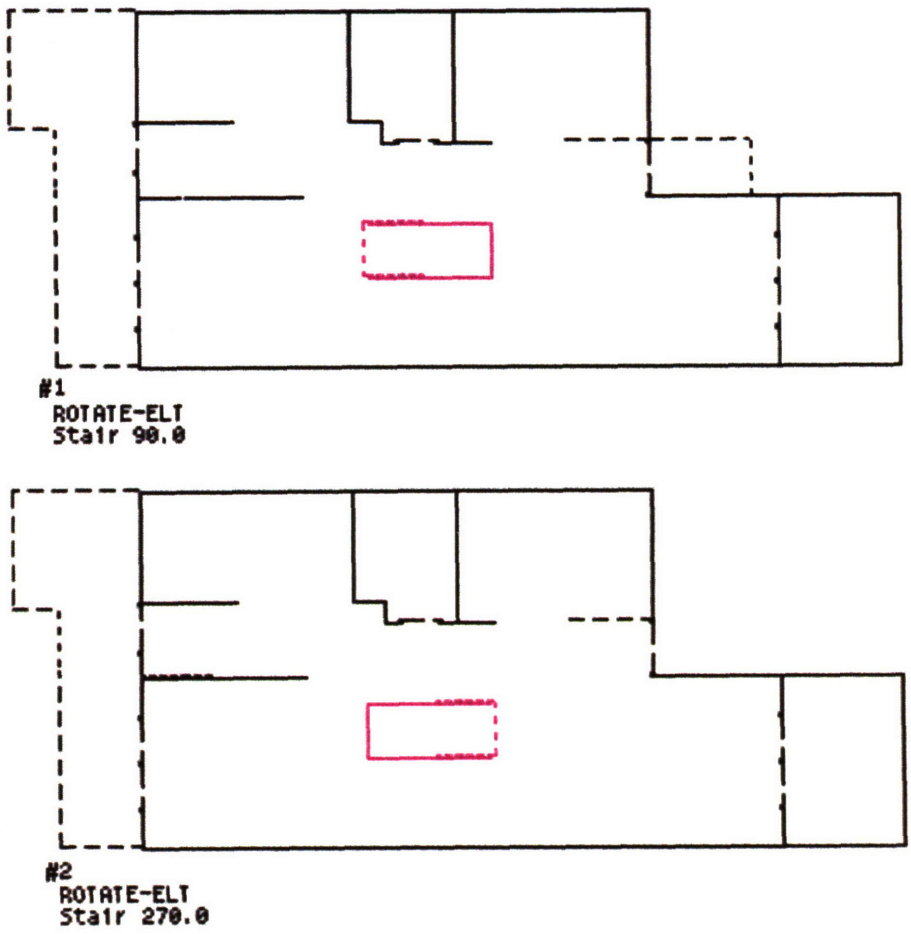



**Figure 1.10:** Two new designs: each has stair rotated to increase visual openness of Dining from Living.

Checking the visual openness of the Dining region in the new designs, TAC finds that more of the Dining region is now visible from the Living region so turning the stair had the intended effect. Figure 1.11 shows the visual openness measurements for one of the new designs, design #2, which is shown at the bottom of Figure 1.10.

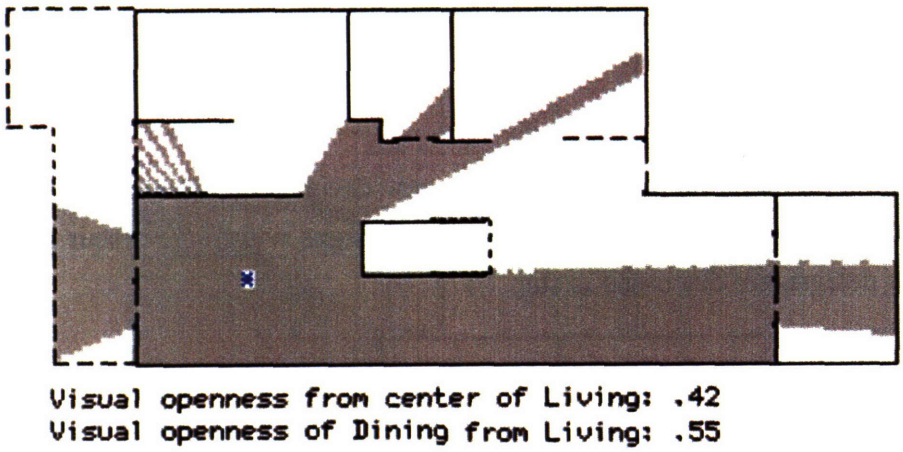



**Figure 1.11:** Visible areas and visual openness measures for design #2.

6. Evaluating all goals

The visual openness goal is satisfied for both new designs, so TAC checks that the other two goals, which were previously satisfied, are still satisfied in the designs.

Figure 1.12 shows one of the results of checking the second goal, which specifies that the visual openness of the Living and Dining region from the front door be less than 0.40. The portion of the Living region that is visible has not changed, so it is still acceptable. A larger portion of the Dining region is visible, but it is acceptable because it is less than the specified threshold. Thus, the second goal is still satisfied.

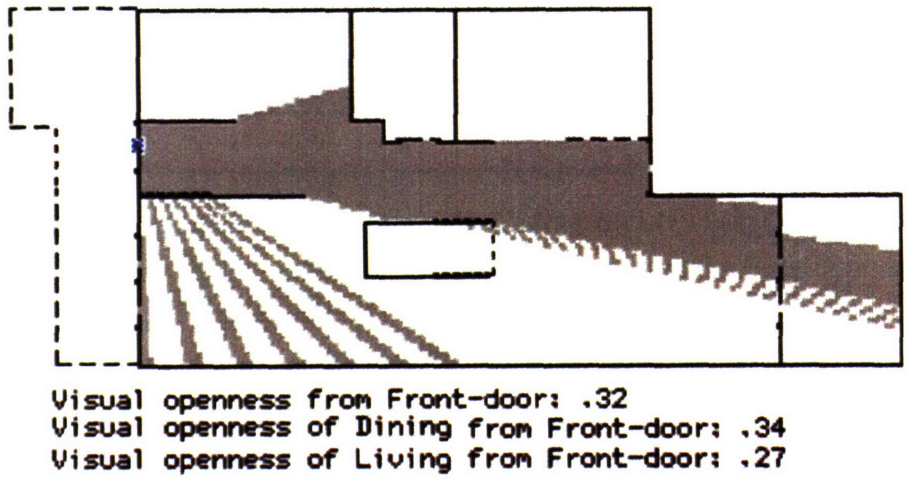



**Figure 1.12:** Visible areas and visual openness measures for front door in design #2.

Figure 1.13 shows one of the results of checking the third goal, which specifies that the stair be as "easily accessible" as in the starting design, i.e. that the values for path length and change of direction of the paths from the exterior doors be no larger than original values. In both new designs, the values have decreased.

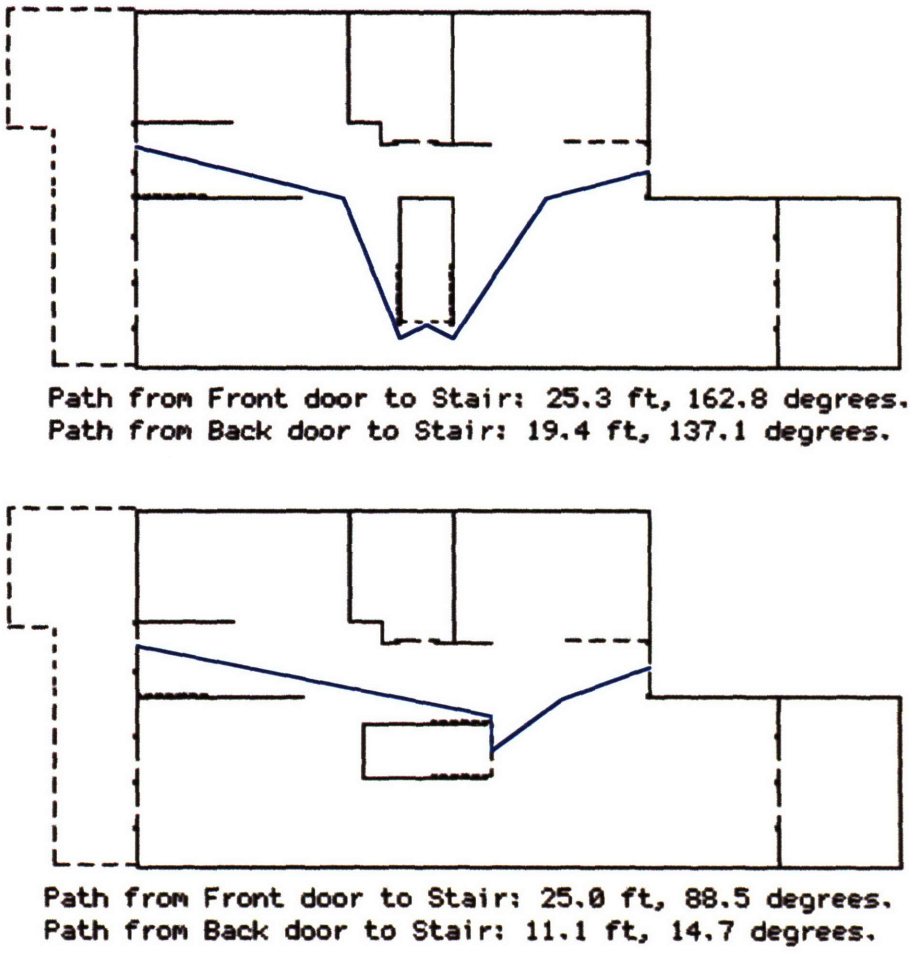



**Figure 1.13:** Comparison of stair access path lengths and changes in direction for starting design (top) and new design #2 (bottom).

All three goals—to increase visual openness between the Living and Dining regions to more than the current values; to keep the visual openness of the Living and Dining regions with respect to the front door less than 0.40 (i.e. so that less than 40% of each region is visible from the front door), and to keep the path length or change in direction between the stair and exterior doors less than or equal to original values—are satisfied for both new designs, so they are solutions.

7. Displaying rejected designs

The designer asks TAC to display new designs that were rejected. The designs, shown in Figure 1.14, have the stair moved to an exterior edge. The move increased the visual openness of the Dining region from the Living region. TAC rejected these designs, however, because they do not satisfy the other two criteria, namely to keep the Dining and Living regions relatively private with respect to the front door (by keeping the visual openness values less than 0.40), and to keep the stair easily accessible from the exterior doors (by keeping path length and changes in direction less than or equal to original values). The rejected designs have more of the Dining region visible from the front door, and have at least one stair access path that is either longer or has a larger change in direction than in the starting design.

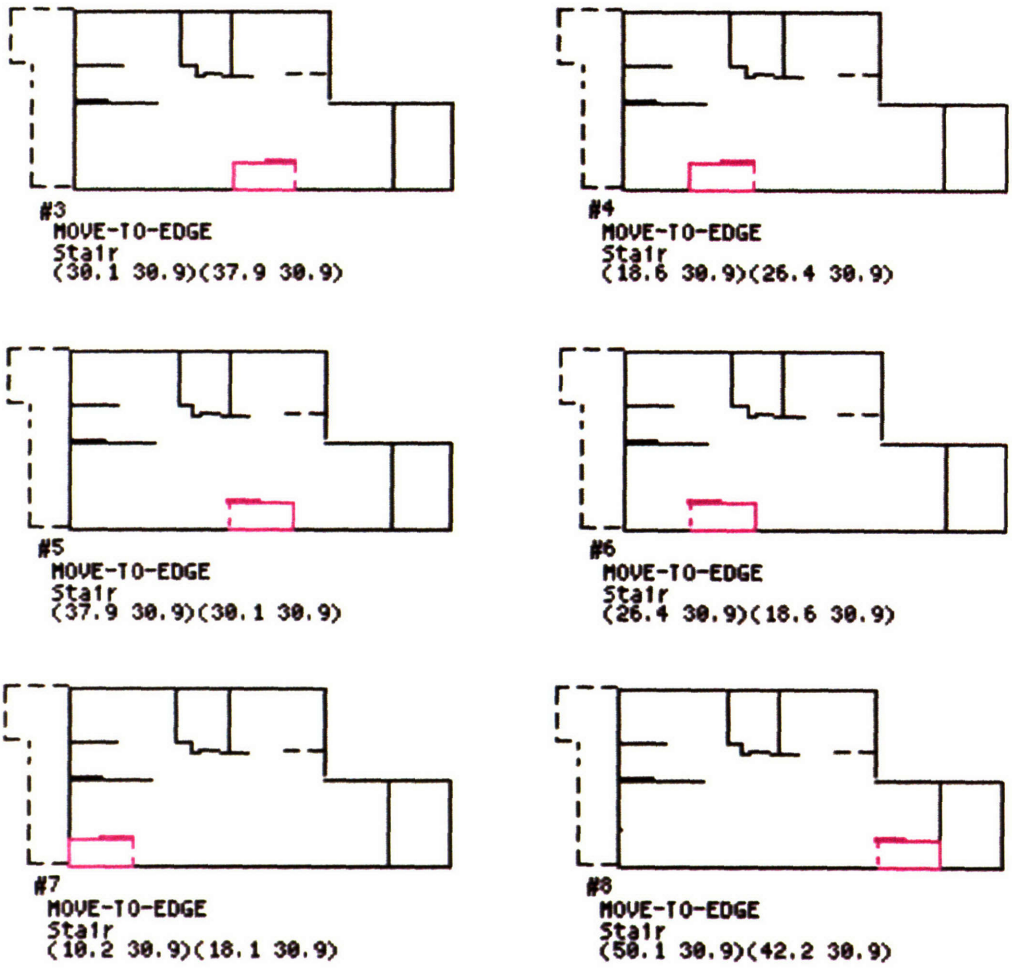



**Figure 1.14:** Rejected new designs with the stair on an exterior edge.[5]

5. When moving the stair in this example, TAC did not take into account other possible stair-related issues, e.g. access to the basement and second floor.

The above example gives a flavor of how TAC works. As will be shown in subsequent chapters, TAC's languages and representations enable it to translate design goals stated in terms of experiential qualities into operators on physical form. Its control structure, which employs what we've called dependency-directed repair, enables it to methodically and efficiently explore a large design space in search of solutions satisfying multiple goals.

TAC is notable as an artificial intelligence tool because of its knowledge, languages, representations, and dependency-directed redesign mechanism. It is notable as an architecture tool because it contributes to the clarification of terms used in architectural discourse, serves as a repository for reusable design knowledge, functions as both a design brainstorming tool and an analysis tool, and provides an example of how to distribute tasks between a designer and a computer assistant.

## 1.4 Motivation

TAC supports the early conceptual stage of design. Its domain is architecture, and its goal is to facilitate the design of physical form that manifests desired experiential qualities. Why focus on conceptual design? Why focus on architectural design and the relationship between physical form and human experience? Why is this problem interesting from an artificial intelligence point of view or from a designer's point of view?

**Why focus on a tool for conceptual design?**

Because there are very few tools for conceptual design.

In early stages of design, the language used is often very abstract. Architects and their clients use experiential terms such as "private" and "open". Engineers might talk about designing a piece of equipment that is "easy to maintain". Clothing designers talk of "baggy" clothing.

In addition, the specification of a design problem evolves along with solutions to the problem. It is rarely possible to specify apriori all design objectives and their relative priorities; some objectives may become apparent only after proposing and evaluating potential solutions (e.g. see Lawson, 1990; Akin, 1986; Schön, 1983). The designer simultaneously explores both the space of design problems and the space of design solutions.

In later stages, a designer performs more routine tasks such as resizing design elements, choosing particular design element types (e.g. kind of window) from a catalog, and positioning design elements in a well-understood portion of the design (e.g. placing refrigerator and stove in a kitchen). Computer-aided design (CAD) tools are available for these tasks. Most are drawing tools and are helpful in later stages, but require more precision than is available or appropriate for early

stages of design; they are awkward to use when a design problem is not yet well-defined. In addition, there is a danger that the precision required by these tools may cause an early drawing to be perceived as a design solution.

Other computer-aided design tools are intended to be analysis tools, but are often awkward and distracting to use. Current computer-aided design systems are inadequate as design tools, and particularly as conceptual design tools.

Krauss (2000):

> Presently, designers do not use computer-aided design systems in early stages of design largely because the present generation of design programs handle only a few, generally simple sets of variables and take too much effort to use for the value gained. For example, a common type of program is one that develops a space allocation diagram, derived from a designated set of spaces and a matrix that indicates the relative importance for proximity between each set of spaces. The list of spaces usually has to be constructed specifically for the routine, and the development of the matrix, which requires cramming a set of complex judgments into a crude and laborious format, is a step extraneous to the real design process. Hence, such programs are not in popular use.

Flemming(1994):

> Frustrations generated by working with commercial CAD systems that appeared simply too "dumb" to be of use in the most interesting phases of design provided a motivation [for becoming interested in AI]. In an attempt to be applicable over the broadest range of disciplines and applications, these systems restricted themselves (and still do) to the support of the most external manifestations of design processes, the preparation of drawings and geometric models, and to the purely syntactic aspects of these representations. They did not support synthesis and interpretation, processes that are highly exploratory in nature and demand fast response and active participation from a computer system.

**Why focus on architectural design and relating physical form to human experience?**

Architectural design is well-suited to our research for several reasons. Most design problems exhibit the difficulties outlined above—they are exploratory in nature; involve the use of abstract, experiential terms; and require construction of both problem specification and solution. The design problems generally involve many conflicting design goals—a good test for our system. In addition, architectural theorists have written extensively about various aspects of the design process, providing a good groundwork for computational studies.

Why focus on the relationship between physical form and human experience? Because it is

essential to architectural design and has not been addressed to date by computer-aided architectural design tools. Architects design spaces that people inhabit, focusing on both the physical form and how that form will affect the inhabitants. They and their clients describe spaces as private, sunny, open, spacious, inviting, etc. Architects use their knowledge from past experiences, from environment behavior research, and from their own theories to create spaces with experiential qualities such as these.[6] As discussed in the chapters that follow, this knowledge can be articulated and structured as general design principles (e.g. Wright, 1954; Moore, et al., 1974; Alexander, et al., 1977; Zeisel and Welch, 1981; Hertzberger, 1993) and used as a basis for a design support system that reasons about experiential qualities and physical form as in the opening scenario.

**Why is this problem interesting from an AI point of view?**

Design is inherently knowledge-based, intelligent behavior. It generally requires some degree of purposeful behavior—a designer has some number of things in mind that he wants to accomplish, then his task is to figure out how to build something that accomplishes them (while the description of what to accomplish constantly changes). When those things are experiential qualities, he must accomplish them indirectly through his choice of physical form. Classic AI issues have to be addressed in order to build computational tools to support this task. For example:

• Knowledge representation: How can abstract concepts such as "open" and "private" be operationalized? How can terms for such qualities be mapped to methods for accomplishing them? Can the same representations be used by the various components of a design tool?

• Reasoning and problem solving strategies: How can the search for design solutions avoid the combinatoric problems inherently associated with large search spaces? How are solutions found when evaluation functions are noninvertible and operators interact in complex ways? How are solutions found for multiple conflicting goals?

---

6. For examples of the relationship between experiential qualities and physical form see the journal *Environment and Behavior.* Also see Broadbent (1980) for a good survey of environmental psychology and a discussion of the intersection of psychology, sociology, anthropology, and architecture. See Hillier and Hanson (1984) for discussion of the relationship between sociology and architecture. See Rapoport (1969, 1977) for discussion of the relationship between anthropology and architecture.

**Why is this problem of interest from a designer's point of view?**

Clarification of terms in architecture is a long-standing need.

> John Louis Petit (1854)
>
> I am far from thinking that nomenclature is a remedy for every defect in art or science: still I cannot but feel that confusion of terms generally springs from, and always leads to, confusion of ideas.

By providing a language and a framework for representing the relationships between experiential qualities and physical form, TAC has the potential to help clarify architectural terms. Such clarification will enable better communication among designers and between designers and clients.

In addition, TAC can serve as an example of a design tool that alleviates some of the frustrations with existing computer-aided design tools. The comments from Krauss (1970) are still apropos:

> A number of computer operations such as cost checking, sorting of program issues, and space allocation have been developed to aid the architect in his work. None, however, have been put into common usage. One of the difficulties in adopting these techniques is that they are developed for solving problems in a far different context than for architectural design. In some cases the aiding routine may require definition of design criteria which is not possible to give in the way required; in others the procedure may be so awkward to the designer that little or nothing is gained; or the tools are not sufficiently useful by themselves, without a surrounding system of more useful computer aid to warrant use.

TAC aims to be a component in such a "surrounding system", providing an intelligent partner that facilitates design by providing a rich framework for representing designs and design knowledge, and by efficiently searching for solutions.

## 1.5 Guide to This Document

This document is organized around the steps involved in using TAC to find solutions to a design problem. Each step is illustrated using examples taken from the set of Frank Lloyd Wright's Prairie houses. A reader interested in what TAC does, rather than how TAC works, may want to focus on Chapters 1 and 8.

Chapter 1 introduces the problem and TAC's solution, gives an example of TAC's functionality, and discusses the motivation for building TAC.

Chapter 2 describes our approach to building TAC.

Chapter 3 describes TAC's representations for design problems, designs, characteristics, and goals.

Chapter 4 describes TAC's goal evaluation.

Chapter 5 describes how TAC proposes repair suggestions using explanations.

Chapter 6 describes how TAC carries out repair suggestions to create new designs.

Chapter 7 describes TAC's control structure and discusses related issues of goal interaction, goal order, and termination. It also presents the results of experiments run to test three different control structures.

Chapter 8 describes two exercises given TAC: a design example using a local house that was in the process of being redesigned, and an analysis example using Frank Lloyd Wright's Prairie houses.

Chapter 9 discusses related work.

Chapter 10 summarizes TAC's contributions and discusses future work.

# Chapter 2

# Approach

This chapter presents our view of the design process and our rationale for building a design assistant instead of an automatic design generation system. It also gives an overview of TAC's dependency-directed redesign strategy and discusses the sources of TAC's intelligence.

## 2.1 The Design Process

TAC adopts the view that architectural design is an exploratory search of a design space, trying to turn goals, often unarticulable at the beginning of the process, into physical form that realizes those goals.[1] Several factors contribute to the need for exploration of the search space, e.g. the complexity of simultaneously satisfying many potentially conflicting goals, the fact that the goals themselves evolve along with the design, and the absence of a fixed set of modification operators that result in new designs. As a result, it is very difficult to produce feasible solutions in one or a few steps. Instead, a designer engages in an iterative cycle of goal specification and design refinement: articulate design goals, produce a potential solution, evaluate it, modify it, and continue until a solution is found. More specifically, a designer proposes an initial design for a problem, perhaps sketching a new design or selecting a previous design that solved a similar problem. He then may modify the design in an attempt to satisfy articulated goals, employing a generate-and-test model. Alternatively, he may proceed in a more opportunistic fashion, noticing new goals as he works and modifying a design in accordance with those goals rather than focusing on previously articulated goals. In either of these situations, the designer is exploring a design space via an iterative evaluation and modification cycle that is guided by characteristics he desires in a solution and knowledge about how to achieve those characteristics.

---

1. Recorded observations of designers in action have led to this view of the design process. Rather than reiterate the observations here, the reader is referred to an in-depth study by Krauss and Meyer (1970). See Akin (1986) for a summary and reference to earlier studies (e.g. Eastman, 1969, 1970; Foz, 1973). See also Lawson (1994), Rowe (1987). See Schön (1983) for a competing theory that says designers "know it when they see it" and often cannot articulate goals. For additional theoretical views on the nature of the design process see Simon (1969, 1973, 1975), Jones (1970), Broadbent (1973), Wade (1977), Heath (1984), Cross (1991).

## 2.2 A Design Assistant

TAC functions as a design assistant because, even though much progress has been made in modeling design decision-making (e.g. Akin, 1986; Smithers, 1996), the design process is not yet understood well enough to build a fully automated system. We believe that an appropriate conceptual design tool acts as an assistant, complementing a human designer so that human and computer together work better, faster, and more easily than either working alone.

A human designer knows how to ascertain client needs, specify design goals, and adjust design goal sets and priorities, especially when clients' needs change; he is good at defining and extending a problem specification as he designs (e.g. Schön, 1983; Akin, 1986; Rowe, 1987; Lawson, 1990). He is good at generating and sketching initial designs, before design goals can be articulated. He knows how to shift his focus of attention between various aspects of a design. He knows how to dynamically add to his design knowledge, inventing new ways to satisfy design goals as he designs. He also knows how to judge when to stop, which is often not obvious because there is no single right answer.[2]

Computers have a valuable role to play as assistants. They can explore a design space methodically and can manage the complexity of many conflicting design goals. Humans have trouble dealing with such complexity due to limited capacity of short-term memory and the difficulty of deducing all consequences of particular design modifications. Computers are far less susceptible to such overload. Computers can provide languages and frameworks for helping a designer articulate and store design knowledge. They also can more easily and quickly generate and keep track of alternatives, and modify and redraw representations of designs.

TAC takes on the tasks computers do well, leaving the tasks humans do well to the designer.

## 2.3 TAC as Design Assistant

As noted earlier, current CAD tools do not support exploration, and as a result are often extraneous to the design process. They also have simple design vocabularies and require too much precision. TAC assists the designer with all but one of these problems. TAC supports exploration by focusing on the evaluation and repair cycle that is central to design, thereby being an integral part of the process rather than extraneous to it. It assists a designer in specifying design goals, evaluating a design with respect to those goals, and repairing the design if necessary. In the process, it generates alternatives easily and quickly, helping to manage the complexity of satisfying multiple goals in a domain in which modification operators interact and evaluation functions are not easily

2. See discussion of satisficing in Simon (1969).

inverted. In addition, its broad vocabulary is that of the designer, and as such, includes terms for experiential qualities and elements of physical form. TAC does not deal with the precision issue that hampers CAD drawing tools.

There are aspects of the design process that TAC does not address. It leaves the task of generating an initial design to the designer. It also leaves the task of design goal specification and respecification to the designer, focusing on design refinement rather than goal refinement. Finally, it generates multiple solutions when they exist, leaving the task of ranking the solutions to the designer.

### 2.3.1 Dependency-Directed Redesign

TAC's dependency-directed redesign strategy embodies a variation of generate-and-test (or more accurately, test-and-generate): given an initial design and a set of goals, TAC evaluates a design with respect to the goals, then generates new designs by proposing, refining, and carrying out repair suggestions (i.e. design modifications). The generate step, which we call repair, is dependency-directed, guided by knowledge of two kinds of dependencies: general dependencies between experiential qualities and physical form, which include methods for realizing qualities in form; and specific dependencies for a particular design, which serve as explanations for unsatisfied goals. If a design does not satisfy all goals, TAC uses the explanations to search its knowledge base for design modification operators likely to repair the design so that all goals become satisfied.

Three aspects of the architecture domain motivated our choice of dependency-directed repair as a control structure: some evaluation functions cannot be easily inverted to set the values of design characteristics, the effects of operators are difficult to predict, and multiple interacting goals cannot be easily satisfied in one or a few steps.

1. The values of some design characteristics cannot be set.

Some design modification operators directly set the values of design characteristics. Changing the color of a door in a design, for example, is straightforward because color is a directly settable design attribute. Many design characteristics in this domain—and all that represent experiential qualities—are not directly settable, however. Instead, their values are changed by modifying the physical form. If we want half of the living room visible from the dining room, for example, we might turn an intervening stair as in the opening scenario. More of the living room may indeed be visible, but it is difficult to guarantee that the visibility value will be 0.5. Visibility is an example of a characteristic measured via a computational geometry routine whose result depends crucially on the specific arrangement of design elements, and that is therefore not easily inverted. For such

a characteristic it is easier to carry out an operator and measure the characteristic in the resulting new design, i.e. to employ TAC's version of generate and test.

2. The effects of operators are difficult to predict.

Effects of operators are difficult to predict because an operator may not have its intended effect or may have unintended effects, and because operators may interact.

An operator has an intended effect that is stated in terms of a desired value for a characteristic of a design, e.g. increasing the number of fireplaces in a design. When design characteristics can be measured independently of context, as with counting the number of fireplaces, the effects of operators that modify those characteristics can be predicted. Adding a fireplace always increases the number of fireplaces in a design regardless of the specific geometry of the design. Other characteristics depend on the particular arrangement of design elements, and the effects of operators intended to achieve those characteristics cannot be predicted. To make all of one space visible from another, for example, a wall might be moved. For some arrangements of design elements, moving the wall may satisfy the visibility goal. In others, an intervening design element (e.g. a stair) may keep the move operator from having its intended effect on visibility. Attempting to predict whether the goal will be satisfied requires just as much work as performing the move operation and checking the design.

An operator also may have unintended effects. Moving the wall may increase visibility between two spaces as intended, but may block the view between two other spaces. It is very difficult at best, and impossible at worst, to precompute all effects of operators; it is easier to carry out an operator, then check the design to determine which aspects of the design have been affected.

Finally, the effects of combinations of operators can be more difficult to predict than effects of single operators. Operators can interact in complicated ways: one operator may move a fireplace to a particular location, displacing a bookcase; a second operator may move the bookcase back to its original location, moving the fireplace to yet another location; a third operator may move the bookcase yet again. Because each move causes the design to be slightly different, new locations may become apparent after performing the move. As a result, it is difficult to predict where each design element will end up, much less what the effect will be on characteristics such as visibility. It is easier to determine the combined effect of a sequence of operators by carrying out the operators.

Each of these examples has illustrated that a generate and test strategy is appropriate when the effects of operators are unpredictable.

3.Multiple interacting goals cannot be easily satisfied in one or a few steps.

As previously mentioned, there is no obvious path from problem definition to solution, especially with interacting design goals and operators whose effects are not completely predictable. It is possible, however, to generate a design that satisfies a subset of the design goals. Such designs, even if not solutions, are often useful intermediate steps from which progress toward a solution can be made. An enabling assumption for this approach is that some goals will be independent, so that working on one goal does not always cause a previously satisfied goal to become unsatisfied. For the goals that do interact, some amount of work to reevaluate and resatisfy goals is necessary, hence the iteration between evaluation and repair.

The difficulties of reasoning with noninvertible evaluation functions, predicting operator effects, and of satisfying multiple design goals in this domain lead to the choice of an iterative control structure that focuses on evaluation and repair. TAC proposes and refines suggestions for modifying a design, then carries out the suggestions and evaluates the resulting new designs to see if intended goals are satisfied. If some of the goals are now satisfied for a new design, TAC assumes that it is making progress and attempts to repair the design by starting a new evaluation and repair cycle.

### 2.3.2 Intelligence

TAC's intelligence derives from the richness of its knowledge base and representations, and its informed search of repair suggestion space.

#### Knowledge

TAC's knowledge consists of both general knowledge and domain knowledge. The general knowledge is used by TAC's reasoners to explain their reasoning, do simple arithmetic, compare and rank vectors using a partial order, deduce part-whole relationships, and manipulate nodes and arcs in a graph.

The domain knowledge is a major source of TAC's power as an architectural design assistant. It includes general design principles as well as specific methods for modifying designs. It is organized around types of design elements, which represent physical form such as walls and doors; and design characteristics, which represent properties of a design and range from very concrete details such as square footage to very abstract qualities such as openness or privacy. Much of TAC's intelligence derives from its mapping of experiential qualities to details of physical form.

TAC knows, for example, that it can make one space more visible from another by removing intervening walls. It knows that it can make a space feel more private by making less of it visible or making the path to it from a front door less direct.

TAC also knows about characteristics of Frank Lloyd Wright's Prairie houses. It knows, for example, that Prairie houses often have large main living spaces containing a fireplace that symbolizes home and hearth. (See Section 8.2 for discussion.) Such knowledge serves to illustrate that TAC can represent not only general architectural knowledge, but a designer's particular preferences as well.

### Representation

TAC's representation languages enable close coupling between TAC's components. The results of one component are used as input for the next: evaluation results in the form of explanations are the input to the repair suggestion mechanism; repair suggestions are the input to design creation. TAC's languages enable this smooth integration by sharing a common syntax and by being expressive, concise, and unambiguous.

TAC's representation of designs supports evaluation of a wide variety of design characteristics. Some of the characteristics are concerned with the geometry of design elements (e.g their sizes and location); others are concerned with the boundaries of spaces or circulation paths in a design. TAC handles the requirements for different information by representing a design as a set of models, each representing a different aspect of a design. It then chooses the model appropriate for a particular task.

### Reasoning and Search

Exhaustive search of the design space is not an option in this domain; there are far too many possible designs. TAC limits its search by first searching repair suggestion space and pruning that space, so that it proposes only repairs that have a good chance of leading to solutions. It carries out the repair suggestions, modifying the original design to create new designs that it thinks are likely to satisfy at least some of the specified goals. It then attempts to repair any of the new designs that are not solutions.

TAC's repair suggestion mechanism identifies relevant suggestions by comparing desired values of design characteristics with actual values. It figures out, for example, that if a characteristic's actual value is less than desired, it must increase the value. It checks its knowledge base for modifications that are likely to increase the characteristic's value, then checks the particular design to determine which of the modifications are relevant to the current context. It proposes and carries out the selected modifications, then checks each resulting new design to make sure the modification had the intended effect.

In summary, TAC is an intelligent design assistant that employs a strategy we've called dependency-directed redesign which enables it to evaluate designs with respect to a set of design goals, propose and refine repair suggestions, then carry out those suggestions to create new designs that satisfy the goals.

# Chapter 3

# Defining a Design Problem

TAC is organized around the notion of a design problem, by which we mean a design and a set of design goals. Let's say that we have a design and a set of design goals of the sort mentioned in the opening scenario, e.g. we want living spaces that feel "open" to one another, but "private" with respect to the front door, and we want the stair to be "easily accessible". We also might want the design to have four bedrooms. It's fairly easy to translate having four bedrooms into a particular design; it's much harder to translate qualities such as open, private, and accessible. The mapping between these qualities and physical form is not straightforward. It is possible, however, to represent these qualities in such a way that they can be measured and reasoned about, and thus used to create designs that satisfy desired goals.

## 3.1 Representing Designs

A design is represented by five kinds of models—a design element model, an edge model, a territory model, a use space model, and a circulation model. Each of the models abstracts different design details, enabling easier, faster access to information relevant to a particular task.

The *design element model* contains information about design elements, i.e. the objects that create physical form: walls, windows, doorways, fireplaces, etc. Included is information about size, location, materials, and component parts.

The *edge model* is a two-dimensional geometric abstraction of the design element model and contains points and nonoverlapping edges. Each edge in the edge model is the result of applying one or more abstraction methods to particular design elements. The information summarizing the origins of an edge is called the edge's derivation, and as described later, is used by TAC when suggesting repairs and creating new designs. There are three edge derivation methods: an edge may be a one-dimensional abstraction of a design element, such as a wall; a member of a two-dimensional footprint (i.e. outline) of a design element, such as a fireplace; or a one-dimensional projection of a design element. A projection, also called a projected edge, does not correspond to a physical object, but is an extension of edges derived from design elements. As shown in Figure 3.1, projected edges extend in either parallel or orthogonal directions from a design element.

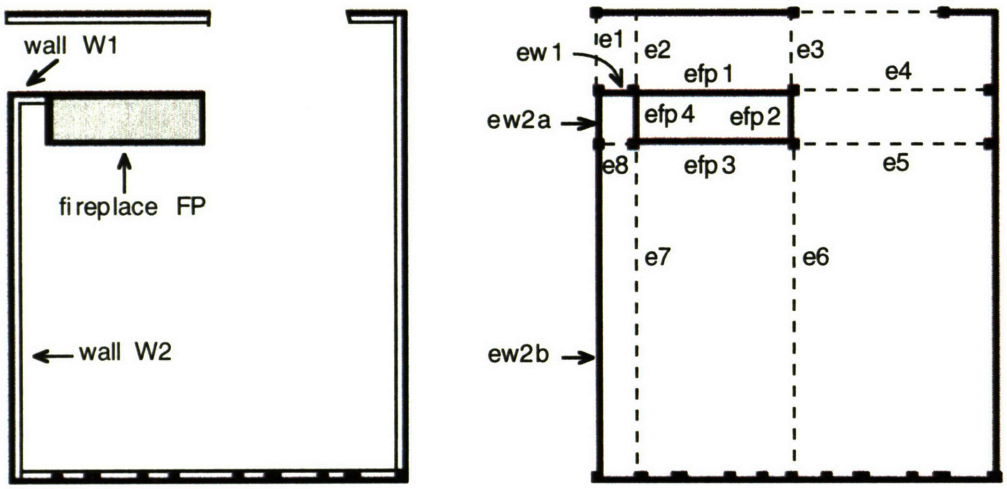



**Figure 3.1:** Partial design element model (left) and derived edge model (right).

Projected edges, shown dotted, end at intersection with design elements, shown solid.

Edge derivations:

ew1 is derived from wall W1 via abstraction.

ew2a, ew2b are derived from wall W2 via abstraction.

efp1 - efp4 are derived from fireplace FP via footprint.

e1 - e8 are projected edges with the following derivations:

e1 from W1 via orthogonal projection, W2 via parallel projection

e2 from W1 via orthogonal projection and from FP via orthogonal and parallel projections

e3-e8 from FP via orthogonal and parallel projections

Projected edges help bound territories—regions of space defined by design elements. TAC's territories are bounded, i.e. closed polygons, though they may overlap, and larger territories may be formed by unioning smaller ones. (For discussion of territories, see Alexander, et al., 1977; Hertzberger, 1993. For discussion of territories and their formalization, see Kincaid, 1997.) Territories are represented as ordered sets of edges and grouped into a *territory model*, another geometric abstraction of the design element model. A design model may have several territory models; it has only one edge model.[1]

Each territory model may be associated with one or more *use space models*. A use space model contains a set of use spaces, each of which is a territory plus an activity label, e.g. "dining". The activity label represents the intended use of the territory.[2]

---

1. For clarity of exposition, designs in this document will have one territory model, and territories will be room-sized.

2. In this document, territories will be named using terms such as Living and Dining, rather than, e.g. T13799. The names, however, do not carry functional information; use spaces represent that information. An example of reasoning with use spaces is presented in Section 8.1.

Finally, *a circulation model* is a graph with nodes representing doorways and arcs representing paths between doorway midpoints. The nodes and arcs contain dimension and location information.

The figure below summarizes the relationships between the five kinds of models. TAC's representation for a design differs from other work in the field in three significant ways: (1) Design elements are the fundamental units of design; they are manipulated in order to satisfy design goals. (2) Territories are derived from the design elements instead of being supplied apriori. (3) Territories and their intended uses are represented separately. Focusing on design elements and deriving territories from them is paramount when reasoning about physical form. Other systems either represent only design elements or only spaces. Neither of these representations enables translation of abstract design goals into physical form. Finally, representing use spaces and territories separately enables TAC to reason about physical form independently of intended use.

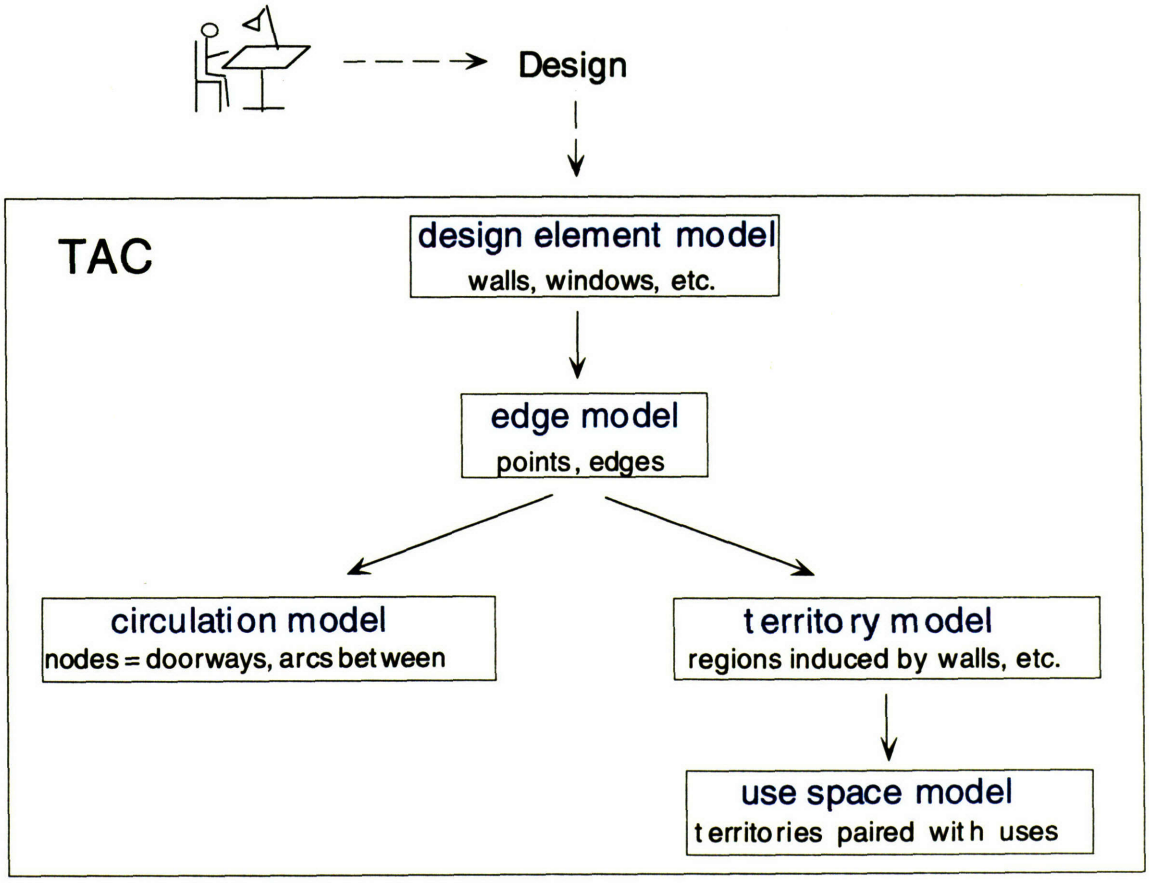



**Figure 3.2:** TAC's input is in the form of a design element model and an edge model.[3] Circulation and territory models are automatically derived; a use space model is defined with input from the user.

3. Design element and edge models currently are entered by hand using a 2D design editor. In future versions, they will be derived automatically from an annotated sketch (Gross, 1996).

Figure 3.3 shows the Mrs. Thomas Gale house, a Prairie house designed by Frank Lloyd Wright. Figure 3.4 shows diagrams of models for the first floor of this house.

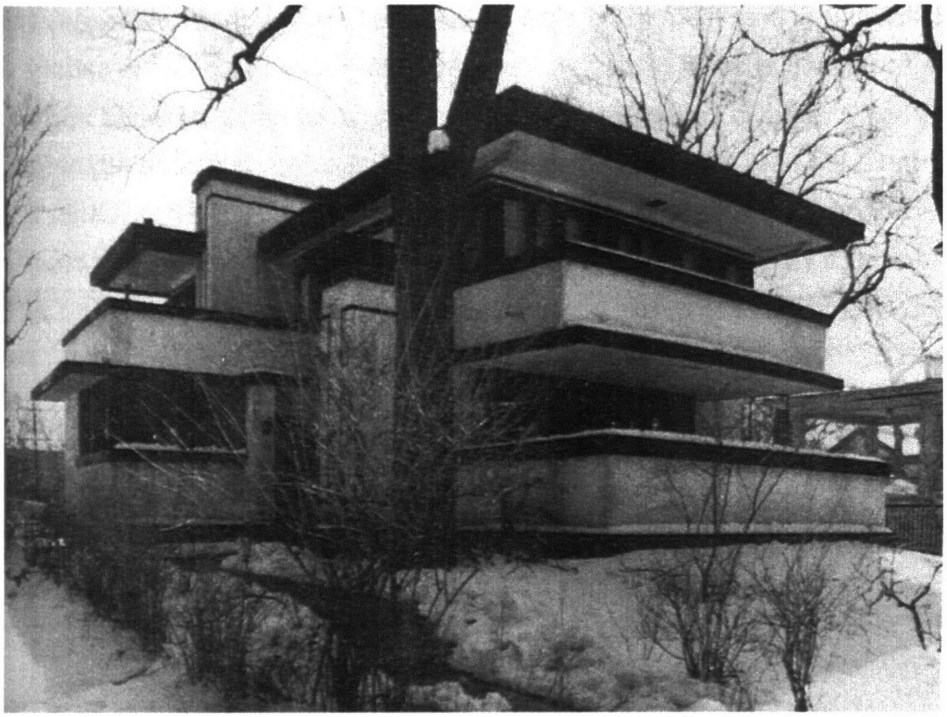
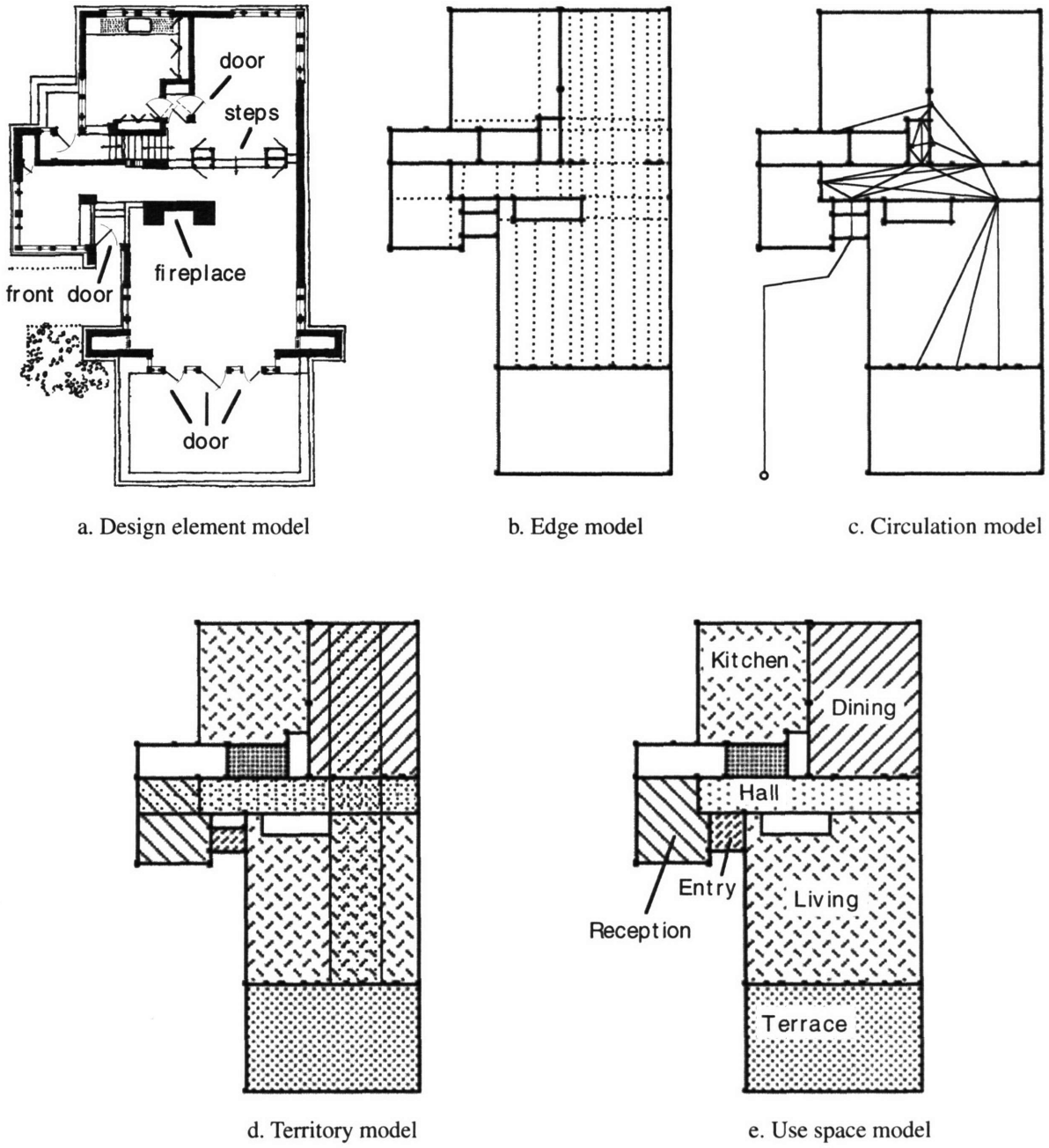


**Figure 3.3:** Mrs. Thomas Gale house, Oak Park, Illinois (1904, 1909)




a. Design element model

b. Edge model

c. Circulation model

d. Territory model

e. Use space model

**Figure 3.4:** Models for the Mrs. Thomas Gale house.

a. Floorplan showing several design elements in the design element model.

b. Edge model: contains points and edges derived from design elements.

c. Circulation model: shows paths from exterior approach point through interior.

d. Territory model: territories formed by design elements; overlapping territories shown.

e. Use space model: use spaces, which are territories plus activity labels.

## 3.2 Representing Design Goals

In addition to a design, a design problem also contains *design goals*, which specify desired features for a design. Design goals are represented as an expression and a desired value for the expression. If we'd like a Living territory to be visually open from a Dining territory, for example, we specify the expression `(visually-open Living from Dining)` and a desired value of `true`. If we'd like one fireplace in a Living territory, we specify the expression `(fireplace-count in Living)` and a desired value of 1. Both `visually-open` and `fireplace-count` are examples of *design characteristics*, constructs that represent TAC's architectural knowledge. TAC also contains domain-independent knowledge from geometry, arithmetic, logic, and computation. It represents this knowledge using *TAC-functions*. Design characteristics and TAC-functions are TAC's main source of knowledge and form the core of the languages used throughout TAC—by the designer to define other design characteristics and to specify goals, and by TAC to construct explanations for goal evaluations and to propose repair suggestions.

### 3.2.1 Design Characteristics

#### What is a design characteristic?

Design characteristics represent architectural properties of a design; they are concepts such as visual openness, physical accessibility, square footage. They are quantitative-, qualitative-, boolean-, set-, or vector-valued. Some design characteristics, such as square footage, can be computed directly from design elements, which represent physical form. Other design characteristics, such as those that represent experiential qualities, are derived from the computed design characteristics and are related to physical form via those characteristics. The decision about which design characteristics are calculated directly from a design and which are derived from other design characteristics is made by the designer.

Design characteristics form a decomposition hierarchy, with characteristics computed from physical form at the bottom and those derived from them higher up. In this way experiential qualities such as privacy are mapped into details of physical form. The following design characteristics, which appear in examples throughout this document, illustrate this mapping. The decomposition hierarchy for these characteristics is shown at the end of this section.

Example 1: `Visual-openness` is quantitative-valued and measures the portion of a territory visible from another territory.[4] It produces a value between 0 and 1.0. In the Gale house, for example, the `visual-openness` of the Living territory from the Dining territory is 0.78. Figure 3.5 shows the visible portion of the Living territory. Figure 3.6 shows the view from the Living territory toward the Dining territory.

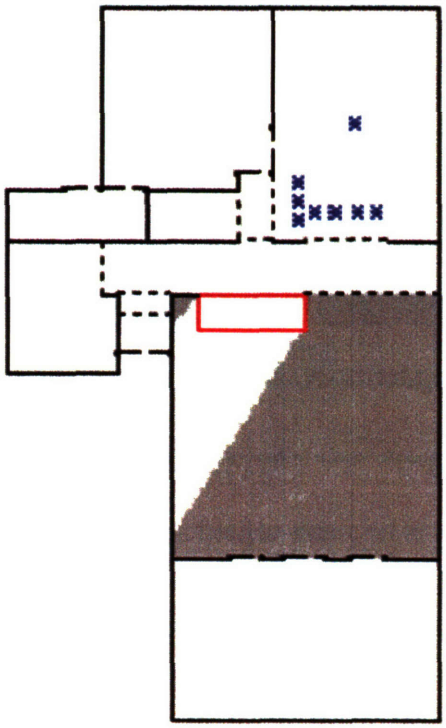


**Figure 3.5:** Shaded region of Gale Living territory is visible from Dining territory; `visual-openness` value of Living from Dining is 0.78. Each * represents a viewpoint used by the visibility calculation.

4. The visual openness routine works from the 2D territory model: it tiles a territory, figures out which tiles are visible from specified viewpoints, then calculates the ratio of visible tiles to total number of tiles. Tile size and viewpoint placement are user-controlled parameters. Defaults: 6" square tiles; viewpoints at territory center and along relevant openings, 2.0 feet inside the territory and 1.5 feet apart.

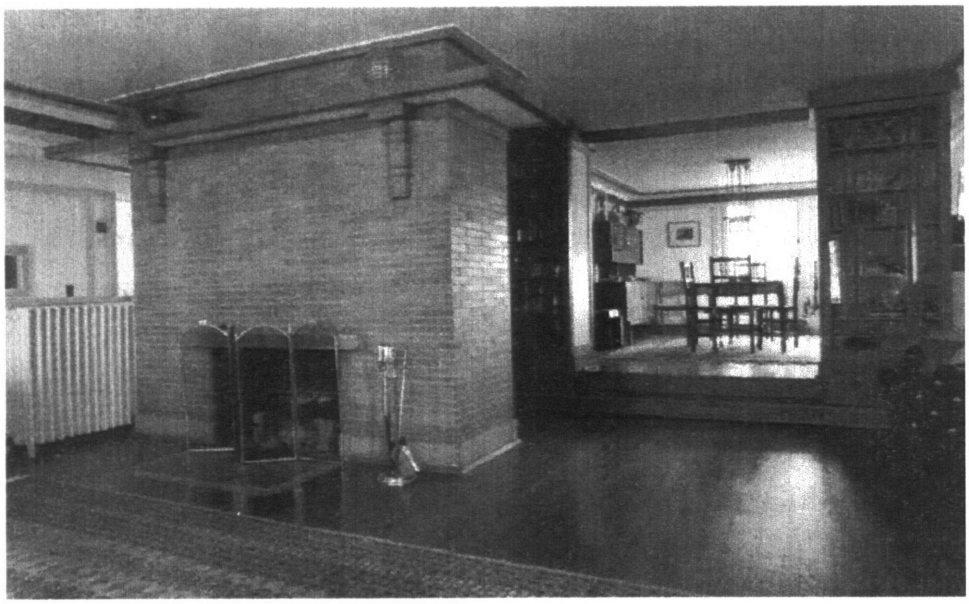


**Figure 3.6:** Living territory of Gale house; view to Dining territory.

Example 2: `Visually-open` is boolean-valued and defined in terms of `visual-openness` by putting a threshold on `visual-openness`: A territory is considered `visually-open` from another territory if at least 0.6 of its area is visible from the other territory. The Living territory in the Gale house is considered `visually-open` from the Dining territory because its `visual-openness` value is 0.78.

Example 3: `Privacy` is vector-valued, with components for `visual-openness` and `physical-accessibility`. `Visual-openness` is described above. `Physical-accessibility` is vector-valued, with components for distance between two design objects and change in direction along the shortest path between two design objects[5]. Each of these components is quantitative-valued. The distance is calculated from the locations of the two design objects, the change in direction is calculated from a path derived from the physical form. (See Figure 3.7 for an example.) The design characteristic `privacy` thus has components for visual openness, distance between two design objects, and change in direction along a path between two design objects. We could define a boolean-valued design characteristic called `private` by putting thresholds on some or all of these. Notice, however, that the components for `privacy` are incommensurate, and it's not necessarily meaningful, nor obvious how to combine them into a single boolean-valued design

5. We use the term "design object" to mean a design element (e.g. wall), territory, use space, or design.

characteristic representing the concept of being private. We'll see later that even without defining boolean-valued characteristics for vector-valued characteristics, TAC can rank values for vector-valued characteristics and reason about how to increase or decrease them by using a partial order.

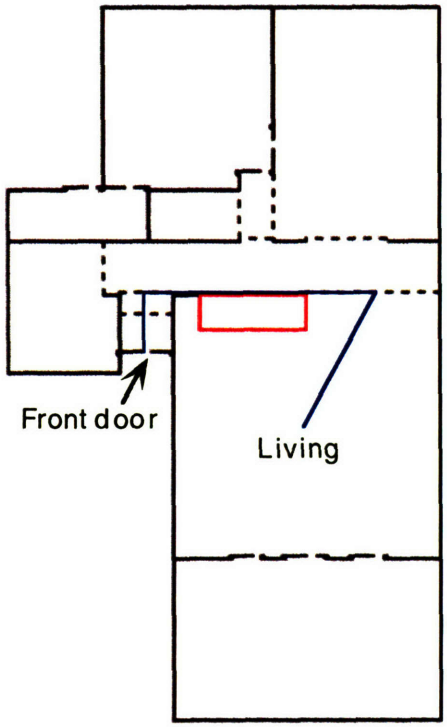



**Figure 3.7:** Path for calculating change in direction from Gale Front door to Living territory. Change in direction is 207.0 degrees; path was found using the circulation model.

Example 4: `Perceived-main-entryness` is vector-valued and gives a measure of the perception of an exterior door as a main entry. Characteristics that influence a visitor's choice of door when approaching a house are components of `perceived-main-entryness`. These include whether the door is visible from the street, whether there is a path that leads to the door from the street and how straight that path is, how far the door is from the street, and how formal the door is. We define a door as formal if it is of solid wood instead of glass and/or is hinged rather than sliding.

Example 5: The design characteristic `perceived-main-entry` is defined in terms of `perceived-main-entryness`. Its value is the exterior door that is most likely to be perceived as the main entry, i.e. the exterior door with the largest `perceived-main-entryness` value. TAC constructs a partial order that ranks exterior doors by their `perceived-main-entryness` values,

and returns the top of the partial order as the value of `perceived-main-entry`. (TAC's construction and use of a partial order is discussed in more detail later.)

**How are design characteristics defined?**

As the above examples have illustrated, design characteristics are related to physical form and often to other design characteristics. A characteristic is directly related to physical form by means of an opaque evaluation function that operates on one or several of the models representing a design. The characteristic `visual-openness`, for example, is calculated via a computational geometry routine that operates on a design's territory model. As noted earlier, characteristics directly related to physical form are at the bottom of a decomposition hierarchy, with characteristics derived from them higher up. The decomposition hierarchy is implemented via three kinds of dependencies each of which is illustrated below.

Example 1: The design characteristic `visually-open` is related to `visual-openness` by providing it with an *evaluation function body* that checks whether a `visual-openness` value is greater than 0.6: `(gt (visual-openness x from y) 0.6)`. The function body is written using a Lisp-like functional language which we've called the design characteristic definition language. The language's terms are the names of design characteristics and TAC-functions, variable names, and constants. The names of design characteristics or TAC-functions occupy the first (functional) position of expressions.[6] (See Appendix B for a list of design characteristics and TAC functions, and Appendix C for lists of language terms.)

Example 2: The design characteristic `privacy` has two *components*, `visual-openness` and `physical-accessibility`. The design characteristic `physical-accessibility`, in turn, has two components, `distance-btw` and `change-in-direction-btw`. Each of these components has an opaque evaluation function. The evaluation function for `privacy` collects all components into a vector of design characteristic expressions: `((visual-openness x from y) (distance-btw x and y) (change-in-direction-btw x and y))`. It evaluates each of those expressions with respect to supplied arguments, examining design element, edge, territory, and circulation models in the process, and returns a vector consisting of the visual openness between the two arguments, distance between the two arguments, and change in direction between the two arguments.

6. Throughout this document, we present TAC expressions using a more English-like syntax than is implemented. Examples of TAC's expression syntax can be found in Appendix B.

Example 3: A design characteristic may have *necessary conditions*, which turn the body of the characteristic's evaluation function into an `if` statement:

`if` necessary conditions are true

`then` compute a value

`else` return the symbol `no-value`.

The design characteristic `perceived-main-entryness`, for example, has components representing physical accessibility and formality of a door. The characteristic also has two necessary conditions: for an exterior door to be perceived as a main entry, the door must be visible from and have a path from the usual approach point.[7] The defined domain of a design characteristic provides an additional implicit necessary condition: only exterior doors have `perceived-main-entryness` values. The components and necessary conditions for `perceived-main-entryness` are represented using expressions in TAC's design characteristic definition language:

```
perceived-main-entryness (x)
  • necessary conditions ((built-exterior-paths to x from usual-approach)
                          (visible-from x usual-approach))
  • components           ((physical-accessibility of x from usual-approach)
                          (formality-of-entry x))
```

The design characteristics in the above definition are as follows: `built-exterior-paths` returns a set of exterior paths between two objects; `visible-from` determines whether one object is visible from a second; `physical-accessibility`, discussed earlier, returns values for distance between and change in direction along a path between two objects; `formality-of-entry` returns a vector containing values for door `solidity`, which represents the portion of a door that is wood, and `degree-of-hinge`, which represents door hinge type. The design characteristic, `degree-of-hinge`, is an example of a qualitative-valued design characteristic whose range is an ordered set of values. A hinged door has a higher `degree-of-hinge` value than a sliding door, for example. Relating door hinge type to formality, we say that a hinged door is more formal than a sliding door.[8]

The value returned for the design characteristic `perceived-main-entryness` is thus either the vector of values `((distance-btw x and usual-approach) (change-in-direction-btw x and usual-approach) (solidity of x) (degree-of-hinge of x))` or the symbol `no-value`.

---

7. The usual approach point represents the primary location from which someone would approach a building. A building may have several approach points; one of them may be deemed the usual one.

8. Calling this design characteristic "degree-of-hinge" is a bit awkward; the important point is that its values can be ranked.

As noted above, design characteristics form a decomposition hierarchy. This hierarchy is established via dependencies derived from evaluation function bodies, components, and necessary conditions. Shown below are dependencies for the above examples. Characteristics at the bottom of the hierarchy, i.e. with no characteristics below them, are computed directly from design objects.

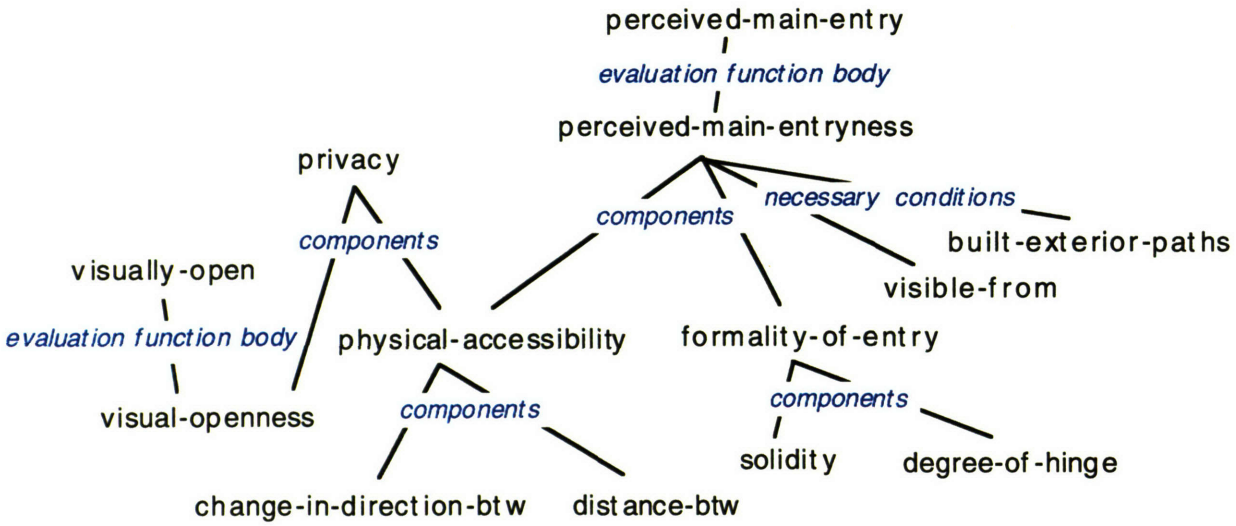



**Figure 3.8:** Dependency links for example design characteristics.

### 3.2.2 TAC-Functions

TAC-functions extend the expressiveness of TAC's languages—design characteristic definition language, goal specification language, and repair suggestion language—and are used in the same way that design characteristics are: in functional expressions and as a repository for knowledge about how to construct explanations and suggest repairs. As previously mentioned, design characteristics represent architectural concepts. TAC-functions represent domain independent concepts such as geometric concepts, arithmetic relations, logical relations, and computational constructs.

Example 1: `gt`

The TAC-function `gt` was shown previously in the evaluation function body for the design characteristic `visually-open`: `(gt (visual-openness of x from y) 0.6)`. The TAC-function `gt` is boolean-valued and generalizes mathematical greater-than by working not just on numbers but also on a number and the symbol `no-value`. Any value is considered greater than `no-value`.

Example 2: `more-of`

The TAC-function `more-of` further generalizes mathematical greater-than by providing a comparison function for members of vectors and ordered sets. The function might be used, for example, to express a desired relationship between two `perceived-main-entryness` values:

```
(more-of (perceived-main-entryness of Front-door)
         than (perceived-main-entryness of Side-door))
```

Vector values such as `perceived-main-entryness` are compared by comparing their corresponding components: A vector is greater than another vector if all components are greater than or equal to the corresponding components in the other vector, and at least one component is strictly greater. The semantics of the vector must be known: the semantics of each component determines the appropriate component comparison function. A `perceived-main-entryness` value is more than another one, for example, if its first two components, which represent physical accessibility, are smaller; and its last two components, which represent formality, are larger. In other words, an exterior door has more `perceived-main-entryness` than another if it is closer to the usual approach, reached through a straighter path, is composed of more wood, and is hinged (e.g. as opposed to sliding). We'll see later how TAC represents the knowledge that enables it to automatically define a `more-of` function for each vector-valued design characteristic.

Values that are members of ordered sets are compared via `more-of` by comparing their positions in the set: a value is greater than another if it is later (or earlier) in the set. As with vectors, the semantics of the ordered set must be known. The value `hinged` is considered more than `sliding` when considering the design characteristic `degree-of-hinge`, for example.

# 3.3 Design Problem Example

We define a design problem by specifying a design and a set of design goals. Given the design characteristic `visually-open` introduced earlier, for example, we can define a design problem by specifying the Gale design and the design goals:

```
<design-problem: Gale-1>
  design:  <design: Gale>
  goals:
    (<goal: (visually-open Living from Dining) true>
     <goal: (visually-open Living from Kitchen) false>
     <goal: (visually-open Living from Front-door) false>)[9]
```

9. The printed representation for a goal shows the expression immediately following the term "goal"; the desired value follows the expression. The printed representation for a design object is its name.

# Chapter 4

# Evaluating Design Goals

A design goal is evaluated by evaluating its expression; a value and an explanation are returned. TAC determines whether or not a goal is satisfied by comparing this returned value with the desired value stored in the goal. The following sections give examples of evaluating design goals and describe explanations and how they are generated. Chapter 5 discusses the use of explanations as the starting point for TAC's repair suggestion mechanism.

## 4.1 Examples

Example 1: `visually-open`

Below is a territory model for the main floor of the Tomek house, a Prairie house designed and built by Frank Lloyd Wright in 1904. (See Appendix D for an alternate territory model containing territories smaller than room-size.)

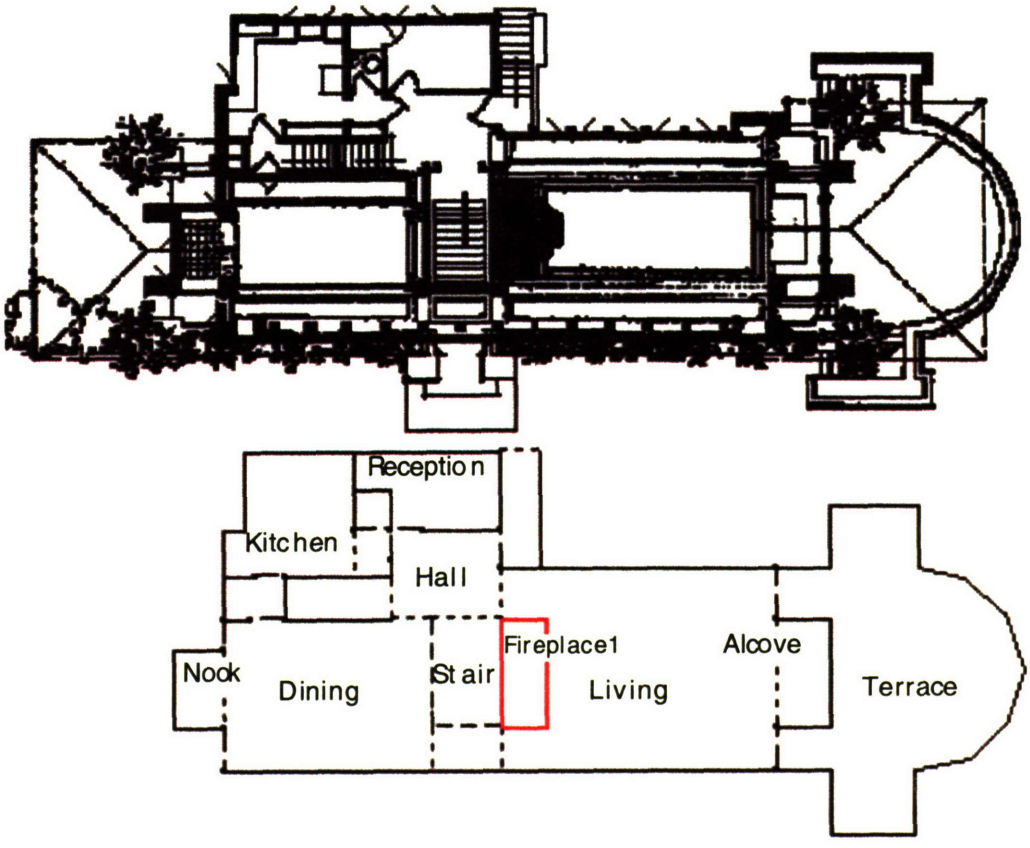



**Figure 4.1:** Floorplan and territory model for Tomek house main (second) floor.

Let's say that we want the Living territory to be visually open from the Dining territory. We specify the goal: `<goal: (visually-open Living from Dining) true>.`

Evaluating the goal, TAC returns:

goal satisfied? `no`
desired value: `true`
actual value: `false`
explanation for actual value: `<expl: (visually-open Living from Dining) false>`

If we examine the explanation, which we do in more detail in the next section, we find that for a territory to be `visually-open` from another, the `visual-openness` value must be greater than 0.6. The `visual-openness` value of the Living territory from the Dining territory is 0.44, so the above `visually-open` goal is not satisfied. Figure 4.2 shows the result of the `visual-openness` calculation. The photograph in Figure 4.3 shows the lack of visual openness between the Living and Dining territories.

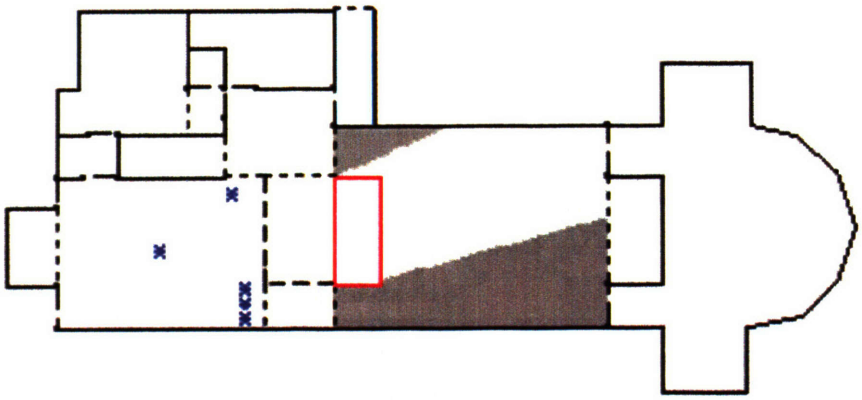


**Figure 4.2:** Shaded region of Living territory is visible from Dining territory;
`visual-openness` value of Living from Dining is 0.44.
Each * represents a viewpoint used in the calculation.

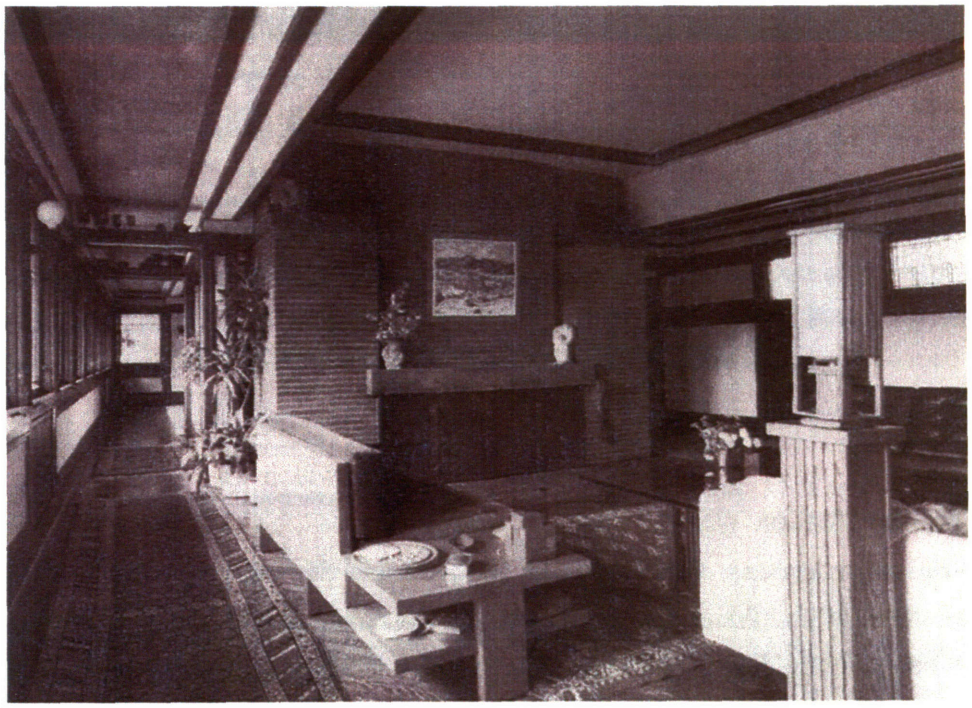


**Figure 4.3:** Living territory of Tomek house and view to Dining territory.

Example 2: `perceived-main-entryness`

Returning to the Gale house, introduced in Section 3.1, let's say that we want the front door to have a higher `perceived-main-entryness` value than the largest (middle) terrace door, called Terrace-door-2.

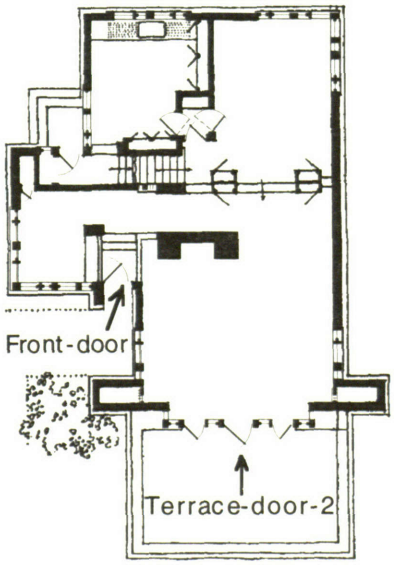



**Figure 4.4:** Floorplan showing doors used for `perceived-main-entryness` comparison.

We specify the goal:

```
<goal: (more-of (perceived-main-entryness of Front-door)
         than (perceived-main-entryness of Terrace-door-2)) true>
```

TAC evaluates the goal and returns:

goal satisfied? `yes`
desired value: `true`
actual value: `true`
explanation for actual value:

```
<expl:(more-of (perceived-main-entryness of Front-door)...) false>
```

The explanation for the goal expression's value tells us that the expression reduces to `(more-of ((37.68 149.54 0.5 hinged) no-value)`, which then reduces to `true`. Further examining the explanation we find that the vector `(37.68 149.54 0.5 hinged)` is the `perceived-main-entryness` value for the front door, and that this value represents distance (in feet) between the door and the usual approach point, the change in direction (in degrees) along the shortest path between the door and the usual approach point, the portion of the door that is wood, and the door hinge type. We find that `no-value` is the `perceived-main-entryness` value for the middle terrace door, and that the door does not have a value because no exterior paths lead to it from the usual approach point.

Figure 4.5 shows the path used to calculate the distance between and change in direction between the Gale front door and the usual approach point.

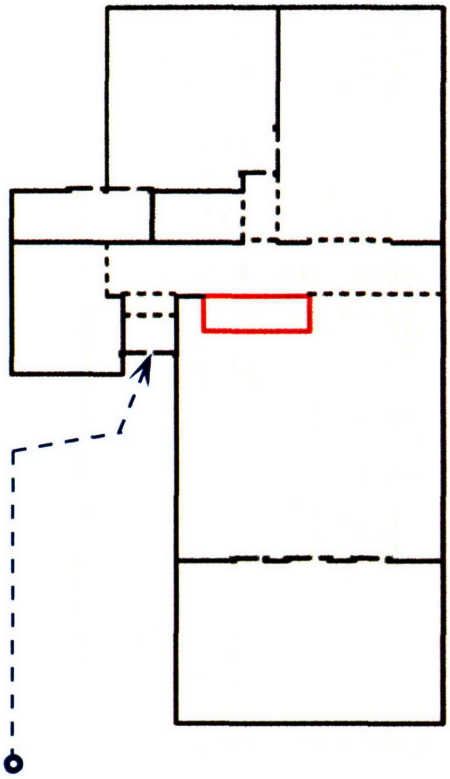


**Figure 4.5:** Path from usual approach point (**O**) to Gale Front door.

## 4.2 Explanations

An explanation represents a trace of a goal expression's evaluation. It is a tree whose nodes contain information about each step in the evaluation. By walking down the tree from the top explanation node, TAC can trace the reduction of the original goal expression to a value and reason about why a particular goal is not satisfied. It then can use that information to propose suggestions for repairing the design. Using an explanation to guide repair is a key step in TAC's dependency-directed redesign.

Example 1: `visually-open`

In the first example in the previous section, TAC evaluated this goal for the Tomek house:

```
<goal: (visually-open Living from Dining) true>
```

As shown in Figure 4.6 below, the trace of the evaluation is:

- start with the goal expression: `(visually-open Living from Dining)`
- substitute `visually-open`'s evaluation function body into the goal expression:
  `(gt (visual-openness Living from Dining) 0.6)`
- reduce the embedded function body expression to a value: `(gt 0.44 0.6)`
- reduce the resulting expression to a value: `false`

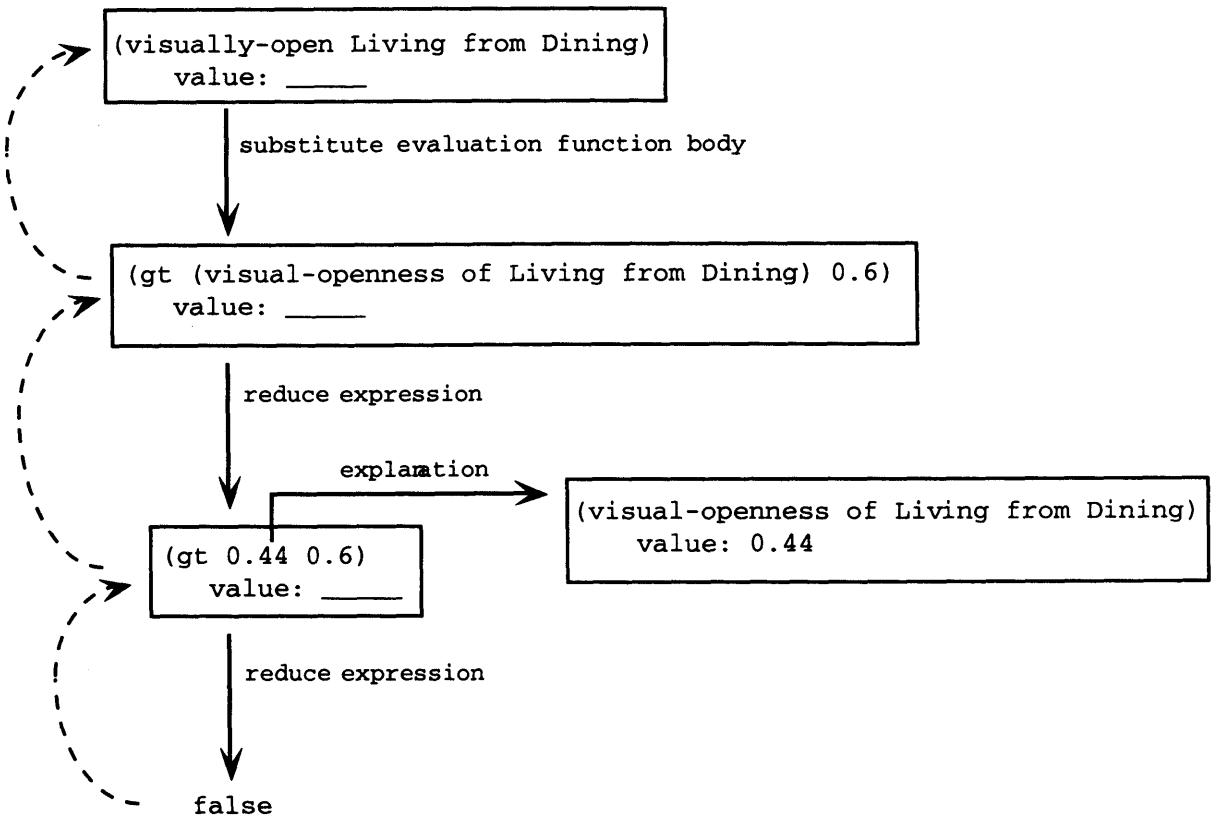



**Figure 4.6:** Explanation for `(visually-open Living from Dining)`.
Explanation nodes are boxed; dotted lines show propagation of value back to goal expression.

Each step in the trace is represented as an explanation node. Each node contains:

- an expression for the evaluation step
- a value for the expression
- a descriptor for the evaluation step's method; choices are:
  - reduce expression
  - substitute evaluation function body
  - collect components

Figure 4.6 illustrates two of the three descriptors for an evaluation step—reduce expression and substitute evaluation function body. The third descriptor, collect components, results in a more complex explanation structure and is illustrated in the next example. (See Appendix E for more details about constructing explanations.)

Example 2: physical-accessibility

Consider the second goal introduced in the previous section which specifies that the Gale house front door have more perceived-main-entryness than the middle terrace door:

```
<goal: (more-of (perceived-main-entryness of Front-door)
                than (perceived-main-entryness of Terrace-door-2)) true>
```

Recall that the value of perceived-main-entryness is a vector whose first two components represent physical-accessibility. The explanation for these two components for the Gale front door is shown in Figure 4.7. Notice that the first step in the evaluation is to collect components, and that the result of this step, namely the expression (make-vector ...) contains explanations for each of the components. We'll see later that TAC uses these component explanations when suggesting modifications to a design that will increase or decrease the perceived-main-entryness of an exterior door.

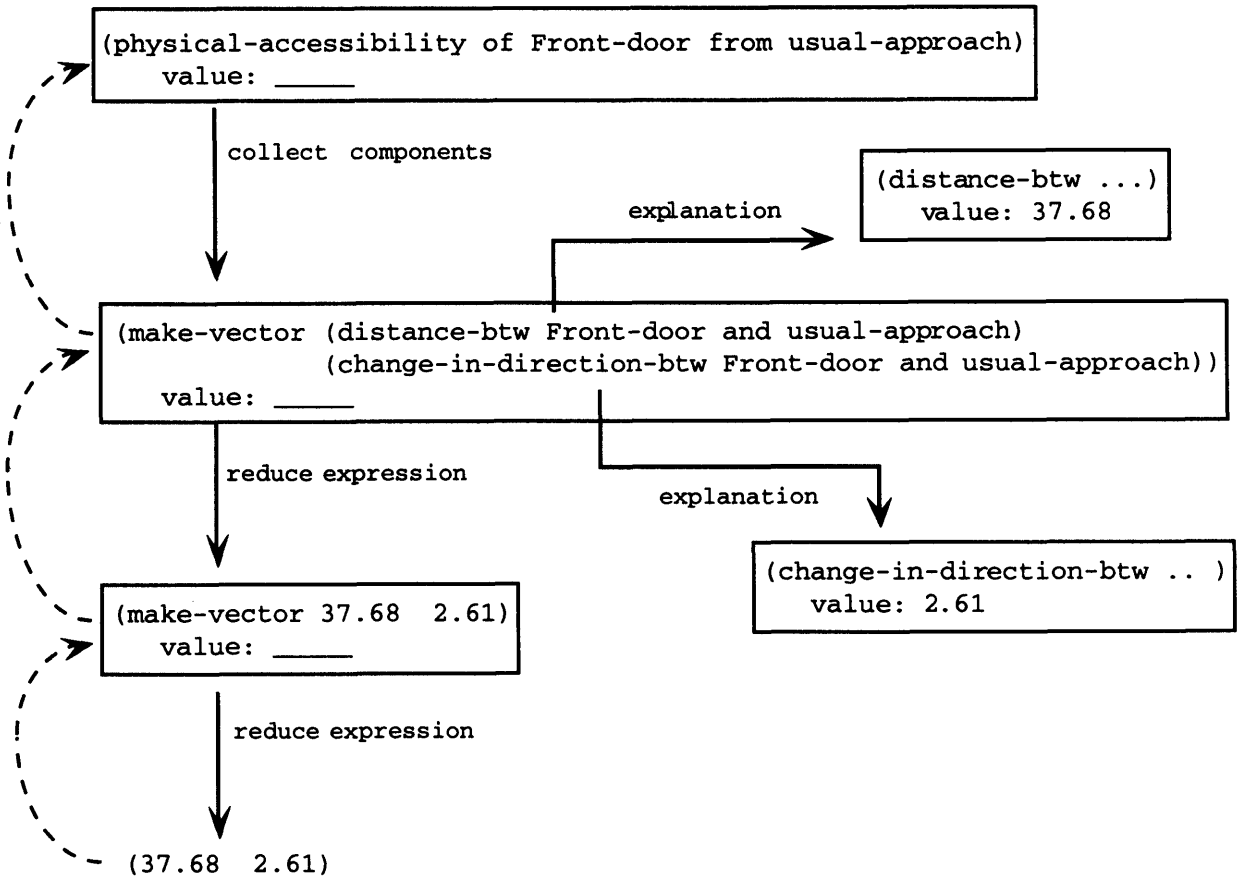



**Figure 4.7:** Explanation for physical-accessibility expression.
Explanation nodes are boxed; dotted lines show propagation of value.

# Chapter 5

# Suggesting Repairs

We saw in a previous example that the Living territory in the Tomek house is not visually open from the Dining territory. What changes might a designer suggest in order tc fix this situation? Since the value of the design characteristic `visual-openness` is not high enough, a designer might suggest increasing that value. What changes to the design might make this true? A designer might suggest removing or puncturing any objects that block the view between the two territories. The fireplace blocks the view, so he might suggest removing the fireplace, or perhaps puncturing it, as Frank Lloyd Wright did in the Robie house.

These suggestions are examples of the kinds of suggestions that TAC makes. Note that the suggestions can be made at any of several levels of abstraction. To make the Living territory visually open from the Dining territory:

Suggestion 1: Increase the `visual-openness` value. This suggestion is called a *value suggestion*; it suggests a direction in which to change a design characteristic's value.

Suggestion 2: Remove any design element that is blocking the view. This suggestion is called a *design suggestion*; it suggests design modifications in terms of categories of design elements.

Suggestion 3: Remove the fireplace. This suggestion is called a *design element suggestion*; it suggests design modifications in terms of specific design elements.

Notice that the language used in the suggestions is not the same as the language used to specify design goals. TAC must map the design goal specification language, which contains terms for architectural and mathematical concepts, into the repair suggestion language, which contains terms for design modifications. The next sections describe the different types of suggestions and the representation that enables this mapping.

## 5.1 Types of Suggestions

TAC begins by proposing repair suggestions at the highest level of abstraction—value suggestions. Value suggestions are stated in terms of design characteristics' values and propose satisfying a goal by increasing, decreasing, setting, or keeping a particular value the same. For the first three of these value changes, TAC translates the suggested change into design suggestions, which

propose modifications stated in terms of abstract categories of design elements, e.g. remove all elements blocking a view between two things. TAC does not propose design suggestions for keeping a value the same because many arrangements of design elements can yield a particular value. TAC assumes that if a desired value changes, the repair mechanism will modify the design so as to restore the desired value. (Chapter 7 explains how.) From design suggestions, TAC then proposes design element suggestions, which specify modifications to particular design elements. Separating suggestions into different levels of abstraction allows TAC to save more detailed information about why a particular design modification was performed, and, as discussed in Chapter 7, to search more efficiently for design solutions by reasoning about interactions between suggestions.

The following example illustrates the hierarchy of suggestion types.

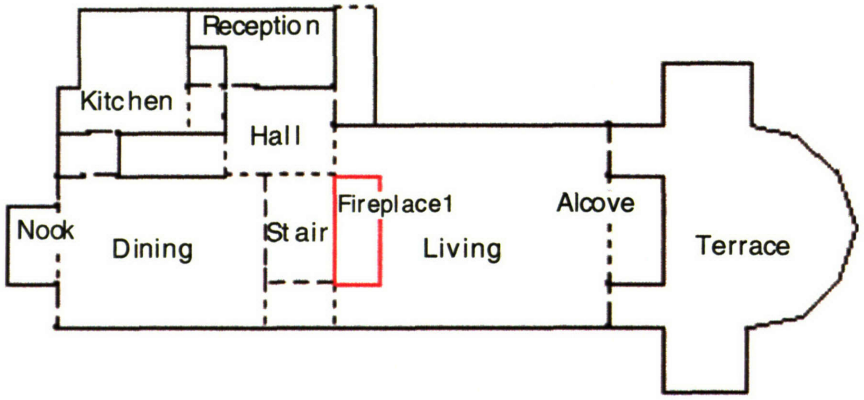



**Figure 5.1:** Territory model for Tomek house main (second) floor.

Let's again specify a goal stating that we want the Tomek Living territory to be visually open from the Dining territory:

```
<goal: (visually-open Living from Dining) true>.
```

TAC evaluates the goal, and as we've seen, finds it not satisfied.

TAC then proposes a value suggestion[1]:

```
(increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6)
```

Checking its knowledge base, TAC finds that removing or puncturing design elements that block the view between two things is likely to increase visual openness, so for the Living and Dining territories it proposes:

```
(or (remove blocking-elements-btw Living and Dining)
    (puncture blocking-elements-btw Living and Dining))
```

1. To improve readibility, the printed representation for a suggestion will be written in a more English-like syntax than is implemented.

Checking the design, it determines that the fireplace blocks the view, so it substitutes that element into the above suggestions and proposes design element suggestions:

```
(or (remove Fireplace1)
    (puncture Fireplace1))
```

As we'll see later, TAC creates a design for each of these suggestions. An example of a punctured fireplace is shown below.

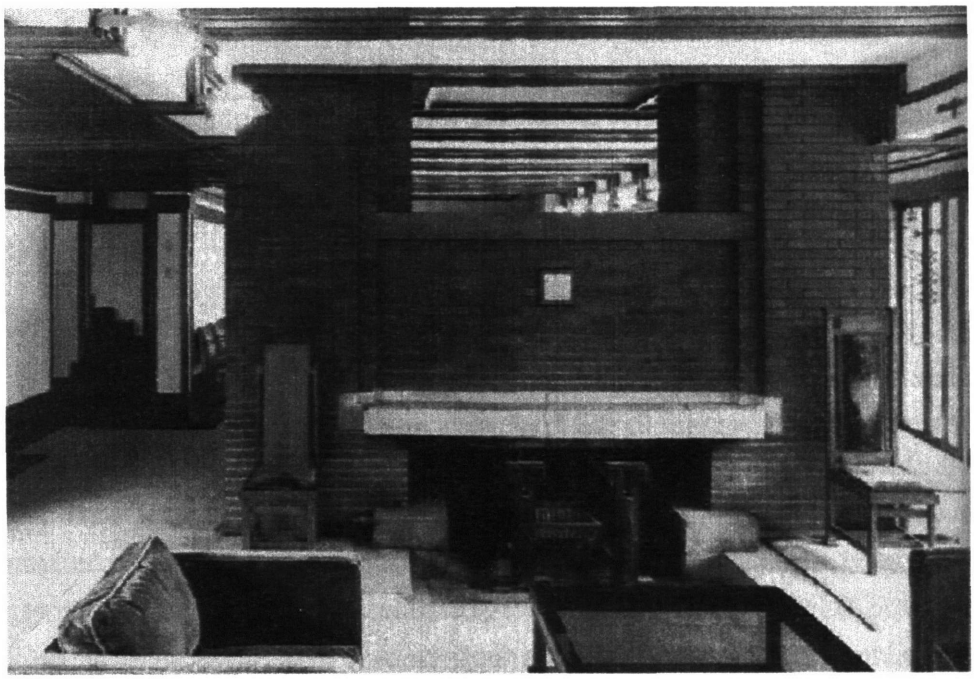


**Figure 5.2:** "Punctured" fireplace in the Robie house, designed by Frank Lloyd Wright. View is from the Living territory into the Dining territory.

## 5.2 Representing and Using Repair Knowledge

The knowledge needed for suggesting repairs is associated with design characteristics and TAC-functions. Design characteristics and TAC-functions have what we call *fixers*, domain-independent functions that know how to "fix" undesired values. Fixers propose value suggestions: they reason about how to get from a current design characteristic value to a particular desired value and recommend increasing, decreasing, setting, or keeping values. Fixers rely on design characteristics having what we call *setters*, *increasers*, and *decreasers*, each of which represents knowledge needed for translating value suggestions into design and design element suggestions, i.e. for translating a suggestion stated in terms of changing a value into suggestions stated in terms of modifications to particular design elements.

### 5.2.1 Fixers and Setters

We consider the design characteristic `visible-center` to illustrate TAC's representation and use of repair knowledge. This design characteristic tells us whether or not a territory's center is visible from another territory's center. It is boolean-valued and takes two arguments, each of which is a territory. Its evaluation function is defined in terms of the design characteristic `visible-from`:

```
visible-center (x y)
```

- evaluation function body `(visible-from (center of x) (center of y))`

Let's say that we want the center of the Living territory in the Tomek house to be visible from the center of the Dining territory. We specify the design goal:

```
<goal: (visible-center Living from Dining) true>
```

Evaluating the goal, TAC finds that it is unsatisfied and returns:

goal satisfied? `no`
desired value: `true`
actual value: `false`
explanation object for actual value: `<expl:(visible-center Living from Dining) false>`

To propose repair suggestions for making the Living territory's center visible from the Dining territory's center, TAC uses the goal expression's explanation. Beginning at the root node of the explanation, TAC regresses a desired value through the tree until it finds an explanation node containing an expression whose undesired value it knows how to "fix". TAC knows how to fix an expression if the design characteristic (or TAC-function) in the functional position has a fixer associated with it.

For the goal above, TAC starts with the explanation node for the goal expression `(visible-center Living from Dining)` shown at the top of Figure 5.3. It checks to see whether the design characteristic `visible-center` has a fixer, and it doesn't find one. TAC then follows a link via `visible-center`'s evaluation function body to the next explanation node, which contains the expression `(visible-from (center of Living) (center of Dining))`. It finds that `visible-from` has a fixer. So it calls the fixer, which proposes value suggestions for fixing the `visible-from` expression's undesired value of `false`.

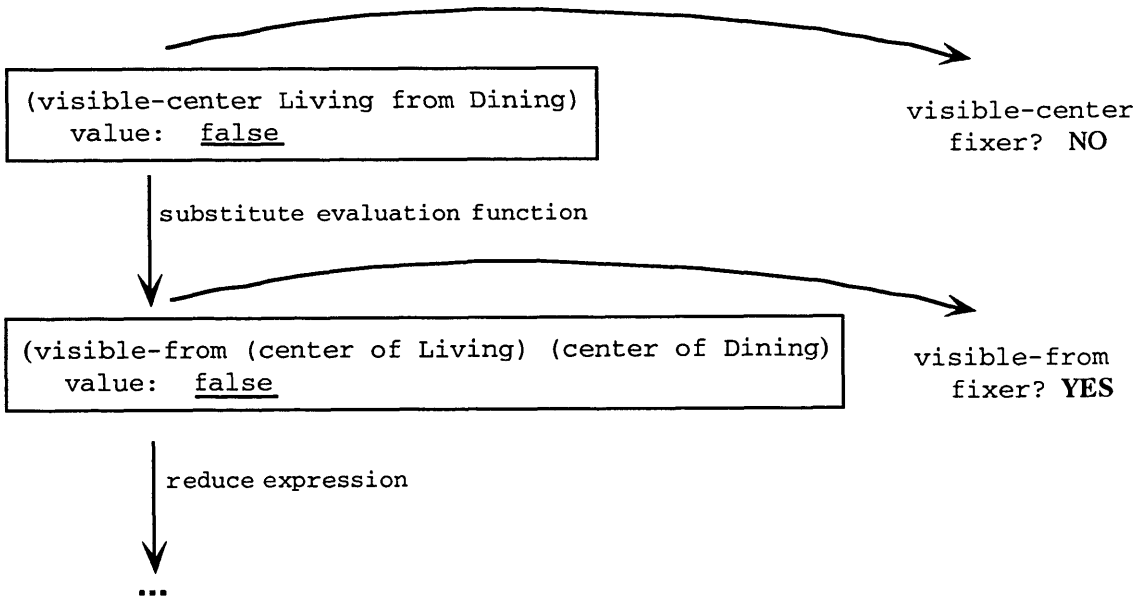



**Figure 5.3:** Searching for fixers in the explanation for `(visible-center Living from Dining)`. Boxes represent explanation nodes.

Recall that fixers reason about how to get from one value to another value. In the above example, there is not much reasoning in figuring out how to suggest getting from a value of `false` to a value of `true`: If the value is `false` and the desired value is `true`, suggest setting the value to `true`. Later steps in the suggestion proposal process will figure out how to "set" the value. As we'll see, TAC doesn't actually set the value. Instead, it suggests design modifications that it expects will cause a design characteristic to have the desired value.

Continuing with the example, TAC calls the fixer for `visible-center`, which proposes the following value suggestion:

```
(set-value of (visible-from (center of Living) (center of Dining))
          to true)
```

TAC then checks its knowledge base to see if it knows how to "set" a value for the design characteristic `visible-from`, i.e., it looks for *setters* associated with `visible-from`. Setters are design modification expressions, which consist of a design modification operator, also called a modifier, followed by an argument that describes a design element or set of design elements. For the design characteristic `visible-from`, for example, TAC finds the following setters:

```
visible-from (x y)
```

- setters for true

```
((remove blocking-elements-btw x and y)
 (puncture blocking-elements-btw x and y at sight-line-btw x and y)
 (screenify blocking-elements-btw x and y at sight-line-btw x and y))
```

- setters for false

```
((add blocking-element-btw x and y along sight-line-btw x and y)
 (fill * any (open-edges-btw x y) along sight-line-btw x y))
```
[2]

To make two things visible from one another, the setters recommend removing the design elements that block the view (termed blocking elements), puncturing the blocking elements along the line of sight, or screenifying the blocking elements along the line of sight.[3]

To propose suggestions to set the value of the expression `(visible-from (center of Living) (center of Dining))` to true, TAC substitutes `(center of Living)` and `(center of Dining)` for `x` and `y` in the setters and proposes:

```
(or (remove blocking-elements-btw (center of Living) and (center of Dining))
    (puncture blocking-elements-btw (center of Living) and (center of Dining)
              at sight-line-btw centers)
    (screenify blocking-elements-btw (center of Living) and (center of Dining)
              at sight-line-btw centers))
```

TAC then identifies the actual blocking elements in the design (by mapping edges in the territory model to the design elements from which they are derived) and substitutes the elements into the design suggestion expressions. The fireplace, called Fireplace1, blocks the view, so TAC proposes removing it or puncturing it along the sight line between centers:

```
(or (remove Fireplace1)
    (puncture Fireplace1 at <edge: 111.06...>))
```
[4]

Note that the last design suggestion, to screenify blocking elements, did not create a design element suggestion. TAC checks the applicability of each suggested modification operator, and finds that a fireplace can be removed or punctured, but not screenified. (See Appendix B for a list of modification operators.)

---

2. TAC uses "*" and "any" as syntactic sugar for iteration; "*" is a placeholder for each value returned by the expression following "any".
3. To screenify is to replace part or all of an opaque design element with a screen.
4. A sight line between two points is represented as an edge between the two points. `<edge: 111.06...>` is an abbreviated notation for an edge; `111.06` represents the x coordinate of one of the edge's endpoints.

### 5.2.2 Increasers, Decreasers, and Influences

The design characteristic `visible-from` is an example of a design characteristic for which we can define modification methods for setting its value. What about design characteristics whose value we don't know how to set? We may know, for example, how to increase the value of `visual-openness` between two territories, but not how to set `visual-openness` to a particular value. We say that the design characteristic `visible-from` is *partially invertible*;[5] we say that the design characteristic `visual-openness` is *noninvertible*.

Since the value of a noninvertible design characteristic cannot be set, TAC does not have setters for such a characteristic. Instead, TAC has *increasers* and *decreasers*, which, like setters, are design modification expressions consisting of a design modification operator and an argument that reduces to a design element or a set of design elements.

To illustrate how TAC proposes suggestions for noninvertible design characteristics, we return to the `visually-open` example for the Tomek house.

Recall that the design characteristic `visually-open` is defined in terms of the design characteristic `visual-openness`:

```
visually-open (x y)
```

- evaluation function body `(gt (visual-openness of x from y) 0.6)`

We'd like the Living territory to be visually open from the Dining territory, so we specify the goal: `<goal: (visually-open Living from Dining) true>`.

Evaluating the design goal, TAC finds that it is not satisfied and returns the explanation shown in Figure 4.6 and again here.

5. We use the term "partially invertible" rather than "invertible" because modification operators may not have their intended effects. See Section 2.3.1 for a discussion of this issue.

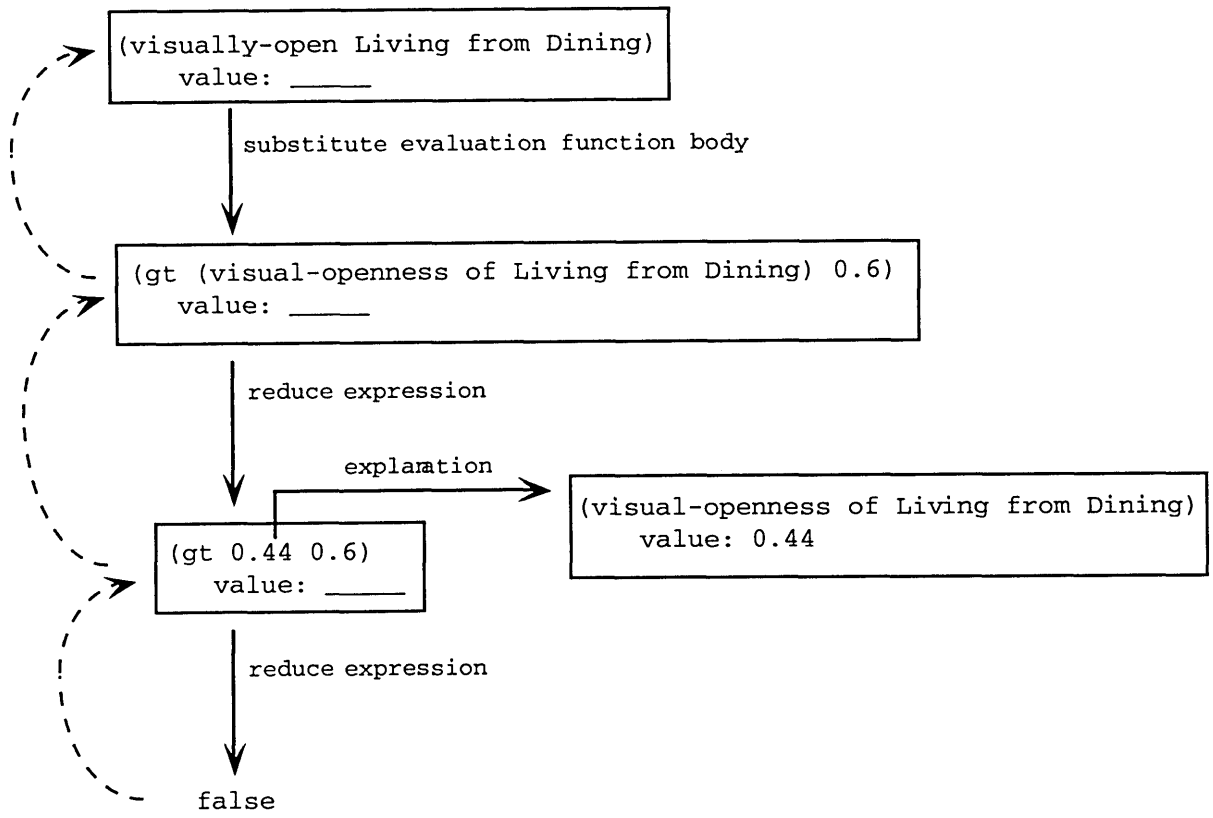



**Figure 5.4:** Explanation for `(visually-open Living from Dining)`. Boxes represent explanation nodes; dotted lines show propagation of value.

To propose value suggestions for making the Living territory visually open from the Dining territory, TAC starts at the root of the explanation and searches for a fixer. It doesn't find a fixer for the design characteristic `visually-open`, so it follows a link to the next explanation node and checks the TAC-function `gt`.

TAC finds that `gt` has a fixer, called `gt-fixer`. TAC calls it and proposes increasing the value of `visual-openness` until it is greater than 0.6:

```
(increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6)
```

How did TAC propose this suggestion?

The `gt-fixer` knows four ways to fix expressions of the form `(gt x y)` so that they are true:

- setting `x` such that `x` is greater than `y`
- increasing `x` until `x` is greater than `y`
- setting `y` such that `y` is less than `x`
- decreasing `y` until `y` is less than `x`

In this example, `x` is the explanation node whose expression is `(visual-openness Living Dining)`, and `y` is the constant 0.6. Since `y` is a constant, it cannot be changed, so the last two possible suggestions, setting or increasing its value, are not relevant. That leaves either setting the `visual-openness` value or increasing it.

If TAC is to set, increase, or decrease the value of an expression, the design characteristic or TAC-function in the functional position of the expression must have a setter, increaser, or decreaser. In the case of visual openness, its value cannot be set directly. The value is calculated by an opaque geometric procedure that figures out how much of a territory is visible from a specified location. The procedure is not invertible: it cannot be reversed to figure out how the design should be changed to realize a particular value. Hence, the design characteristic `visual-openness` does not have setters, and TAC does not suggest setting the value.

One suggestion remains: increasing the `visual-openness` value. There are ways to increase or decrease visual openness: Increasing the opacity of design elements between the two territories in question is likely to decrease the visual openness; decreasing the opacity of the design elements is likely to increase the visual openness. So we think of the opacity of intervening design elements as *influencing* the value of visual openness. Since visual openness increases as opacity of design elements decreases, we think of opacity as *negatively influencing* visual openness. In more mathematical terms, a positive influence is present when a monotonically increasing function relates the two design characteristics; a negative influence is present with a monotonically decreasing function.

To change the value of visual openness, we thus look at ways to change the opacity of intervening design elementts. To decrease the opacity of a design element we might remove that element, puncture it, turn it into a screen, move it somewhere else, or make it transparent (e.g. out of glass). To increase opacity, we might fill in a half-height wall to make it full height, turn an open doorway into a screen, or replace a window with a wall.

TAC represents an *influence* as a direction and a design characteristic expression. For example, TAC represents the negative influence of opacity on visual openness in the definition of the design characteristic `visual-openness`:

```
visual-openness
  • influences    ((- (opacity-of-elements-btw x y)))
```

TAC represents the methods for increasing or decreasing opacity as increasers and decreasers, respectively, on the design-characteristic `opacity-of-elements-btw`:

```
opacity-of-elements-btw (x y)
```

- decreasers
```
((remove blocking-elements-btw x and y)
 (puncture blocking-elements-btw x and y)
 (screenify blocking-elements-btw x and y))
```
- increasers
```
((fill * any (open-edges-btw x and y))
 (screenify * any (open-edges-btw x and y)))
```

Getting back to the current example, TAC is left with one possible value suggestion for making the Living territory visually open to the Dining territory: increasing the `visual-openness` value until it is greater than 0.6. TAC doesn't find increasers on `visual-openness` so it follows an influence link to `opacity-of-elements-btw`. Since the influence is in the negative direction, to increase visual openness, TAC must decrease opacity, so it looks for decreasers and finds those shown above. TAC then suggests:

```
(increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6)
```

Following the influence link, it suggests:

```
(decrease-value of (opacity-of-elements-btw Living and Dining)
                    until visual-openness greater than 0.6)
```

After proposing this suggestion, TAC substitutes the current arguments, `Living` and `Dining`, into the decreasers for the design characteristic `opacity-of-elements-btw` and proposes these design suggestions:

```
(or (remove blocking-elements-btw Living and Dining)
    (puncture blocking-elements-btw Living and Dining)
    (screenify blocking-elements-btw Living and Dining))
```

As in the previous example, TAC then proposes design element suggestions by substituting the actual blocking elements into the design suggestion expressions. It thus proposes removing or puncturing the existing fireplace, again pruning the inapplicable screenify suggestion:

```
(or (remove Fireplace1)
    (puncture Fireplace1))
```

Summarizing, the goal `<goal:(visually-open Living from Dining) true>` is unsatisfied because the `visual-openness` value is 0.44, which is not greater than the required minimum value of 0.6. TAC reasons that it must increase the value until it is greater than 0.6. To do this, it must decrease the opacity of elements between Living and Dining territories. To decrease opacity, it proposes removing, puncturing, or screenifying any elements that block the view between the two territories. It finds that the fireplace blocks the view, so it proposes removing or puncturing the fireplace; it prunes the screenify suggestion because fireplaces are not screenified.

## 5.3 Compound Suggestions

So far we've described *simple suggestions*, suggestions that contain a single design modification operator. TAC also may propose *compound suggestions*, which are conjunctions of simple suggestions.

In the `visual-openness` example above, TAC proposed either removing or puncturing the fireplace. Had another design element been blocking the view, for example the staircase, TAC also would have proposed:

```
(and (remove Fireplace1)
     (remove Staircase1))
(and (puncture Fireplace1)
     (puncture Staircase1))
```

Each of these suggestions is a compound suggestion and is carried out by performing the first simple suggestion's modification and creating a new design, then performing the second simple suggestion's modification on the new design. (Carrying out suggestions is discussed in Chapter 6.)

## 5.4 More Examples of Suggesting Repairs

The following examples further illustrate TAC's suggestion mechanism. The first example looks at requesting a fireplace on an interior edge and illustrates TAC's use of existential quantification and constraints in suggestions. The second example looks at proposing suggestions for a vector-valued design characteristic and illustrates TAC's use of vector comparison functions. The third example illustrates TAC's use of a partial order to compare and rank the values of vector-valued design characteristics.

Example 1: `on-interior-edge`

Figure 5.5 shows a territory model for another Frank Lloyd Wright Prairie house, the Horner house, built in 1908.

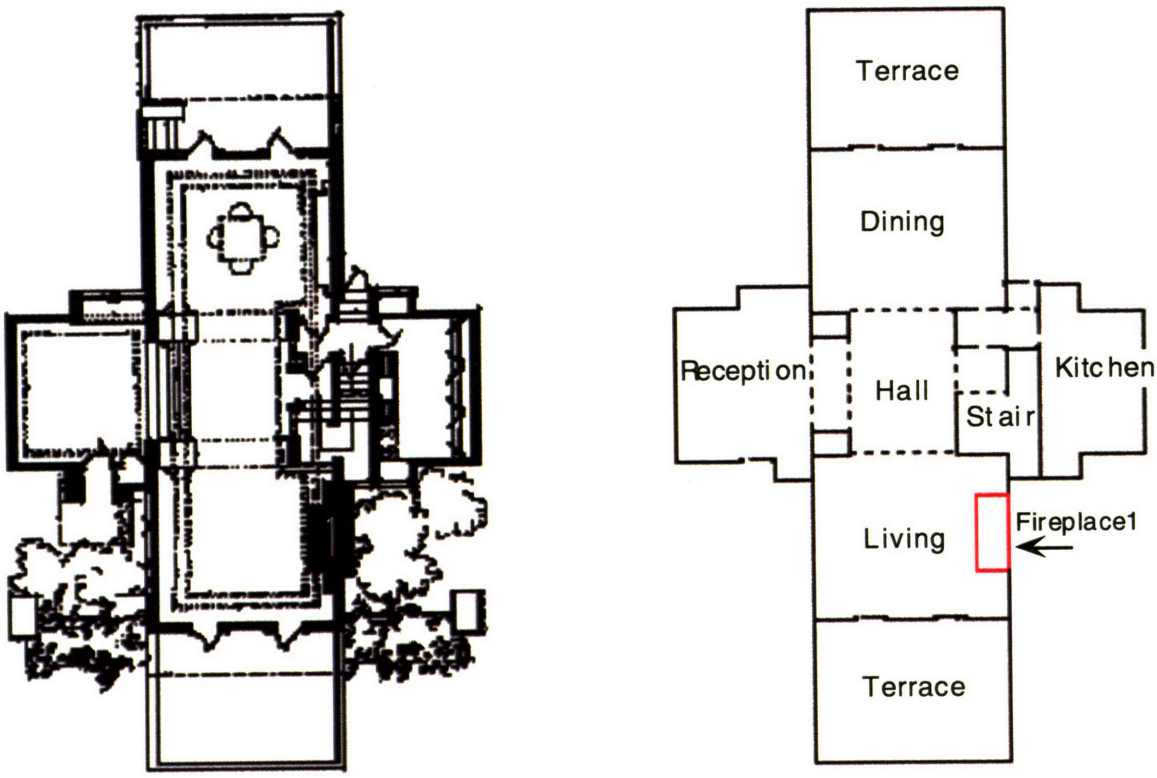



**Figure 5.5:** Floorplan and territory model for Horner house main (first) floor.

A goal often satisfied in the Prairie houses is having a fireplace on an interior edge of the Living territory:

```
<goal: (on-interior-edge fireplace Living) true>.
```

The design characteristic `on-interior-edge` takes an element type and a territory (or design) and checks all the elements of the specified type in the territory (or design) to see if at least one of them is on an interior edge. Its evaluation function body is defined in terms of the TAC-function `some`, which takes a predicate and a sequence of objects:

```
on-interior-edge (type territory)
```

- evaluation function body
```
(some element-on-interior-edge
      (elements-of-type type territory))
```

TAC finds the above goal unsatisfied for the Horner house and returns:

goal satisfied? `no`
desired value: `true`
actual value: `false`
explanation for actual value:

```
<expl:(on-interior-edge fireplace Living) false>
```

TAC examines the explanation and proposes two value suggestions: making the existing fireplace be on an interior edge or increasing the number of fireplaces in the Living territory to two, making sure that the new fireplace is on an interior edge.

```
(or (set-value of (element-on-interior-edge Fireplace1)
                  to true)
    (increase-value of (element-count fireplace in Living)
                  to 2 such that on-interior-edge is true))
```

TAC then translates the value suggestions into design suggestions, proposing to move the fireplace to an interior edge or add a new fireplace to an interior edge:

```
(or  (move Fireplace1 to any interior-edges-for Fireplace1)
     (add fireplace to Living such that on-interior-edge))
```

Checking the design, TAC proposes moving the fireplace to particular edges or adding a new fireplace to an interior edge.[6]

```
(or (move Fireplace1 to <edge: 15.75...>)
    (move Fireplace1 to <edge: 29.06...>)
    ...
    (add fireplace to Living such that on-interior-edge))
```

How does TAC propose these suggestions? It starts with the explanation for the goal's expression, `(on-interior-edge fireplace Living)`, and regresses the desired value of `true` through the tree, part of which is shown in Figure 5.6.

6. The `add` routine is opaque, so this design suggestion is not expanded further to specify a particular location.

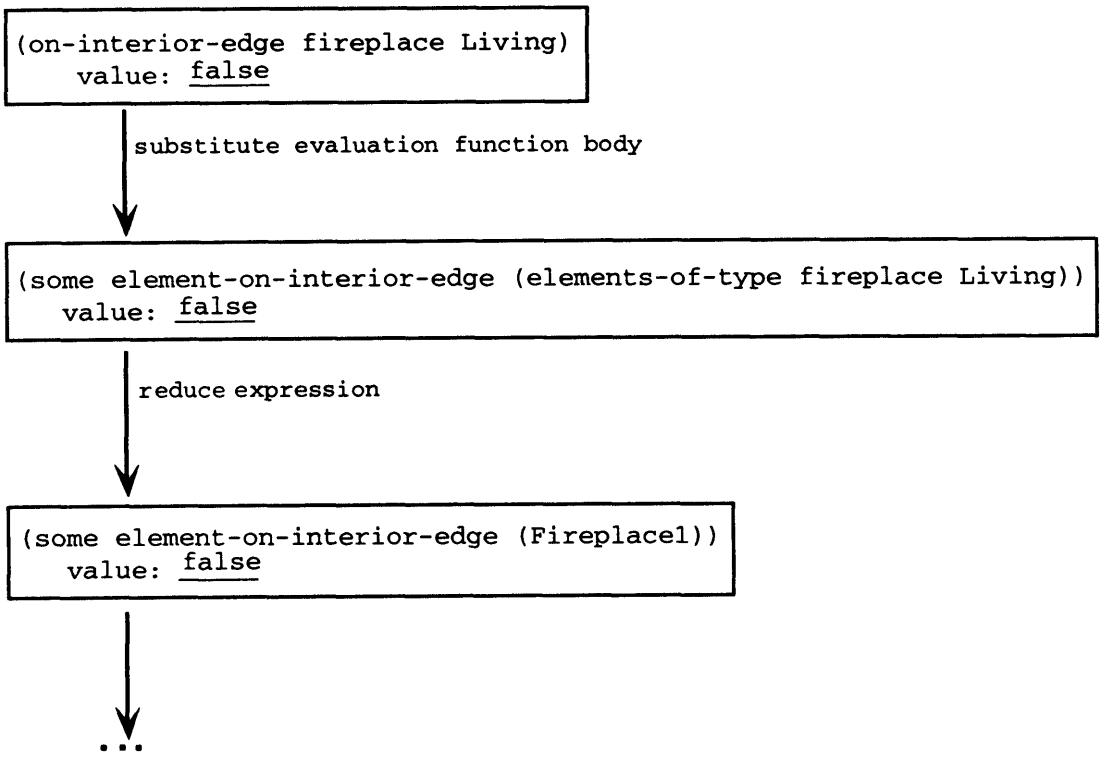



**Figure 5.6:** Part of explanation for `(on-interior-edge fireplace Living)`.

From the root explanation node, TAC walks down the tree looking for a fixer for the desired value `true`. The design characteristic `on-interior-edge` doesn't have one, so TAC follows a link to the next explanation node, which represents substitution of `on-interior-edge`'s evaluation function body into the goal expression. The TAC-function in the functional position of this node's expression, `some`, has a fixer, namely `some-fixer`. The `some-fixer` knows that the TAC-function `some`, which has the semantics of existential quantification, takes a boolean-valued design characteristic and a set-valued design characteristic as arguments. It also knows that in order for an expression of the form `(some x y)` to be true, the boolean-valued characteristic `x` must be true for at least one of the members of the set `y`. So it suggests either fixing any of the current members of `y` so that `x` is true, or fixing the set so that it contains a new member for which `x` is true. In this example, `x` is the design characteristic `element-on-interior-edge`, which checks whether a particular design element is on an interior edge; and `y` is the set of fireplaces in the Living territory, represented by the expression `(elements-of-type fireplace Living)`, which reduces to the set containing Fireplace1. TAC calls the `some-fixer`, which proposes the value suggestions shown earlier: set the value of `element-on-interior-edge` to `true` for Fireplace1; or increase the count of fireplaces in the Living territory to two, making sure that the new fireplace is on an

interior edge by saving `on-interior-edge` as a constraint in the suggestion.[7]

For the first value suggestion, TAC finds the setter for `element-on-interior-edge`: for a design element `x`, `(move x to any (interior-edges-for x))`. TAC substitutes the fireplace for `x`, identifies relevant interior edges, and proposes moving the fireplace to those edges.

For the second value suggestion, TAC finds the increaser for `element-type-count`: given a type `x` and a territory `y`, `(add x to y)`. It proposes adding a fireplace to the Living territory and includes the constraint that the new fireplace be on an interior edge. The constraint is passed on to the add modifier which uses the information when locating a new fireplace in the design.

Thus, TAC proposes:

```
(or (move Fireplace1 to <edge: 15.75...>)
    (move Fireplace1 to <edge: 29.06...>)
    ...
    (add fireplace to Living such that on-interior-edge))
```

Figure 5.7 summarizes how TAC's `some-fixer` proposes the above suggestions.

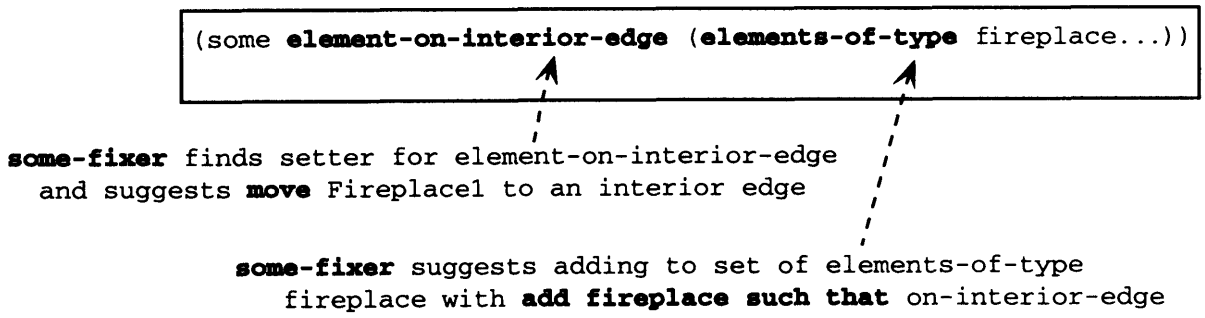



**Figure 5.7:** How `some-fixer` proposes moving the fireplace or adding a new fireplace.

Two of the new designs that have a fireplace on an interior edge are shown in Figure 5.8.

7. TAC knows that increasing the count, which is represented by the design characteristic `element-count`, increases the size of the set returned by `elements-of-type` because the evaluation function body of `element-count` is `(number-of (elements-of-type ...))`. See Appendix B for definitions of design characteristics. Constraints are represented as expressions in a constraint specification language, which is the same as the goal specification language.

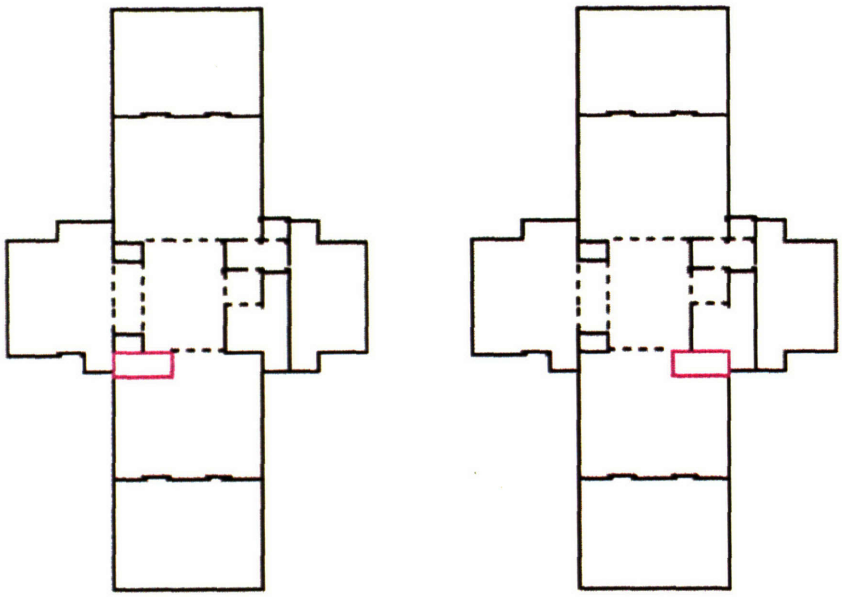


**Figure 5.8:** Two new Horner designs with a fireplace on an interior edge.

Example 2: `perceived-main-entryness`

So far, we've seen how TAC proposes suggestions for setting boolean-valued design characteristics, and for increasing or decreasing quantitative-valued design characteristics. How does TAC propose suggestions for vector-valued design characteristics?

Recall that the design characteristic `perceived-main-entryness` gives a measure of the perception of an exterior door as a main entry. Earlier we noted several factors that might influence a visitor's choice of door: visibility and the presence of built paths to the door from the usual approach point, distance and straightness of path between the door and the usual approach point, portion of the door that is wood, and the type of hinge on the door. The first two of these we represented as necessary conditions on `perceived-main-entryness`—they must be true for an exterior door to have a value. The second two we represented as components of `physical-accessibility`. The last two we represented as components of `formality-of-entry`.

Let's specify a goal regarding the perceived main entry for the Chatham house, a house in Arlington, Massachusetts.[8] A floorplan for the first floor is shown in Figure 5.5. Also shown are two possible approaches to the house: one to the front door and one to the side door.

8. The Chatham house was in the conceptual design stage of remodeling when this research was being carried out. Comparisons of TAC's design suggestions and the architects' suggestions are discussed in Section 8.1.

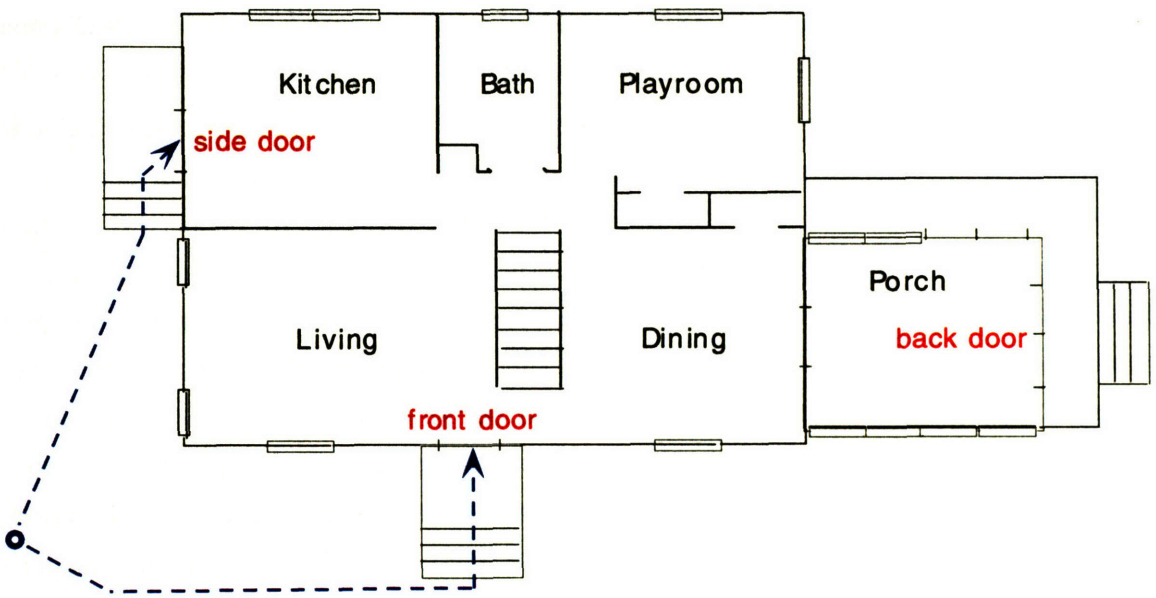



**Figure 5.9:** The Chatham house and approach paths to exterior doors.
The usual approach point is marked by **O**.

The house is situated on a corner lot, and the usual point from which it is approached is the corner between the two exterior doors.[9] Both doors are visible and accessible from the corner, so we assume a visitor will approach the door that he perceives as the main entry. The back door is not visible. If we want a visitor to approach the front door rather than the side door, we specify the goal:

```
<goal: (more-of (perceived-main-entryness of Front-door)
                than (perceived-main-entryness of Side-door)) true>
```

The `perceived-main-entryness` values, representing distance between the door and the usual approach point (in feet), change in direction between the door and the usual approach point (in degrees), portion of door that is wood, and door hinge type, are:

front door: `((23.95 117.24) (1.0 hinged))`

side door: `((22.03 48.94) (0.5 hinged))`.

To determine whether the front door's value is more than the side door's, TAC uses the vector comparison function `more-of`. In the case of `perceived-main-entryness`, the exterior door that is more easily accessible from the usual approach (closer and with a straighter path) and more formal (more solid, i.e. less glass, and hinged) has a higher `perceived-main-entryness` value.

9. A visitor walks up the driveway either to steps that lead to the side door or to a walk that leads to the front door.

TAC figures out that a higher `perceived-main-entryness` value means smaller distance and change in direction values and larger solidity and hinge-type values.[10] To explain TAC's reasoning, we return to the idea of influences introduced earlier.

Recall that influences represent relationships between design characteristics. A negative influence between two design characteristics implies that as one increases, the other decreases, and vice versa. Similarly, a positive influence implies that as one increases (or decreases), so does the other. In the case of a vector-valued design characteristic the components can be considered influences. An exterior door's `perceived-main-entryness` value increases, for example, as the door becomes more physically accessible and more formal. We represent knowledge such as this by extending component information for vector-valued design characteristics to include influence information. TAC then is able to figure out the behavior of the `more-of` function for vector-valued design characteristics by checking the directions of influence for the characteristics' components. Thus, influence information serves to inform both the `more-of` comparison function and the suggestion proposal mechanism.

Figure 5.10 summarizes the directions of influence for `perceived-main-entryness` and its components. (See Appendix B for details of the design characteristic definitions.)

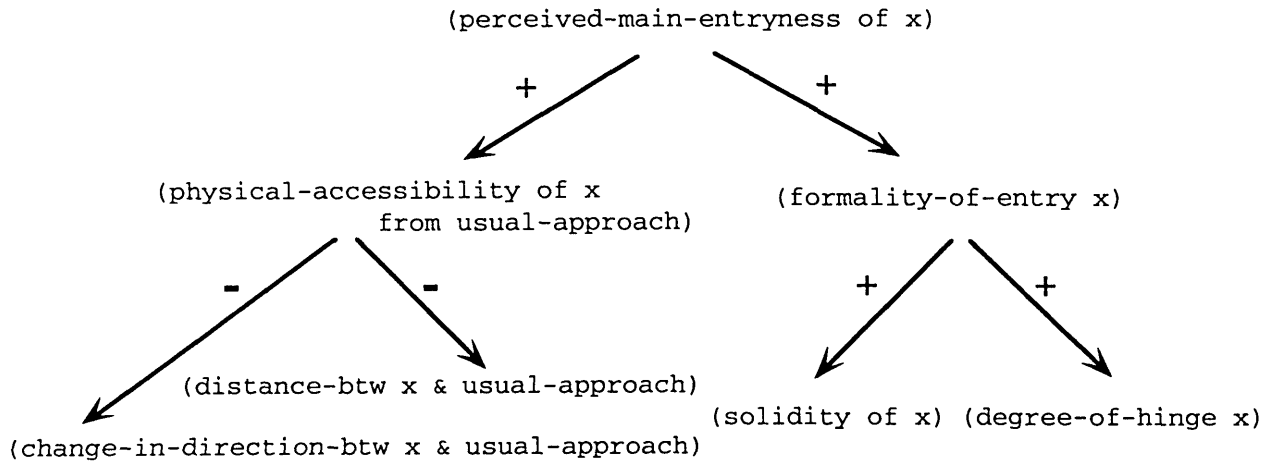



**Figure 5.10:** Directions of influence for `perceived-main-entryness`.

Getting back to the Chatham example, TAC compares the `perceived-main-entryness` values for the front door and the side door and finds that the front door does not have a higher value: the front door is further from the usual approach point and reached via a more crooked path.

10. As mentioned in Section 3.2.2, only one component must be strictly larger (or smaller); the rest may be greater than (less than) or equal.

TAC proposes increasing the value of perceived-main-entryness for the front door, decreasing the value of perceived-main-entryness for the side door, or removing the side door's perceived-main-entryness value (by setting the value to the symbol no-value):

```
(or (increase-value of (perceived-main-entryness of Front-door)
                    until (more-of (perceived-main-entryness of Front-door)
                                   than (perceived-main-entryness of Side-door)))
    (decrease-value of (perceived-main-entryness of Side-door)
                    until (less-of (perceived-main-entryness of Side-door)
                                   than (perceived-main-entryness of Front-door)))
    (set-value of (perceived-main-entryness of Side-door)
               to no-value))
```

How did TAC propose these suggestions? The first two suggestions look very much like those proposed by the gt-fixer introduced earlier. For (more-of x than y) to have a value of true, TAC suggests increasing x until x is more than y, or decreasing y until y is less than x. The third suggestion above is a new idea: for the expression (more-of x than y) to have a value of true, set the value of y to the symbol no-value. Because any value is more than no-value, the expression will have the desired value of true for any value x.

TAC follows influence links to determine how to increase or decrease perceived-main-entryness component values, and follows necessary condition links to determine how to set a perceived-main-entryness value to no-value.[11] It then proposes value suggestions, saving a stopping condition (until ...) in each suggestion:

```
(or ;; increase pme[12] of Front-door until more than that of Side-door
   (and decrease distance between Front-door and usual-approach
                until (pme of Front-door) is more than (pme of Side-door)
        decrease change in direction between Front-door and usual-approach
                until (pme of Front-door) is more than (pme of Side-door))
   ;; decrease pme for Side-door until less than that of Front-door
   (and increase distance between Side-door and usual-approach
                until (pme of Side-door) is less than (pme of Front-door)
        increase change in direction between Side-door and usual-approach
                until (pme of Side-door) is less than (pme of Front-door))
   ;; make pme of Side-door be no-value
   (or make Side-door not an exterior door
       remove exterior paths between Side-door and usual-approach
       make Side-door not visible from usual-approach))
```

11. Recall that the necessary conditions stipulate that the door must have a built exterior path to it from the usual approach point and that it must be visible from the usual approach point. Also recall that the design characteristic's domain stipulates an implicit necessary condition: only exterior doors have perceived-main-entryness values.

12. "pme" stands for perceived-main-entryness.

Checking its knowledge base for ways to accomplish the above suggestions, and checking the design for edges that might block the view of the side door, TAC proposes the suggestions shown below. (Stopping conditions are the same as those shown above.)

```
(or ;; increase pme of Front-door until more than that of Side-door
    (and move Front-door until ...
         maybe move exterior territories for Front-door
         decrease change in direction between Front-door and usual-approach
                  until ...
         maybe move exterior access territories for Front-door)
  ;; decrease pme for Side-door until less than that of Front-door
    (and move Side-door until ...
         maybe move exterior territories for Side-door
         increase change in direction between Side-door and usual-approach
                  until ...
         maybe move exterior access territories for Side-door)
  ;; make pme of Side-door be no-value
    (and replace Side-door with window
         remove exterior territories and paths for Side-door)
    (and replace Side-door with wall
         remove exterior territories and paths for Side-door)
    remove exterior paths between Side-door and usual approach
    fill <edge: 6.00 20.44...> along <edge: 10.11...>
    fill <edge: 6.00 22.89...> along <edge: 10.11...>)
```

Two things to note: First, the `fill` suggestions are intended to make the side door not visible from the usual approach point by filling open edges, e.g. by replacing them with a wall. Second, the compound suggestions each have additional simple suggestions that represent constraints on design modification operators. In this example, moving an exterior door may result in also moving any exterior territories associated with it (e.g. a porch and steps). Decreasing change in direction along a path results in removing or moving a design element that blocks a more direct path, which may necessitate also changing other connected territories. Removing the railing and entering a different side of a porch, for example, necessitates moving porch steps to the new entry side. See design Chatham#1 in Figure 5.11 for an example. Similarly, replacing an exterior door with a window necessitates also removing any exterior territories associated with the former door (e.g. porch, steps, walks). See design Chatham#2 in Figure 5.11.

In each of the two designs shown below, the front door has a higher `perceived-main-entryness` value than the side door. In Chatham#1, shown on the left, the front door and porch have been moved closer to the usual approach point, and the approach path has been straightened by entering the porch from the side rather than the front; the associated porch steps have been moved to accommodate the new entry side. In Chatham#2, shown on the right, the front door has a higher `perceived-main-entryness` value because the side door is no longer an entry; it has been replaced by a window. Additional designs that TAC creates are shown in Section 8.1.

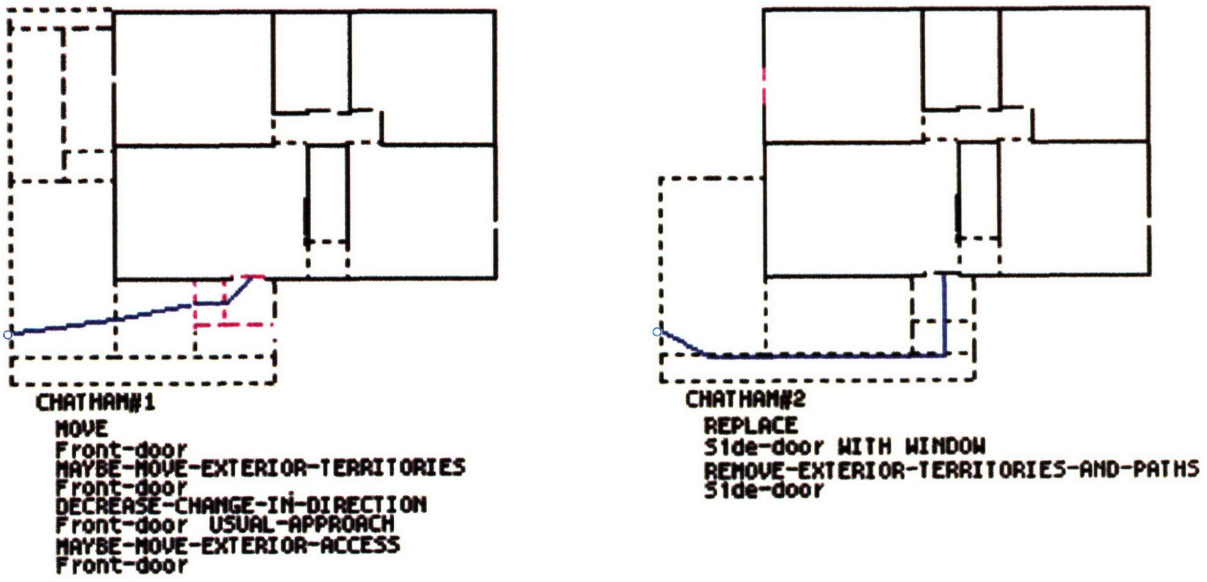



**Figure 5.11:** New Chatham designs with Front door having more perceived-main-entryness.

Example 3: one-perceived-main-entry

Let's now say that we're interested in having one perceived main entry, which we'll define as having one exterior door whose `perceived-main-entryness` value is more than all others. How does TAC either find the single object with the highest value, or propose suggestions for making a single object have a highest value? The key idea is to use a partial order: TAC puts all exterior doors into a partial order based on their `perceived-main-entryness` values, comparing each `perceived-main-entryness` value against all others using the vector comparison function `more-of`.

To have one perceived main entry means to have a single object at the top of the partial order. If there is no single object at the top, TAC proposes suggestions for moving each object up in the partial order and suggestions for moving all others down, so that there is a top object.

Two design characteristics, `perceived-main-entry` and `one-perceived-main-entry`, represent the notion of having a single perceived main entry. The design characteristic `perceived-main-entry` takes a design as an argument and returns an exterior door or the symbol `no-value`. Its evaluation function is defined in terms of the TAC-functions `ordered-elements` and `top-of`:

`perceived-main-entry (x)`

- evaluation function body

```
(top-of (ordered-elements x perceived-main-entryness more-of)
```

The TAC-function `ordered-elements` takes a design, a design characteristic to be used for comparison, and a comparison function. It collects design elements whose type matches the specified design characteristic's domain and constructs a partial order of those elements using the supplied comparison function. The TAC-function `top-of` takes a partial order and returns the top object or `no-value`.

The design characteristic `one-perceived-main-entry` takes a design as an argument and returns `true` if the design has a `perceived-main-entry`, `false` otherwise:

```
one-perceived-main-entry
  • evaluation function body   (if is-a-value (perceived-main-entry x)
                                   true
                               else false)
```

If we're interested in having one perceived main entry for the Chatham house, we specify a goal with the expression `(one-perceived-main-entry Chatham)` and a value of `true`:

```
<goal: (one-perceived-main-entry Chatham) true>
```

TAC finds that the goal is not satisfied and returns an explanation. Examining the `perceived-main-entryness` partial order in the explanation, TAC finds no top, so it proposes making either door the top of the partial order:

```
(or make Front-door top
    make Side-door top)
```

TAC knows that to make an object be at the top of a partial order, it must make the object's value dominate all other values, and it uses the `more-of` function's fixer to accomplish this. It proposes increasing or decreasing `perceived-main-entryness` values or setting a door's value to `no-value`, as in Example 2.

TAC proposes:

```
(or ;; make Front-door top
     increase Front-door pme value
     set Side-door pme value to no-value
     decrease Side-door pme value
   ;; make Side-door top
     increase Side-door pme value
     set Front-door pme value to no-value
     decrease Front-door pme value)
```

TAC then proposes increasing, decreasing, or setting particular `perceived-main-entryness` component and necessary condition values:

```
(or ;; increase Front-door pme until more than Side-door's
    (and decrease distance between Front-door and usual-approach
         decrease change in direction between Front-door and usual-approach)
    ;; set Side-door pme to no-value
    (or make Side-door not an exterior door
         remove exterior paths between Side-door and usual-approach
         make Side-door not visible from usual-approach)
    ;; decrease Side-door pme until less than Front-door's
    (and increase distance between Side-door and usual-approach
         increase change in direction between Side-door and usual-approach)
    ;; increase Side-door pme until more than Front-door's
    increase solidity of Side-door to 1.0
    ;; set Front-door pme to no-value
    (or make Front-door not an exterior door
        remove exterior paths between Front-door and usual-approach
        make Front-door not visible from usual-approach)
    ;; decrease Front-door pme until less than Side-door's
    decrease solidity of front door to 0.5)
```

TAC then proposes the following 14 specific suggestions, similar to those in Example 2:

```
(or ;; increase Front-door pme until more than Side-door's
    (and move Front-door until ...
         maybe move exterior territories for Front-door
         decrease change in direction between Front-door and usual-approach
                   until ...
         maybe move exterior access territories for Front-door)
    ;; set Side-door pme to no-value
    (and replace Side-door with window
          remove exterior territories and paths for Side-door)
    (and replace Side-door with wall
          remove exterior territories and paths for Side-door)
    remove exterior paths between Side-door and usual approach
    fill <edge: 6.00 20.44...> along <edge: 10.11...>
    fill <edge: 6.00 22.89...> along <edge: 10.11...>
    ;; decrease Side-door pme until less than Front-door's
    (and move Side-door until ...
          maybe move exterior territories for Side-door
          increase change in direction between Side-door and usual-approach
                   until ...
          maybe move exterior access territories for Side-door)
    ;; increase Side-door pme until more than Front-door's
    increase solidity of Side-door to 1.0
    ;; set Front-door pme to no-value
    (and replace Front-door with window
          remove exterior territories and paths for Front-door)
    (and replace Front-door with wall
          remove exterior territories and paths for Front-door)
    remove exterior paths between Front-door and usual-approach
    fill <edge: 10.11 30.89...> along <edge: 25.00...>
```

```
fill <edge: 22.33 30.89...> along <edge: 25.00...>
;; decrease Front-door pme until less than Side-door's
decrease solidity of front door to 0.5)
```

Designs created by carrying out these suggestions are shown in Section 8.1 and Appendix G.

# Chapter 6

# Performing Repairs

After proposing repair suggestions for a design that does not satisfy desired goals, TAC performs repairs by carrying out the suggestions. The result is a set of new designs. Carrying out suggestions is straightforward, but involves details having to do with the automatic derivation of territories from design elements.

### Creating a New Design

Carrying out a suggestion for a design is a four step process: Make a copy of the design, perform the proposed modification on the new design, update the circulation model, update the territory model.

A modification operator may affect all models that represent a design. It changes the design element model, for example, by removing, moving, or puncturing a particular design element. Since the edges in an edge model are derived from the design elements, each of these operators also changes the edge model. The circulation model is changed since it is derived from the edge model. It must be updated by creating nodes and arcs for new doorways, and by removing arcs for doorways (and their corresponding nodes) that have been removed. The territory model may or may not be changed. If a design element's edges help form a territory boundary and the design element is moved or removed, then the territory's boundary may change. If as a result of changes, the territory is no longer *well-formed*, by which we mean bounded by a connected set of edges, then it is removed. TAC then attempts to redefine the territory using both old and new edges. An example of TAC's territory redefinition procedure is shown later in this section.[1]

### Identity Issues

Repair suggestions are intended, of course, to cause an unsatisfied goal to become satisfied. After carrying out a suggestion, TAC checks the resulting new design to see if the unsatisfied goal is now satisfied. Given a goal specifying that the Living territory in the Tomek house be visually open from the Dining territory, for example, TAC suggests removing the fireplace, creates a new design with the fireplace removed, then checks whether the Living territory in the new design is

1. Many computational geometry routines depend on having closed polygons, hence our insistence on well-formedness. TAC redefines territories and defines new territories by walking through all edges in the new edge model, identifying all closed polygons not already part of defined territories.

visually open from the Dining territory. In this example, checking for goal satisfaction is straightforward because the two design objects in the goal—the Living and Dining territories—have not changed. Even though the fireplace helped bound the Living territory in the previous design, removing it does not change the territory's bounds: the fireplace and stair shared an edge, so that edge, now belonging only to the stair, still bounds the territory. Figure 6.1 shows the unchanged territory boundaries and the results of TAC's `visual-openness` calculations for the original design and the new design.

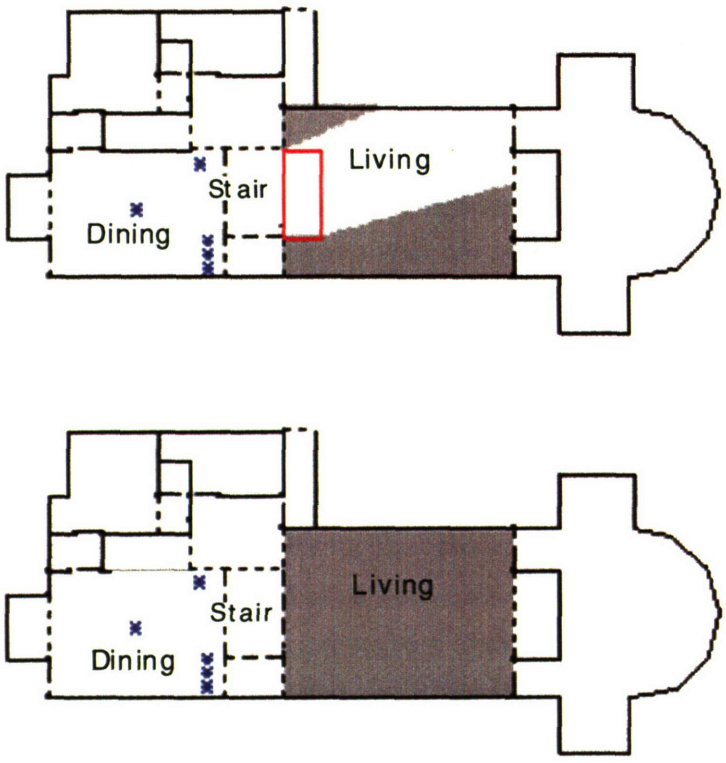



**Figure 6.1:** Tomek territory models: with fireplace and without.
Shaded region is visible; each * is a viewpoint. `Visual-openness` of Living from Dining in Tomek (top model) is 0.44; with fireplace removed, it is 1.0 (assuming the edge shared by fireplace and stair is replaced by a stair railing).

What happens if a new design is modified in such a way that one of the design objects specified in the intended goal no longer exists? How can the goal be evaluated in the new design? This problem is a difficult and important one and has been encountered in other evaluation and repair systems in which objects can appear and disappear (e.g. Simmons, 1988). We did not study the problem extensively, but rather mimicked what human designers do in such situations: map objects in a new design to objects in a previous design using an "approximately the same" test. An example illustrates.

Consider the Chatham house introduced earlier, and imagine that we want the Living territory to be visually open from the Dining territory. TAC finds that the Living territory is not visually open from the Dining territory. Results of the visual openness calculation are shown in Figure 6.2.

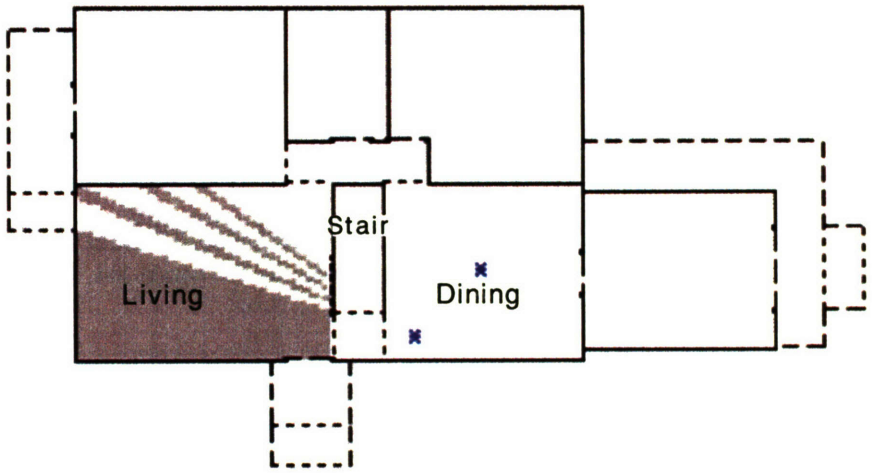



**Figure 6.2:** Territory model for Chatham house, first floor.
Region of Living territory visible from Dining is shown shaded;
`visual-openness` value is 0.55.

One of TAC's suggestions for making the Living territory visually open from the Dining is to rotate the stair. Figure 6.3 shows a new territory model with the stair rotated and the region of the Living territory now visible from the Dining territory.

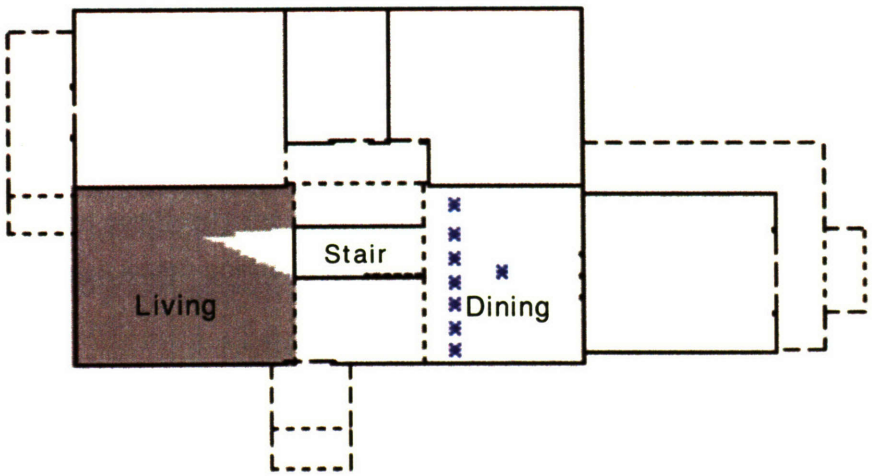



**Figure 6.3:** New Chatham design with stair rotated.
`visual-openness` value of Living from Dining is 0.94.

While the Living territory in the new design is visually open from the Dining territory, the territory bounds have changed (Figure 6.4), and the original territories for which the goal was specified no longer exist. The original territories were removed because they were not well-formed after stair rotation. TAC was able to create new territories based on the new stair location and map them to the original territories.[2] It then evaluated the goal with respect to the new territories. In other cases, the physical form may be so changed that new territories cannot be mapped to old ones. In such a situation, it is not possible to check for goal satisfaction, and TAC signals this to the designer.

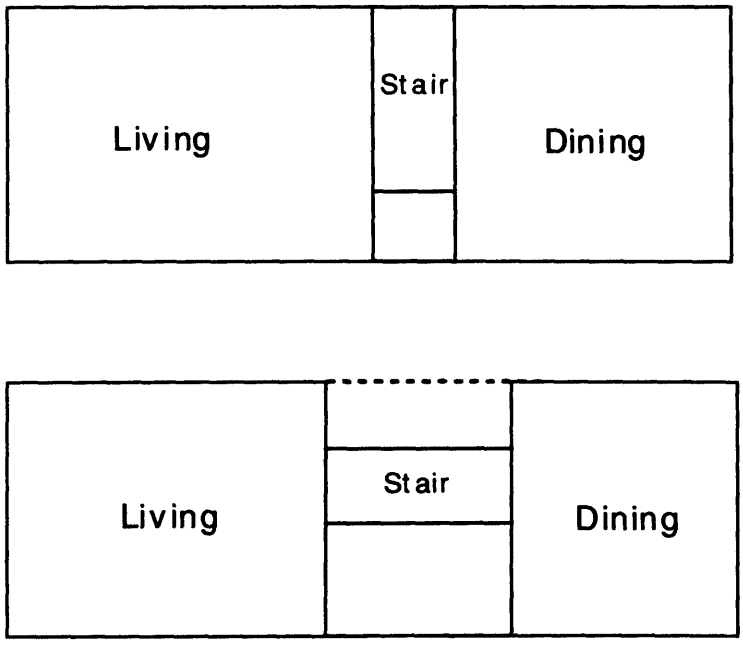



**Figure 6.4:** Bounds for Living and Dining territories. Top shows user-defined territories in original design; bottom shows TAC-defined territories in new design.

2. The territories were partially bounded by edges of the stair and edges projected from the stair. Rotation of the stair created new projected edges and changed edge locations in such a way that the original territories were no longer well-formed (i.e. closed polygons). TAC defined new territories by identifying all closed polygons in the new design. It then mapped new territories to old ones by finding unions of polygons that most closely matched original territory bounds. See Appendix F for more details about edge and territory model modifications.

### Bookkeeping Issues

To simplify keeping track of edges and their derivations, and to ensure that no edges overlap, projected edges derived from the design element to be modified are removed prior to the modification. (Recall that projected edges do not correspond to design elements, but rather are extensions of one or more edges derived from design elements. See Section 3.1 for examples.) The modification is then carried out, and new projected edges are added back for the modified design element. Some of these new projected edges may help form boundaries for new territories. In the above example, the projected edges for the stair were removed prior to rotation. Figure 6.5 below shows the overlapping edges that result if projected edges are not removed prior to rotating the stair. See Appendix F for more details about removing projected edges.

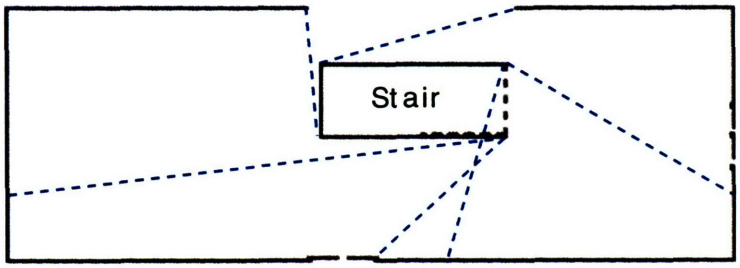



**Figure 6.5:** Rotating a stair without removing projected edges.

To simplify an edge model after a modification has been carried out, TAC attempts to decrease the number of edges by combining neighboring edges into single edges where possible.[3] If edges are not combined, the edge models for designs that are several modifications from an original design run the risk of containing many small edges: with each new modification, new edges are added, each of which may split existing edges (since edges do not overlap). In a model with many small edges, tests for sameness among designs are less efficient and more difficult, and the number of suggestions needlessly increases when a modifier is dependent on particular edge locations. Combining edges where possible helps guard against this situation.

3. Two edges can be combined if they have the same derivation information and their shared endpoint has only the two edges.

# Chapter 7

# Control Structure

So far we've seen how TAC represents a design problem as a design and a set of design goals, evaluates design goals, suggests repairs, and carries out the repairs to create new designs. This chapter describes TAC's control structure—how it puts these steps together to perform its dependency-directed redesign, searching a design space for solutions to a design problem by iterating through an evaluate and repair cycle guided by knowledge of which goals need satisfying and how to satisfy them.

This chapter describes three control structures, starting with the one TAC uses, then discusses the issues of goal order and termination, and presents the results of a series of experiments that explored TAC's behavior under different circumstances.

## 7.1 Overview

As discussed in Chapter 2, simultaneously satisfying multiple goals can be very difficult. It is possible, however, to generate an intermediate design that satisfies a subset of the goals, then repair that design to satisfy remaining goals. This approach works because designs generated in this way are often useful intermediate points from which further progress is possible. We experimented with three different control structures for solving design problems in this manner. The control structures share the following characteristics:

- They search for minimal solutions, i.e. solutions that are the fewest number of modification steps from the original design.
- They reduce search by searching the repair suggestion space before creating new designs. They differ in how suggestions are generated and pruned.
- They create new designs by carrying out repair suggestions. They then iterate through evaluation and repair cycles for each new design, stopping when the design is a solution, when the intended goal for the design is not satisfied,[1] when they have seen the design before, or when they have reached an iteration limit. When a particular design satisfies some but not all goals, they repair the design, i.e. enter a new evaluation and repair cycle with that design as the starting point.

1. Recall that a suggestion, and by extension the designs generated by carrying out a suggestion, have an intended goal: the unsatisfied goal expected to be satisfied after carrying out the suggestion.

TAC uses what we call the *sequential-with-lookahead* control structure, which focuses on a single goal at a time, checking for interactions between suggestions for that goal and suggestions for all other goals. It creates new designs by carrying out each surviving suggestion. These new designs then are used as starting points for working on the next goal, and so on. After the last goal has been checked, TAC attempts to repair any designs that do not satisfy all goals.

The other two control structures were designed to test the effectiveness of this control structure's two components: sequential design generation and lookahead suggestion proposal. The *sequential* control structure focuses on sequential design generation: it proposes suggestions and creates new designs sequentially a single goal at a time. It does no lookahead, so it does not check suggestions for interactions with other goals. Instead, it creates new designs for all suggestions. The *concurrent* control structure focuses on the lookahead mechanism: it uses a similar mechanism to check for interactions prior to creating designs. In the context of the original design, it proposes suggestions for all goals at the same time, then checks those suggestions for interaction. In contrast, the sequential-with-lookahead and sequential control structures propose suggestions for one goal at a time in the context of a new design. The sequential-with-lookahead control structure proved to combine benefits and avoids costs of the other two control structures.

The following sections describe the control structures, starting with a discussion of the important issue of goal interaction.

## 7.2 Goal Interaction: Conflict and Synergy

A key aspect of TAC's intelligence is its ability to search the suggestion space prior to creating new designs: TAC avoids generating designs it knows won't satisfy the goals, and it seeks out designs that accomplish the goals efficiently. To limit its design generation, TAC looks for potential interaction between goals: It looks for *conflict*, situations in which satisfying a new goal will cause an already satisfied goal to become unsatisfied (or "clobbered"); and *synergy*, situations in which a single modification will result in more than one unsatisfied goal becoming satisfied. By spotting conflict or synergy, TAC avoids unnecessary design modification steps. We've identified three kinds of conflict and synergy: *obvious*, *predictable*, and *unpredictable*. TAC handles the first two of these: obvious interactions are detected by comparing goals, predictable interactions are detected by comparing suggestions for goals. Unpredictable interactions, by their very nature, cannot be detected ahead of time.

### Obvious Interaction

Conflict and synergy sometimes can be spotted at the goal level by comparing goal expressions and desired values. TAC always checks first for this kind of interaction, which we've called *obvious*.

Consider two goals, one that specifies a Living territory visually open from a Dining territory, and one that specifies a Living territory not visually open from a Dining territory:

```
<goal: (visually-open Living from Dining) true>
<goal: (visually-open Living from Dining) false>
```

It's easy to see that these two goals conflict: the expressions are the same and the desired values are incompatible. We've called this kind of interaction *obvious conflict.*

Similarly, consider a goal that specifies a `visual-openness` value greater than 0.6 and one that specifies a `visual-openness` value greater than 0.5:

```
<goal: (gt (visual-openness Living from Dining) 0.6) true>
<goal: (gt (visual-openness Living from Dining) 0.5) true>
```

It's easy to see that satisfying the first goal will satisfy the second goal. We've called this kind of interaction *obvious synergy.*

### Predictable Interaction

Most of the time, however, interaction cannot be detected at the goal level because goal expressions differ more than those above. So TAC also checks for interaction at the suggestion level: It looks for conflict and synergy between suggestions for goals. Interactions detected at the suggestion level are called *predictable.*

Consider two goals for the Tomek house, one that specifies that the Living territory be visually open from the Dining territory and one that specifies one fireplace in the Living territory:[2]

```
<goal: (visually-open Living from Dining) true>
<goal: (fireplace-count in Living) 1>
```

It's not easy to tell whether there is interaction between these goals. If TAC looks at the following suggestions for the each of these goals, however, it can detect a conflict.

To satisfy the `visually-open` goal, TAC proposes to remove or puncture the fireplace (which is located in the Living territory and blocks the view between Living and Dining):

```
(or (remove Fireplace1)
    (puncture Fireplace1))
```

2. For readability, the `element-count` characteristic, mentioned in Section 5.4, has been replaced by `fireplace-count`. The expression `(fireplace-count in Living)` is equivalent to `(element-count fireplace Living)`.

To keep the `fireplace-count` goal satisfied, TAC suggests keeping the count at 1:

```
(keep-value of (fireplace-count in Living) 1)
```

The first suggestion above, to remove the fireplace, would reduce the number of fireplaces in the Living territory to zero, which conflicts with the desire to keep one fireplace there. TAC is able to prune that suggestion and avoid creating a design that cannot achieve both goals. We call this situation *predictable conflict.* Note that there is a remaining suggestion for the `visually-open` goal, namely to puncture the fireplace, so the goals themselves are not in conflict.

Just as conflict can be predictable, so can synergy. Consider two other goals for the Tomek house, one specifying that the Living territory be visually open from the Dining territory, and one specifying that the center of the Living territory be visible from the center of the Dining territory:

```
<goal: (visually-open Living from Dining) true>
<goal: (visible-center Living from Dining>) true>
```

It's not easy to spot interaction between these two goals, so TAC checks for interaction at the suggestion level.

To satisfy the `visually-open` goal, TAC suggests removing or puncturing the fireplace:

```
(or (remove Fireplace1)
    (puncture Fireplace1))
```

To satisfy the `visible-center` goal, TAC suggests removing the fireplace or puncturing it along the line of sight between territory centers:

```
(or (remove Fireplace1)
    (puncture Fireplace1 at <edge: 111.07...>))
```
[3]

Removing the fireplace may satisfy both goals since this suggestion is proposed for both goals. We've called this situation *predictable synergy.* Puncturing the fireplace along the line of sight may also satisfy both goals and is a more subtle example of predictable synergy. TAC proposes puncturing the fireplace to make the Living territory visually open. It also proposes puncturing the fireplace to make the center of the Living territory visible, but specifies a particular location for the puncture. The second suggestion subsumes the first (by being more specific) and therefore may satisfy both goals.

### Unpredictable Interaction

As discussed earlier, it's very difficult in this domain to predict all effects or interactions of modification operators. As a result, it's not always possible to predict conflict or synergy. Carrying out a modification and checking the resulting new design for goal satisfaction, however, may reveal a conflict or synergy. We've called this situation *unpredictable interaction.*

3. Recall that `<edge: 111.06...>` represents the sight line between centers.

Consider the two goals:[4]

```
<goal: (fireplace-on-interior-edge of Living) true>
<goal: (visually-open Living from Dining) true>,
```

and a suggestion for satisfying the first goal:

```
(move Fireplace1 to <edge 14.11...>).
```

It's difficult to predict how moving the fireplace will affect the visual openness. Unlike the previous example, which relied on a simple counting routine to detect conflict, this example involves a more complicated computational geometry routine that requires a specific arrangement of design elements. In order to calculate a visual openness value, the design elements must be in their designated locations. In this example, rather than trying to predict how moving the fireplace to a new location will affect visual openness, it's easier to move the fireplace, then check the resulting new design. It turns out that moving the fireplace to the interior edge in this example blocks the view and decreases the visual openness between the Living and Dining territories, thereby conflicting with the goal of having the Living territory visually open from the Dining territory. This situation exhibits an *unpredictable conflict.*

Synergy can be unpredictable as well. Consider goals for a fireplace on an exterior edge and a visually open Living territory:

```
<goal: (fireplace-on-exterior-edge of Living) true>
<goal: (visually-open Living from Dining) true>,
```

and a suggestion for satisfying the fireplace location goal:

```
(move Fireplace1 to <edge 101.33...>).
```

The suggestion, which moves the fireplace from between the two territories to an exterior edge, may unblock the view between the Living and Dining territories, thereby causing the Living territory to be visually open from the Dining territory. This suggestion satisfies the first goal, as intended, but may also inadvertently satisfy the second goal. We've called this situation *unpredictable synergy.* As with unpredictable conflict, TAC does not attempt to predict this interaction because carrying out the suggestion and checking the resulting design is easier.

A final note: TAC does not resolve conflicts between goals. When no solutions are found for a design problem, TAC stops. As mentioned Chapter 10, later versions of TAC could try standard conflict resolution techniques, e.g. modifying goal parameters or leaving conflicting goals out of the goal set.

---

4. For readability, the `on-interior-edge` design characteristic, discussed in Section 5.4, has been replaced by `fireplace-on-interior-edge`. The expression `(fireplace-on-interior-edge in Living)` is equivalent to `(on-interior-edge fireplace Living)`.

## 7.3 The Control Structures

### 7.3.1 Sequential-with-Lookahead

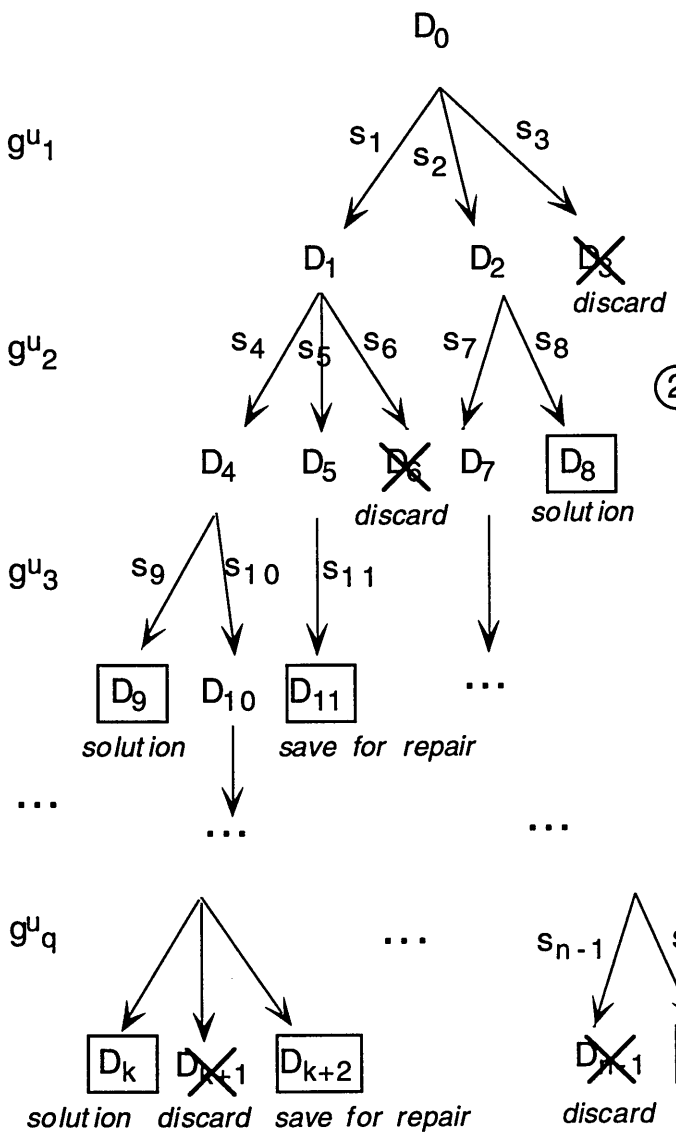
In the sequential-with-lookahead control structure, TAC proposes repair suggestions for a single goal at a time, refines those suggestions by checking them for interactions with all other goals, then creates new designs for the refined suggestions. It prunes some of the new designs created for one goal (e.g. designs that don't satisfy the goal), then uses the remaining designs as starting points for satisfying the next goal, and so on. After TAC has made one pass through the goals, it attempts to repair any designs that do not satisfy all goals: for each design, it starts a new cycle of proposing, refining, and carrying out repair suggestions.

TAC controls its search for design solutions by using a lookahead mechanism to spot potential interactions among repair suggestions prior to creating new designs. It checks first for interactions at the highest level of abstraction, among value suggestions, which propose satisfying a goal by changing a design characteristic's value (e.g. by increasing) or keeping a value the same. It then checks for interactions among design suggestions, which propose design modifications in terms of categories of design elements. Finally, TAC checks for interactions at the lowest level of abstraction, among design element suggestions, which propose design modifications in terms of particular design elements.

The sequential-with-lookahead control structure is summarized in Figure 7.1; the lookahead mechanism is summarized in Figure 7.2. A description and examples of how the control structure and lookahead mechanism work follow the figures.

A final note: TAC rarely detected interactions at the design suggestion level, so the discussion that follows in this and subsequent sections focuses on value and design element suggestions. The figures also reflect this focus.

Given design $D_0$, goals $G = G^u \cup G^s = \{g^u_1\ g^u_2\ g^u_3 \ldots\} \cup \{g^s_1\ g^s_2\ g^s_3 \ldots\}$:




① Starting with $D_0$, propose suggestions $s_i$ for $g_{u1}$ and create new designs. Check designs for uniqueness, satisfied $g_{u1}$, satisfied $G^u$, satisfied G. Save as solutions those that satisfy G; save for repair those that satisfy $G^u$; discard those that are not unique or do not satisfy $g^u_1$.

② Starting with new designs $D_1$, $D_2$ that have not been saved or discarded, propose suggestions for $g^u_2$ and create new designs. Save or discard designs as in step above (checking $g^u_2$ satisfaction instead of $g^u_1$).

③ For each of remaining $g^u_i$, continue proposing suggestions, creating and checking new designs. With last goal $g^u_q$, save designs for repair that satisfy $g^u_q$ (rather than $G^u$).

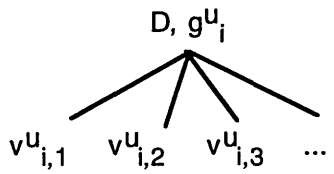
④ After new designs have been created for last goal $g^u_q$ and checked as above, repair designs that were saved for repair.

⑤ After all designs have been repaired, return solutions.

**Figure 7.1:** Sequential control structure; lookahead is used to propose suggestions in step 1.

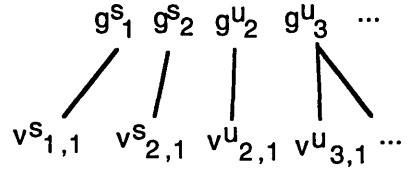
Given design D, goals $G = G^u \cup G^s = \{g^u_1\ g^u_2\ g^u_3 \ldots\} \cup \{g^s_1\ g^s_2\ g^s_3 \ldots\}$, and goal $g^u_i$:

1. Propose value $v^u_{i,n}$ suggestions for $g^u_i$:



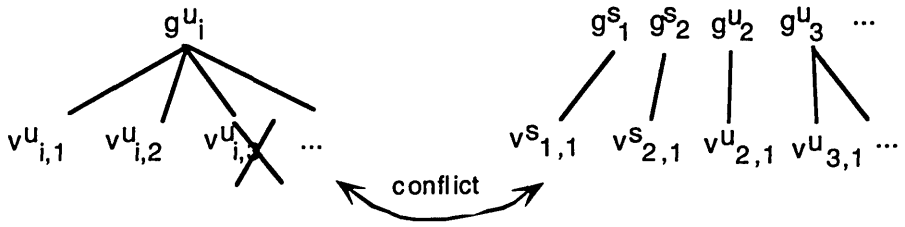

2. Propose value suggestions $v^x_{j,k}$ for each goal in $G - g^u_1 = \{g^s_1\ g^s_2\ g^u_2\ g^u_3 \ldots\}$:

   (Note: $v^s_{j,k}$ are keep-value suggestions; $v^u_{j,k}$ are increase-, decrease-, or set-value suggestions.)

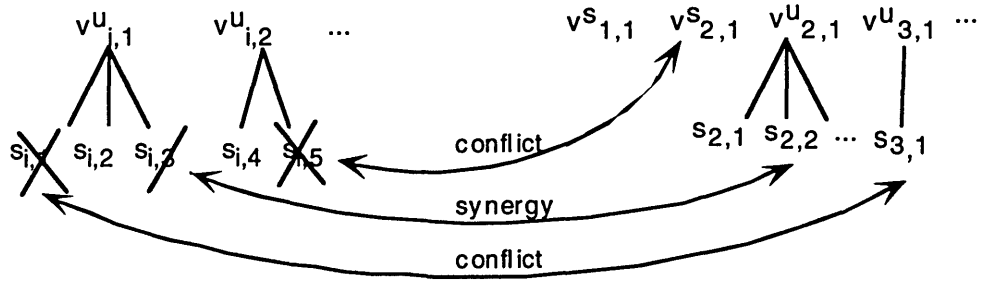



3. For each value suggestion $v^u_{i,n}$:

   . check with each $v^x_{j,k}$ for conflict, or synergy with more specific suggestion; if conflict found, prune $v^u_{i,n}$; if synergy found, replace $v^u_{i,n}$.




4. Propose design element suggestions $s_{i,o}$ for value suggestions $v^u_{i,n}$ (for goal $g^u_i$).
   Propose design element suggestions $s_{j,q}$ for value suggestions $v^u_{j,k}$ (for $G - g^u_i$).
   For each $s_{i,o}$:
   - check with each $v^x_{j,k}$ for conflict; if conflict found, prune $s_{i,o}$.
   - check with each $s_{j,q}$ for synergy with more specific suggestion; if synergy found, replace $s_{i,o}$ with $s_{j,q}$.
   - check with each $s_{j,q}$ for conflict; if conflict found and $s_{j,q}$ is only suggestion for $g^u_j$, prune $s_{i,o}$.




5. Result is set of design element suggestions for goal $g^u_i$: $\{s_{i,2}\ s_{2,2}\ s_{i,4} \ldots\}$

**Figure 7.2:** Suggestion proposal using lookahead.

TAC starts with a design and a set of goals G, and determines which of the goals are satisfied (call this set $G^S$) and which are unsatisfied (call this set $G^U$). As shown in Figure 7.1, for each unsatisfied goal, TAC proposes design element suggestions $s_i$ using the lookahead routine shown in Figure 7.2. It creates one new design per suggestion, checking whether it should:

(a) save the design as a solution, e.g. $D_8$ and $D_9$ in Figure 7.1;

(b) discard the design (because it's been seen before or its intended goal wasn't satisfied), e.g. $D_3$ and $D_6$;

(c) skip modifications for the rest of the unsatisfied goals and save the design for repair (because unsatisfied goals are now satisfied, but some formerly satisfied goals are now unsatisfied), e.g. $D_{11}$;

(d) continue attempting to satisfy each of the remaining unsatisfied goals.

Examples below illustrate each of these situations.

After iteration through the unsatisfied goals, TAC has three sets of designs: those that are solutions, those that have been discarded, and those that are to be repaired. For each design in this last set, TAC enters a repair cycle: it creates a new design problem and evaluates goals, suggests repairs, and creates new designs. Finally, after TAC has finished repairing designs, it returns the original design problem, the set of design solutions, and the design problems created in its search for solutions. The suggestions and new designs created in the search are stored in the design problems.

As shown in Figure 7.2, TAC uses a lookahead routine to refine suggestions for each unsatisfied goal. The routine checks for interactions among and across all three types of suggestions. Given a design and an unsatisfied goal, TAC proposes value suggestions for that goal, then checks those suggestions against value suggestions for the other goals, pruning when it detects conflict or synergy. It then proposes design and design element suggestions, checking them for potential conflict or synergy interactions with value, design, and design element suggestions for the other goals. If TAC detects any interaction, it again prunes the set of suggestions for the unsatisfied goal. Design element suggestions for the unsatisfied goal may be pruned further by a lookbehind routine, which checks these suggestions against the suggestions already carried out to yield the design. If a proposed suggestion is the same as one that has already been carried out for the design, then TAC prunes it. An experiment illustrating this lookbehind mechanism is discussed in Section 7.6.

The examples below, taken from the previous section, illustrate the sequential-with-lookahead control structure.

Example 1: Predictable conflict

Let's say that we want the Living territory in the Tomek house to be visually open from the Dining territory, and we want one fireplace in the Living territory. We specify two goals:

```
<goal: (visually-open Living from Dining) true>
<goal: (fireplace-count in Living) 1>
```

TAC determines that the Living territory is not visually open from the Dining territory, but it does contain one fireplace. It proposes a suggestion for the unsatisfied `visually-open` goal:

```
(increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6)
```

It then performs lookahead by proposing a suggestion for the remaining `fireplace-count` goal:

```
(keep-value of (fireplace-count in Living) 1)
```

TAC detects no interaction between these two suggestions, so it proceeds by checking its knowledge base for ways to increase visual openness. It finds that removing or puncturing design elements that block the view between Living and Dining territories may increase visual openness, so it proposes:[5]

```
(or (remove blocking-elements-btw Living and Dining)
    (puncture blocking-elements-btw Living and Dining))
```

Checking the design, it finds that the fireplace blocks the view, so it more specifically proposes:

```
(or (remove Fireplace1)
    (puncture Fireplace1))
```

TAC then performs lookahead to check for conflict or synergy between these suggestions and the lookahead suggestion, which proposes keeping the fireplace count at one. It spots a conflict with the first suggestion: as discussed in the previous section, removing the fireplace changes the number of fireplaces in the Living territory to zero, which conflicts with keeping the number of fireplaces at one. TAC prunes the remove suggestion, then checks the puncture suggestion. It spots no interaction between this suggestion and the lookahead suggestion, so it leaves the puncture suggestion as is. The lookahead comparisons are summarized in Figure 7.3.

5. The screenify suggestion has been left out in order to simplify this and subsequent examples.

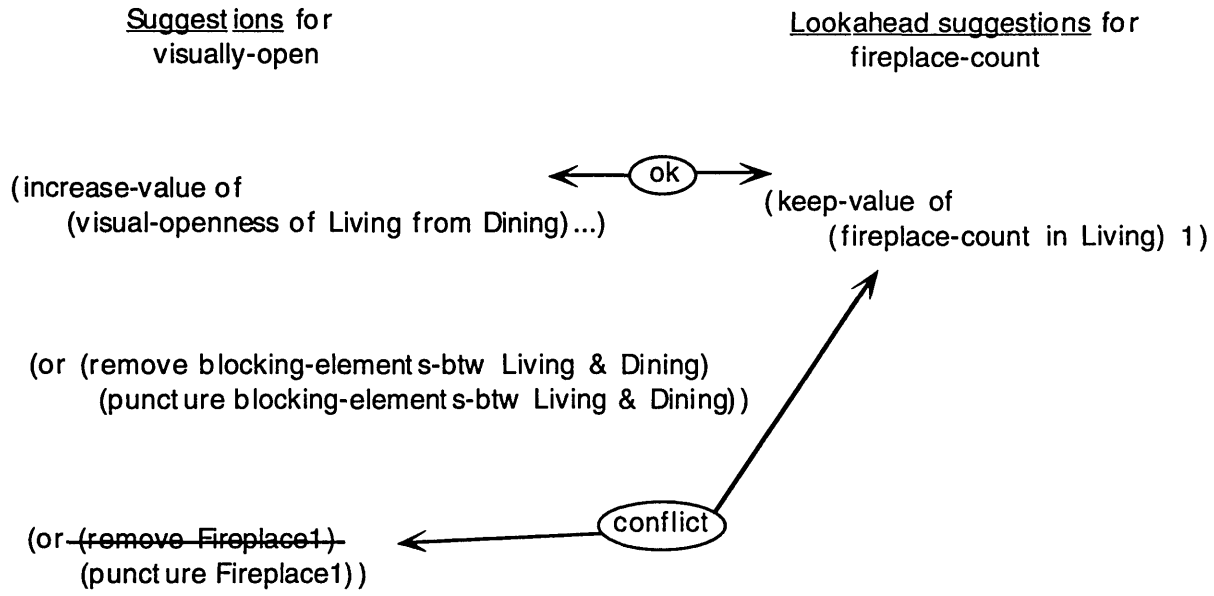



**Figure 7.3:** Lookahead results for `visually-open` and `fireplace-count` goals.

TAC carries out the puncture suggestion, creating a new design Tomek#1 which has the fireplace punctured at a default location (through the middle). TAC checks that the intended goal for the suggested modification is satisfied: it checks that the Living territory in Tomek#1 is visually open from the Dining territory. It finds this true, so TAC continues.

Had the design not satisfied the intended goal, TAC would have discarded the design. It would not have attempted to repair the design because it assumes that if a modification doesn't do what it is supposed to do, the resulting design isn't any closer to a solution than the original design. (As we'll see, TAC repairs designs when an intended goal has been satisfied, but other goals remain unsatisfied.)

Figure 7.4 shows Tomek#1 and the region of the Living territory visible from the Dining territory. The `visual-openness` has increased to 0.61 from the original value of 0.44.

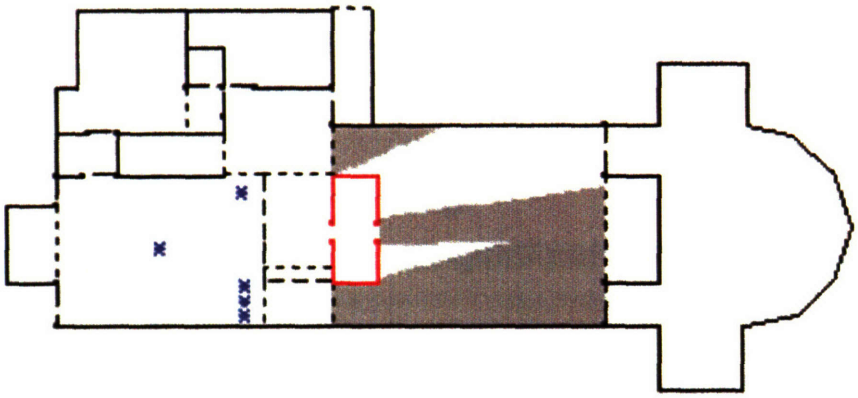


**Figure 7.4:** Tomek#1 and region of Living territory visible from Dining territory.[6] Shaded region is visible; each * represents a viewpoint.

Having generated the new design Tomek#1 for the only unsatisfied goal (the `visually-open` goal), TAC now checks this design to see if the previously satisfied goal—having one fireplace in the Living territory—has been clobbered by the visual openness modification. The goal is still satisfied, so Tomek#1 is a solution. TAC stops at this point because it has generated designs for all unsatisfied goals (one goal, in this case) and has no designs to repair.

Example 2: Predictable synergy

Returning again to a previous example, let's say that we want the Living territory to be visually open from the Dining territory in the Tomek house, and we want the center of the Living territory to be visible from the center of the Dining territory. We specify the two goals:

```
<goal: (visually-open Living from Dining) true>
<goal: (visible-center Living from Dining) true>
```

TAC determines that both goals are unsatisfied. It proposes a suggestion for the first goal:

```
(increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6)
```

It then performs lookahead, comparing this suggestion with a value suggestion for the remaining `visible-center` goal:

```
(set-value of (visible-from (center of Living) (center of Dining))
              to true)
```

TAC finds no interaction, so it continues. Checking its knowledge base, it again finds that visual openness may be increased by removing or puncturing design elements that block the view.

---

6. The 2D projection of the fireplace puncture has been used in the visual openness calculation.

It proposes these suggestions:

```
(or (remove blocking-elements-btw Living and Dining)
    (puncture blocking-elements-btw Living and Dining))
```

Finding that the fireplace blocks the view, it then proposes more specifically:

```
(or (remove Fireplace1>)
    (puncture Fireplace1))
```

TAC then performs lookahead by checking the above suggestions against suggestions for the `visible-center` goal. For this goal, TAC finds that it can make the center of the Living territory visible from the center of the Dining territory by removing or puncturing design elements along the line of sight between the two territories. It proposes the following suggestions:

```
(or (remove blocking-elements-btw (center of Living) and (center of Dining))
    (puncture blocking-elements-btw (center of Living) and (center of Dining)
              at sight-line-btw centers))
```

Substituting the blocking element, the fireplace, into the above suggestions, TAC proposes:

```
(or (remove Fireplace1>)
    (puncture Fireplace1 at <edge: 111.06...>))
```

As noted earlier, there are synergies between suggestions for the two goals. Removing the fireplace may satisfy both goals; TAC leaves this suggestion as is. More interestingly, puncturing the fireplace at a particular location (for the `visible-center` goal) is similar to, but more specific than, puncturing the fireplace at a default location (for the `visually-open` goal). TAC chooses the more specific suggestion, using a unification-like expression matching routine to detect subsumption.

The lookahead comparisons are summarized in Figure 7.5.

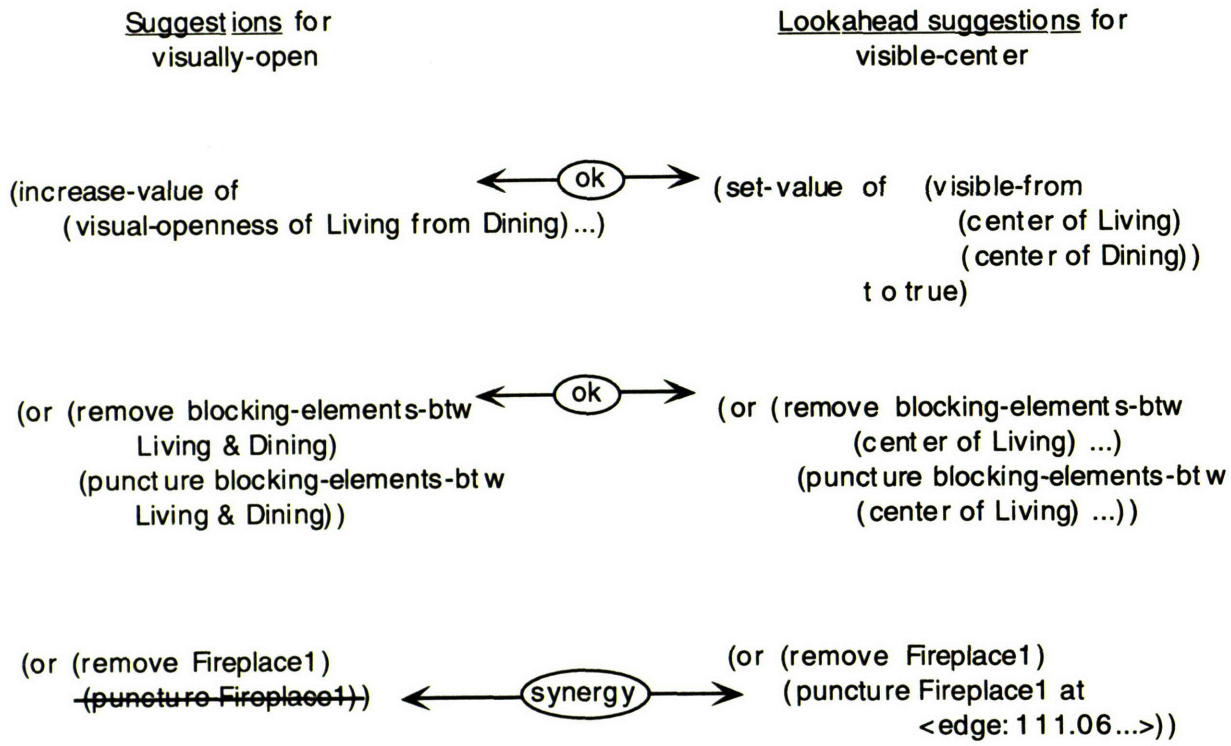



**Figure 7.5:** Lookahead results for `visually-open` and `visible-center` goals.

After lookahead, TAC creates a new design for each of its two suggestions:

`(remove Fireplace1)` → Tomek#1, with the fireplace removed;

`(puncture Fireplace1 at <edge: 111.06...>))` → Tomek#2, with the fireplace punctured along the sight line between territory centers.

Territory models for the new designs are shown below.

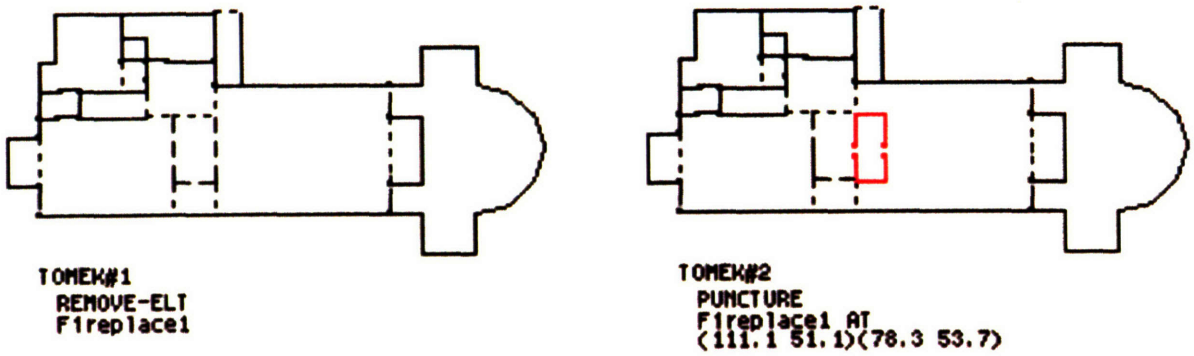



**Figure 7.6:** New designs for Tomek `visually-open` and `visible-center` goals.

TAC checks that the intended goal for each modification is satisfied: it finds that the Living territory in each design is indeed visually open from the Dining territory. It then continues, using each of these designs as a starting point for evaluating, suggesting repairs, and creating new designs for the next unsatisfied goal, namely, the `visible-center` goal.

For Tomek#1, TAC finds that the `visible-center` goal is now satisfied, so it proposes no suggestions. Removing the fireplace caused both unsatisfied goals to become satisfied (not too surprisingly, since the lookahead mechanism proposed that suggestion for both goals). For Tomek#2, TAC also finds the goal satisfied, so again proposes no suggestions. Thus, Tomek#1 and Tomek#2 are both solutions. TAC now has generated new designs for all unsatisfied goals, and since it has no designs to repair, it stops.

Example 3: Unpredictable conflict

Let's return to the another Frank Lloyd Wright Prairie house, the Horner house, which was introduced in Section 5.4 and is shown again below.

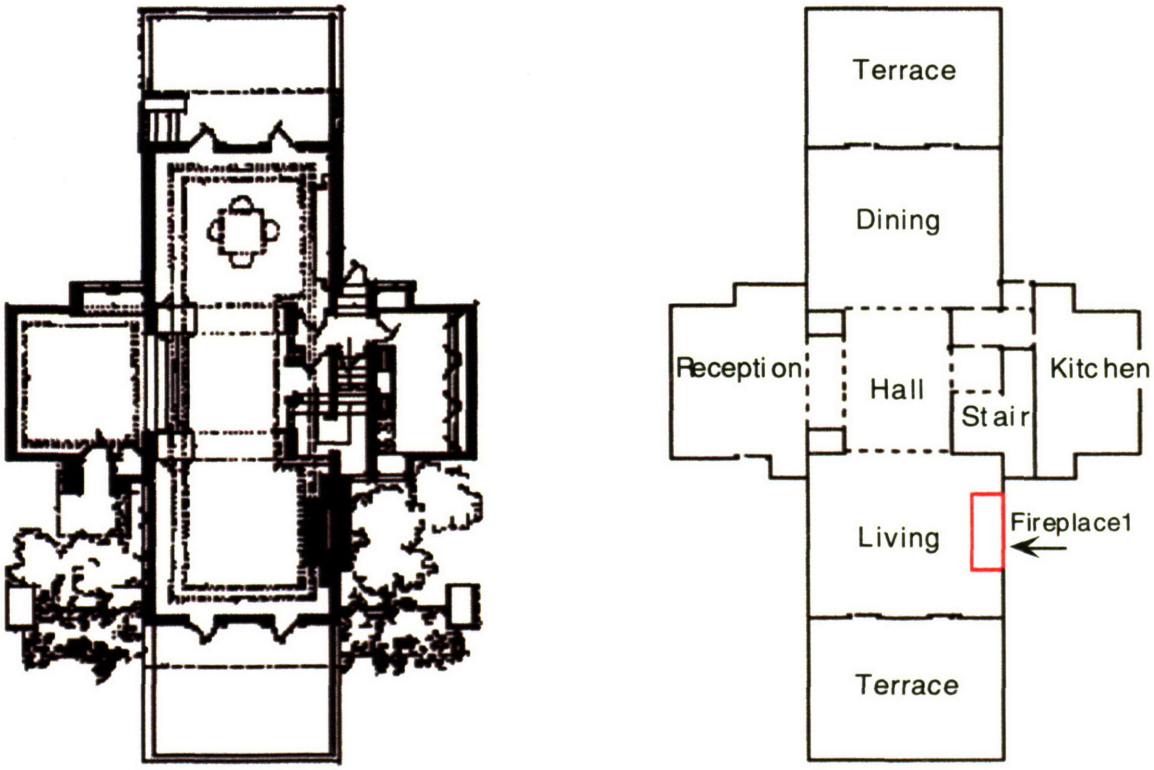



**Figure 7.7:** Floorplan and territory model for Horner house main (first) floor.

We specify two goals that are often satisfied in Frank Lloyd Wright's Prairie houses: a fireplace on an interior edge of the Living territory, and the Living territory visually open from the Dining territory.

```
<goal: (fireplace-on-interior-edge of Living) true>
<goal: (visually-open Living from Dining) true>
```

TAC determines that the visually open goal is satisfied, but that the house does not have a fireplace on an interior edge. TAC proposes two suggestions for the unsatisfied goal: make Fireplace1 be on an interior edge (so that the design characteristic `fireplace-on-interior-edge` is true), or increase the number of fireplaces by adding one to an interior edge.

```
(or (set-value of (on-interior-edge Fireplace1)
                  to true)
    (increase-value of (fireplace-count in Living)
                       to 2 such that on-interior-edge is true))
```

TAC checks these suggestions against a suggestion proposed for the remaining `visually-open` goal:

```
(keep-value of (visual-openness Living from Dining greater than 0.6)
               true)
```

It detects no interaction so it continues, checking its knowledge base for ways to satisfy the `fireplace-on-interior-edge` goal. It finds that it can either move the existing fireplace to an interior edge or add a new fireplace to an interior edge:

```
(or (move Fireplace1 to any interior-edges-for Fireplace1)
    (add fireplace to Living such that on-interior-edge))
```

It then identifies the interior edges in the Living territory and proposes:[7]

```
(or (move Fireplace1 to <edge: 18.75...>)
    (move Fireplace1 to <edge: 15.75...>)
    (move Fireplace1 to <edge: 29.06...>)
    (move Fireplace1 to <edge: 31.69...>)
    (move Fireplace1 to <edge: 15.75...>)
    (add fireplace to Living such that on-interior-edge))
```

TAC checks these suggestions for conflict or synergy with the suggestion to keep the visual openness value greater than 0.6. It finds no interaction, so it carries out the suggestions and creates six new designs, shown in Figure 7.8.

---

7. The `add` routine is opaque, so this design suggestion is not expanded further to specify a particular location. The current version of the `add` routine produces one new design, rather than all possible designs.

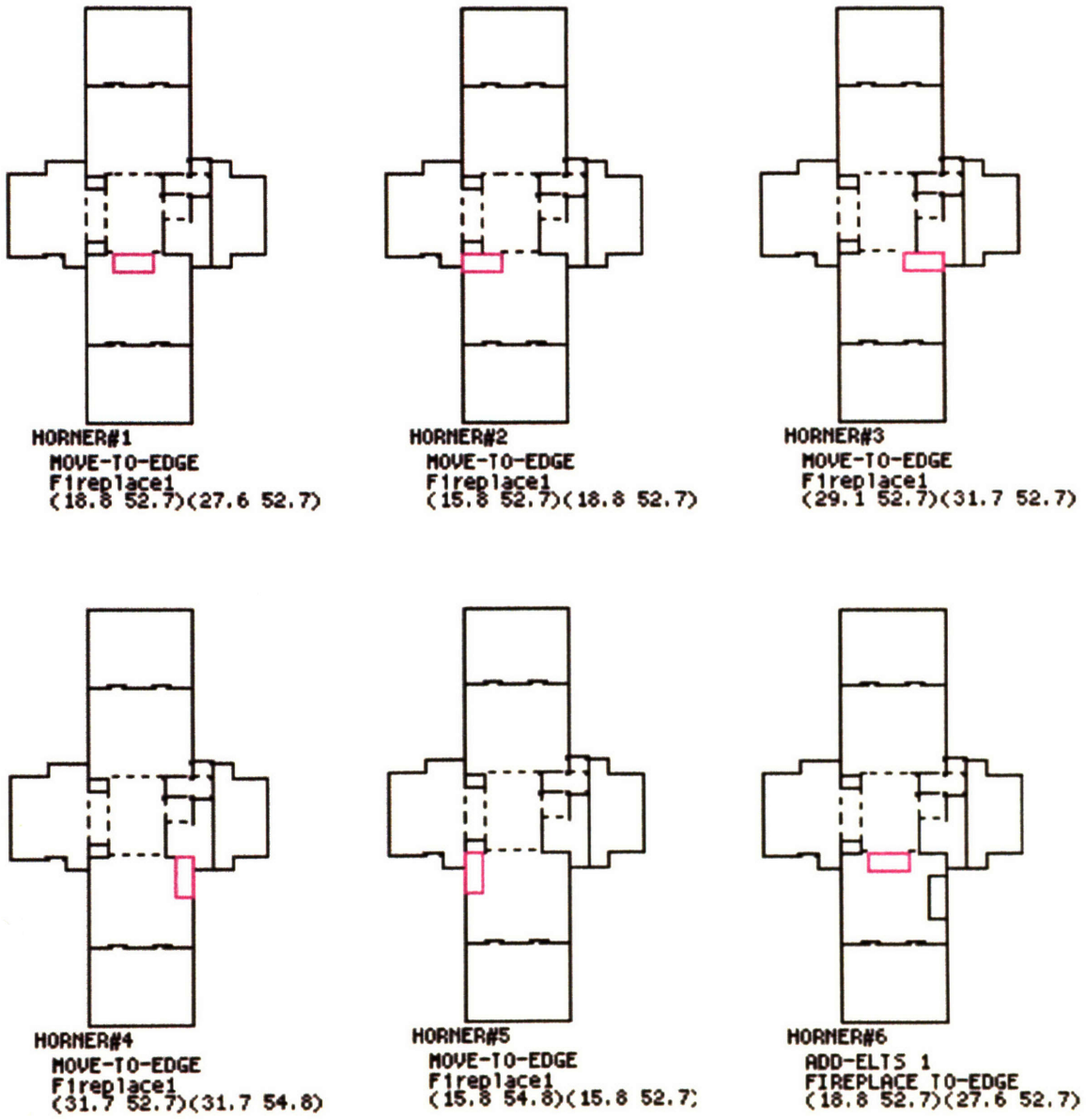



**Figure 7.8:** Five designs with fireplace moved to interior edge, one with new fireplace added.

The intended goal, having a fireplace on an interior edge, is satisfied for all designs except Horner#4 and Horner#5. (A less stringent definition might have allowed these configurations which have the back of the fireplace on both an interior edge and an exterior edge.) These two designs are discarded, and the remaining designs are checked to make sure the `visually-open` goal is still satisfied. The Living territory is still visually open from the Dining territory in Horner#2 and Horner#3, so these two designs are solutions. The Living territory is not visually open in Horner#1 and Horner#6. The modifications have clobbered the `visually-open` goal, so these

two designs must be repaired. In one of these designs, Horner#1 shown below, the `visual-openness` value is 0.5; we'd like for it to be greater than 0.6.

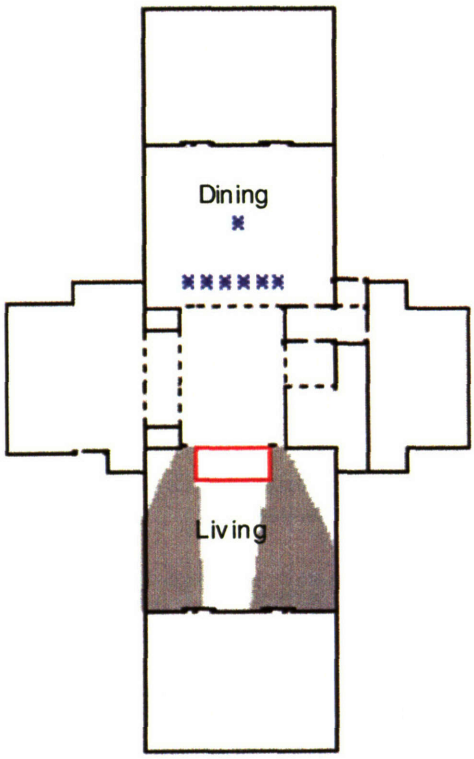



**Figure 7.9:** Horner#1: one of the designs to be repaired.
Shaded region is visible; * represents a viewpoint.

To repair Horner#1, TAC proposes increasing the visual openness:

```
(increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6)
```

It then performs lookahead by proposing a suggestion for the remaining `fireplace-on-interior-edge` goal. It proposes that some fireplace be kept on an interior edge of the Living territory:

```
(keep-value of (some fireplace in Living on-interior-edge)
               true)
```

TAC spots no interaction between these two suggestions, so it continues, proposing suggestions for increasing visual openness:

```
(or (remove blocking-elements-btw Living and Dining)
    (puncture blocking-elements-btw Living and Dining))
```

Checking the design, TAC finds that three design elements—the fireplace, a bookcase, the stair—block the view, so it proposes:[8]

```
(or (remove Fireplace1)
    (remove Bookcase1)
    (remove Stair)
    (puncture Fireplace1)
    (puncture Stair)
    (and (remove Fireplace1) (remove Stair))
    (and (puncture Fireplace1) (puncture Stair)))
```

TAC checks these suggestions against the lookahead suggestion, which proposes keeping a fireplace on an interior edge. TAC notices that removing the fireplace would conflict, since there would be no fireplace at all. It prunes that suggestion and is then left with:

```
(or (remove Bookcase1)
    (remove Stair)
    (puncture Fireplace1)
    (puncture Stair)
    (and (puncture Fireplace1) (puncture Stair)))
```

The lookahead comparisons are summarized in Figure 7.10.

8. TAC does not propose all possible combinations or orderings of modifications. The conjunctions in this example are the result of TAC's finding that regions of the Living territory are blocked by more than one element.

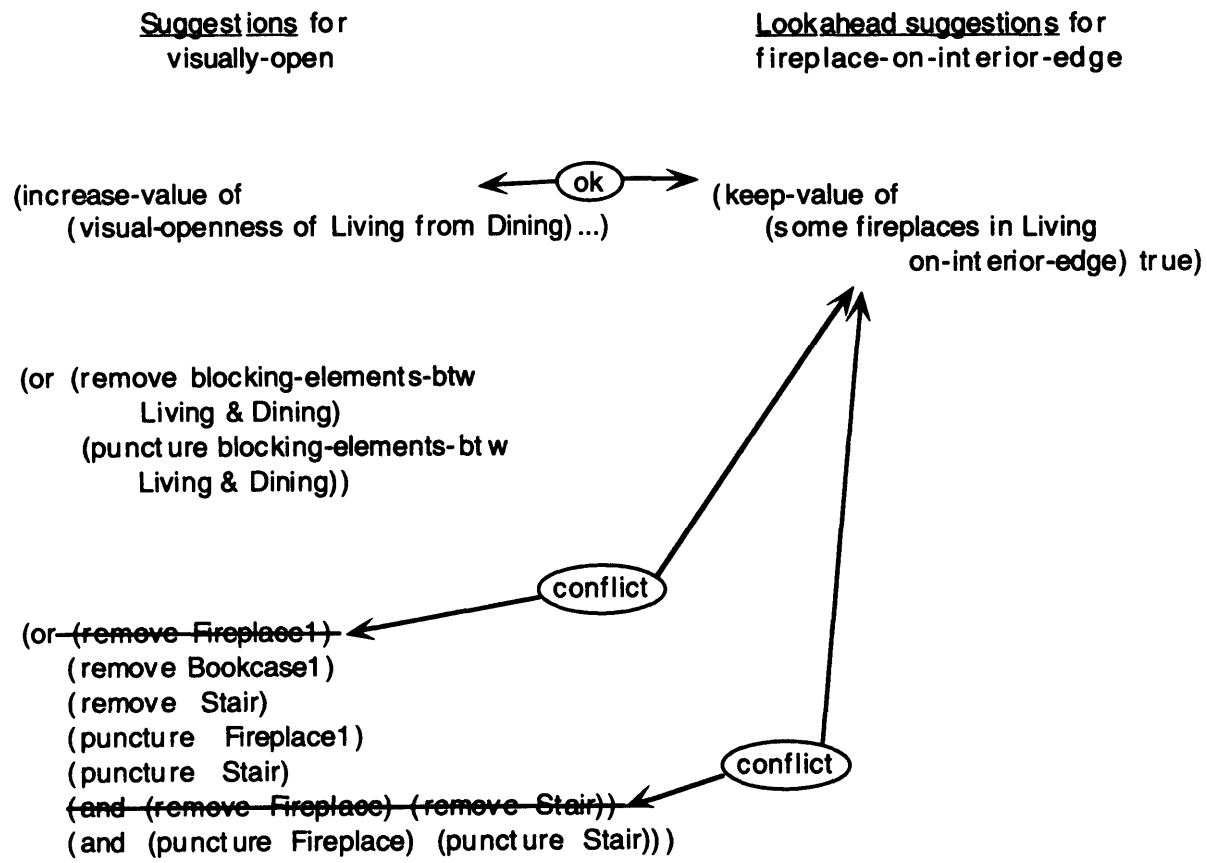



**Figure 7.10:** Lookahead results for `visually-open` and `fireplace-on-interior-edge` goals.

TAC carries out the five suggestions that remain after pruning. The first four suggestions—removing the bookcase, removing the stair, puncturing the fireplace, puncturing the stair—yield solutions. The last suggestion results in a duplicate design, which is discarded: TAC carries out that suggestion's first modification (puncturing the fireplace), finds the intended `visually-open` goal satisfied, so it doesn't carry out the second modification. As a result, the design is the same as the one produced by the third suggestion and is discarded. The issue of sameness is a complicated one, and we've chosen a strict definition: Two designs are the same if they contain the same edges and the same design elements. The more general and difficult problem of determining what "same-as" means for architectural designs is a research question in itself. (See Section 7.4.1 for further discussion.)

At this point, Horner#1 has been repaired, and TAC has found four solutions to add to its previous two. The four new solutions are shown in Figure 7.11. One of the solutions, along with its improved visual openness value, is shown in Figure 7.12.

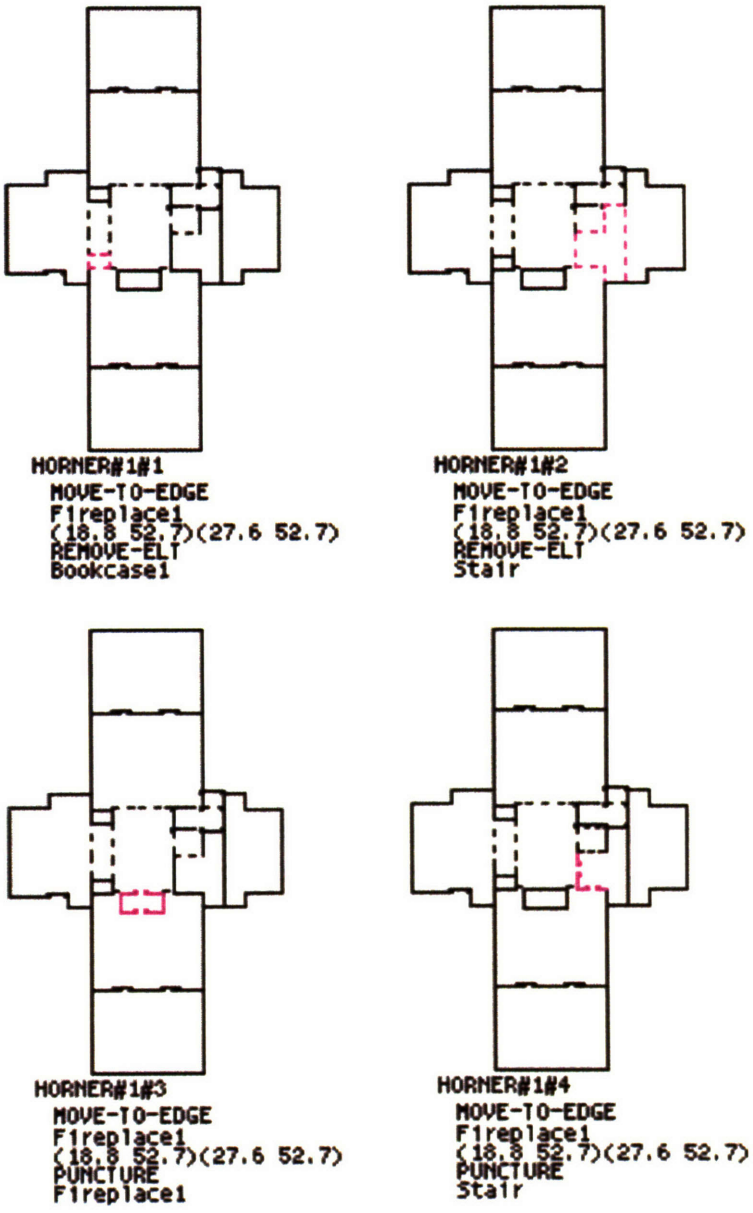



**Figure 7.11:** Repair of Horner#1 results in four new designs which are solutions.

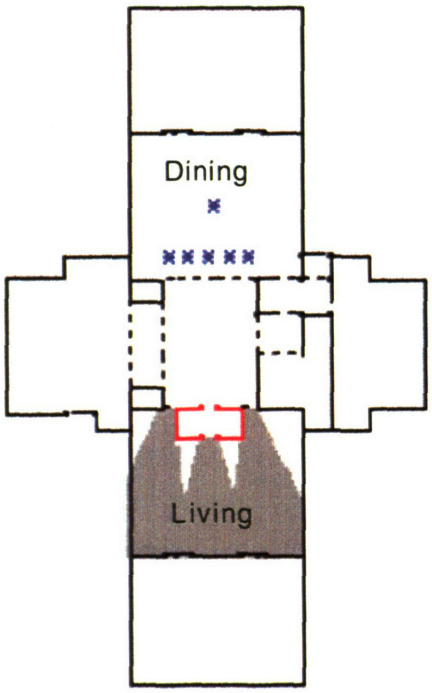



**Figure 7.12:** Horner#1#3, with fireplace moved to interior edge and punctured. Shaded region of Living territory is visible from Dining territory. The visual openness value is 0.8.

Repair of Horner#6 proceeds similarly. Recall that Horner#6 has two fireplaces, Fireplace1 in the original location on an exterior wall, and Fireplace2 which was added to an interior edge.

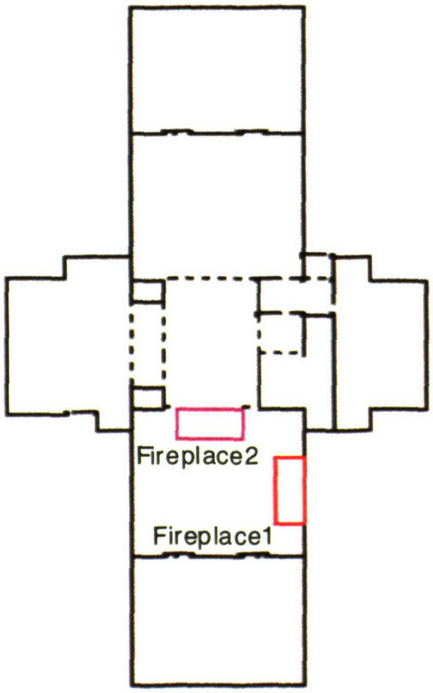



**Figure 7.13:** Horner#6: a new fireplace has been added to Horner.

Horner#6 has a fireplace on an interior edge, but the Living territory is no longer visually open from the Dining territory. TAC proposes increasing the visual openness:

```
(increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6)
```

It performs lookahead by comparing the suggestion above with a suggestion for the remaining `fireplace-on-interior-edge` goal:

```
(keep-value of (some fireplace in Living on-interior-edge)
               true)
```

TAC spots no interaction, so it proposes increasing visual openness by removing or puncturing the bookcase, the stair, or Fireplace2:

```
(or (remove Fireplace2)
    (remove Bookcase1)
    (remove Stair)
    (puncture Fireplace2)
    (puncture Stair)
    (and (remove Fireplace2) (remove Stair))
    (and (puncture Fireplace2) (puncture Stair)))
```

TAC checks each of these suggestions against the lookahead suggestion, which proposes keeping a fireplace on an interior edge, and does not spot conflicts. The first suggestion, to remove Fireplace2, will indeed cause the design to no longer have a fireplace on an interior edge, but this conflict cannot be spotted simply by comparing suggestions; the design must be checked. Since the lookahead mechanism reasons about suggestions, rather than checking the design, it leaves the remove suggestion as is.

TAC creates one new design for each suggestion. The first suggestion results in a design identical to the starting design, so it is discarded. The next four suggestions yield solutions. The last two suggestions result in designs that are discarded because they are identical to two of the solutions.[9]

At this point, TAC has no more designs to repair and returns its ten solutions.

As a final note, if we add a design goal stipulating that we want only one fireplace in the Living territory, TAC's lookahead mechanism prunes the suggestion to add a fireplace. Horner#6 and its four solutions are not created, and TAC returns six solutions.

---

9. As when repairing Horner#1, the first modification in each conjunction satisfied the intended `visually-open` goal, so the second modification was unnecessary.

Example 4: Unpredictable synergy

Let's return to a design discussed in the previous example, Horner#1, shown in Figure 7.9, and again below.

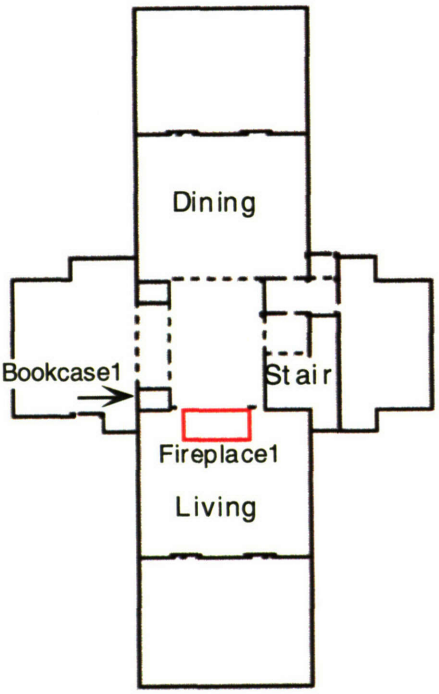



**Figure 7.14:** Territory model for Horner#1.

Let's say that we're interested in having one fireplace, a fireplace on an exterior edge, and the Living territory visually open from the Dining territory. We specify the goals:

```
<goal: (fireplace-count in Living) 1>
<goal: (fireplace-on-exterior-edge of Living) true>
<goal: (visually-open Living from Dining>) true>
```

TAC finds that the last two goals are unsatisfied. It proposes value suggestions for the first unsatisfied goal, having a fireplace on an exterior edge: make Fireplace1 be on an exterior edge or increase the number of fireplaces by adding one to an exterior edge.

```
(or (set-value of (on-exterior-edge Fireplace1)
                  to true)
    (increase-value of (fireplace-count in Living)
                       to 2 such that on-exterior-edge is true))
```

To perform lookahead, TAC proposes suggestions for the two remaining goals:

```
(and (keep-value of (fireplace-count in Living) 1)
     (increase-value of (visual-openness of Living from Dining)
                  until visual-openness greater than 0.6))
```

It checks the `fireplace-on-exterior-edge` suggestions against these lookahead suggestions and spots a conflict: The second `fireplace-on-exterior-edge` suggestion increases the fireplace count, conflicting with keeping the number of fireplaces at one, so it is pruned. The suggestion to make Fireplace1 be on an exterior edge remains:

```
(set-value of (on-exterior-edge Fireplace1)
              to true)
```

TAC checks its knowledge base and finds that it can set the value of the `on-exterior-edge` characteristic to true for a design element by moving that element to an exterior edge. For Fireplace1, it proposes:

```
(move Fireplace1 to any exterior-edges-for Fireplace1)
```

Checking the design, it finds two exterior edges appropriate for Fireplace1, and proposes: [10]

```
(or (move Fireplace1 to <edge: (15.75...>)
    (move Fireplace1 to <edge: (31.69...>))
```

TAC then returns to its lookahead suggestions—keeping the fireplace count at one and increasing visual openness. For the second of these suggestions, it further proposes:

```
(or (remove blocking-elements-btw Living and Dining)
    (puncture blocking-elements-btw Living and Dining))
```

It then checks the design, and proposes:

```
(or (remove Fireplace1)
    (remove Stair)
    (remove Bookcase1)
    (puncture Fireplace1)
    (puncture Stair)
    (and (remove Fireplace1) (remove Stair))
    (and (puncture Fireplace1) (puncture Stair)))
```

TAC checks the move suggestions proposed for satisfying the `fireplace-on-exterior-edge` goal against these lookahead suggestions. It does not find conflicts, so it leaves the move suggestions as they are.

Lookahead comparisons are summarized in Figure 7.15.

10. TAC was told to choose only edges that were long enough to accommodate the fireplace.

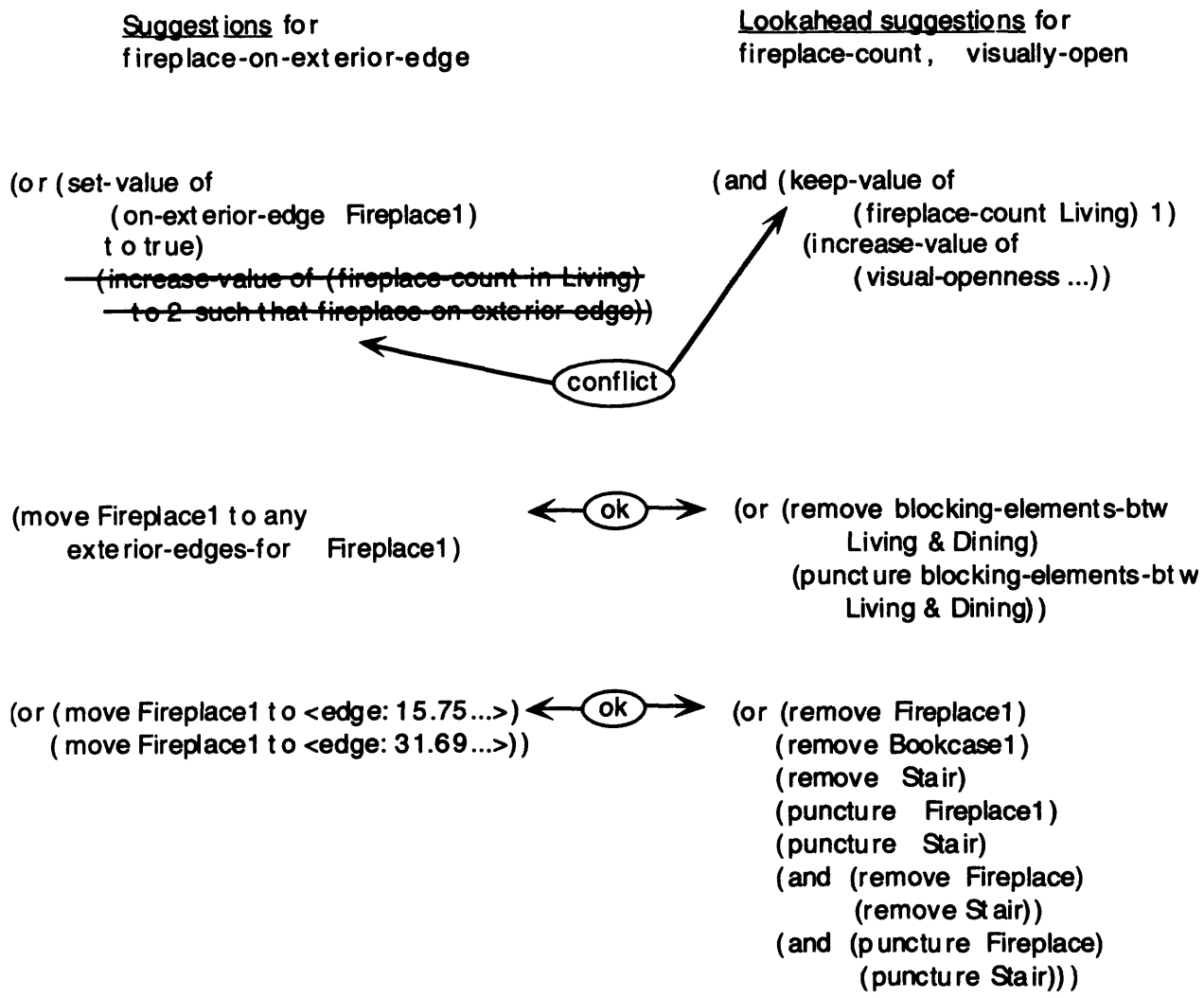



**Figure 7.15:** Lookahead results for `fireplace-on-exterior-edge` goal.

Notice that TAC does not need to check for conflict among the lookahead suggestions; e.g. it does not need to prune the suggestion for removing a fireplace, even though it looks as if that suggestion will conflict with keeping the number of fireplaces at one. The extra work is unnecessary because the lookahead suggestions are used only to check for interaction with the currently proposed suggestions, those for the `fireplace-on-exterior-edge` goal in this example. When suggestions are proposed for the `visually-open` goal in a later step, TAC will prune any conflicting suggestions at that point.

Figure 7.16 illustrates checking for conflicts between suggestions and lookahead suggestions in the example above. The `fireplace-on-exterior-edge` goal is $g^u_1$, the `fireplace-count` goal is $g^s_1$, and the `visually-open` goal is $g^u_2$.

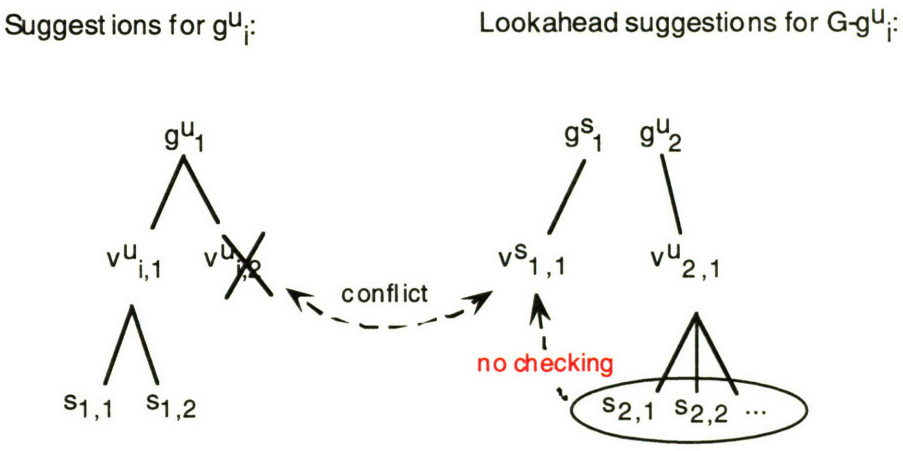



**Figure 7.16:** TAC checks for conflict between suggestions for current goal and other goals, not among lookahead suggestions for other goals. $g^u$ and $g^s$ are unsatisfied and satisfied goals; $v_x$ are value suggestions; $s_x$ are design element suggestions.

TAC now has two suggestions for satisfying the fireplace on exterior edge goal:

```
(or (move Fireplace1 to <edge: (15.75...>)
    (move Fireplace1 to <edge: (31.69...>))
```

It creates a new design for each suggestion (Figure 7.17).

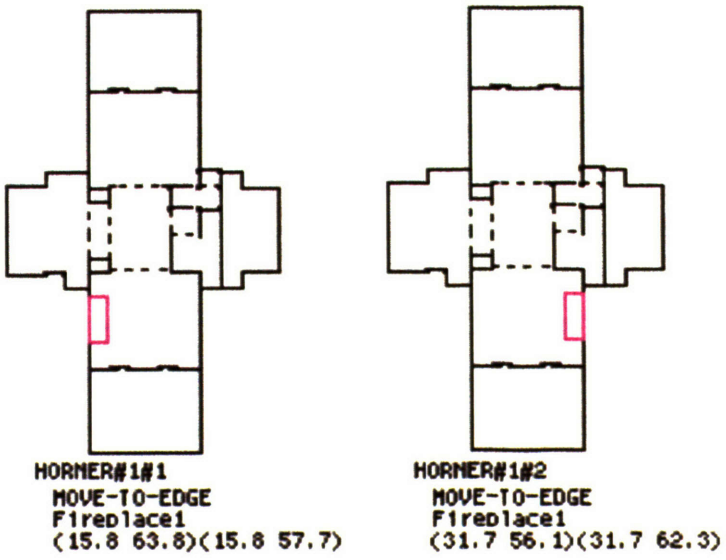



**Figure 7.17:** New designs with fireplace on exterior edge.

TAC uses these two designs as starting points for satisfying the second unsatisfied goal, having the Living territory visually open from the Dining territory. TAC checks that goal and finds it now satisfied, so it proposes no suggestions and generates no new designs. TAC has discovered a synergy: moving the fireplace to an exterior edge has also satisfied the `visually-open` goal. Discovered synergies could be added to TAC's knowledge base. In this example, moving an object to an exterior edge could be added as a means of increasing a `visual-openness` value (i.e. as an increaser on the design characteristic `visual-openness`[11]).

TAC has now generated designs for the unsatisfied goals and has no designs to repairs, so it returns the two solutions shown above.

Benefits and Costs

As the examples above have shown, the sequential-with-lookahead control structure works on a single goal at a time, proposing repair suggestions and creating designs as it searches breadth-first for design solutions. The design space is exponential in nature, but the lookahead mechanism helps control the search. As shown in Figure 7.1, the tree of designs has a branching factor (s) of the number of suggestions per goal, which is not a constant, but in practice averages about six. It has a depth (u) of the number of unsatisfied goals. In the worst case, the size of the tree would be $s^u$. The lookahead mechanism decreases the size, however, by pruning suggestions prior to design creation, decreasing both s and u. It decreases the number of suggestions (branching factor) by detecting conflict at various levels of abstraction; it decreases the number of unsatisfied goals (depth) by detecting synergy between suggestions. The lookahead mechanism also reduces exponential growth by pruning designs that are not unique, thereby eliminating duplicate portions of the tree that would be generated below them.

The sequential-with-lookahead control structure does have costs. Each new design repaired is the root of a new tree of designs. If the repair successfully accomplishes its goal, however, the new designs have fewer unsatisfied goals than their precursors, and the new trees are smaller than the previous ones. The sequential-with-lookahead control structure also incurs the cost of producing suggestions for each goal multiple times: suggestions for a goal are proposed once when that goal is the focus of suggestion proposal, and every time lookahead runs for the remaining goals. In the worst case, each formerly satisfied goal becomes unsatisfied in a new design, causing TAC to propose suggestions for all goals each time it does lookahead. In the best case, however, the number of unsatisfied goals decreases with each new design generated, thereby decreasing the number of times lookahead runs. In either case, suggestion proposal is a less costly operation than

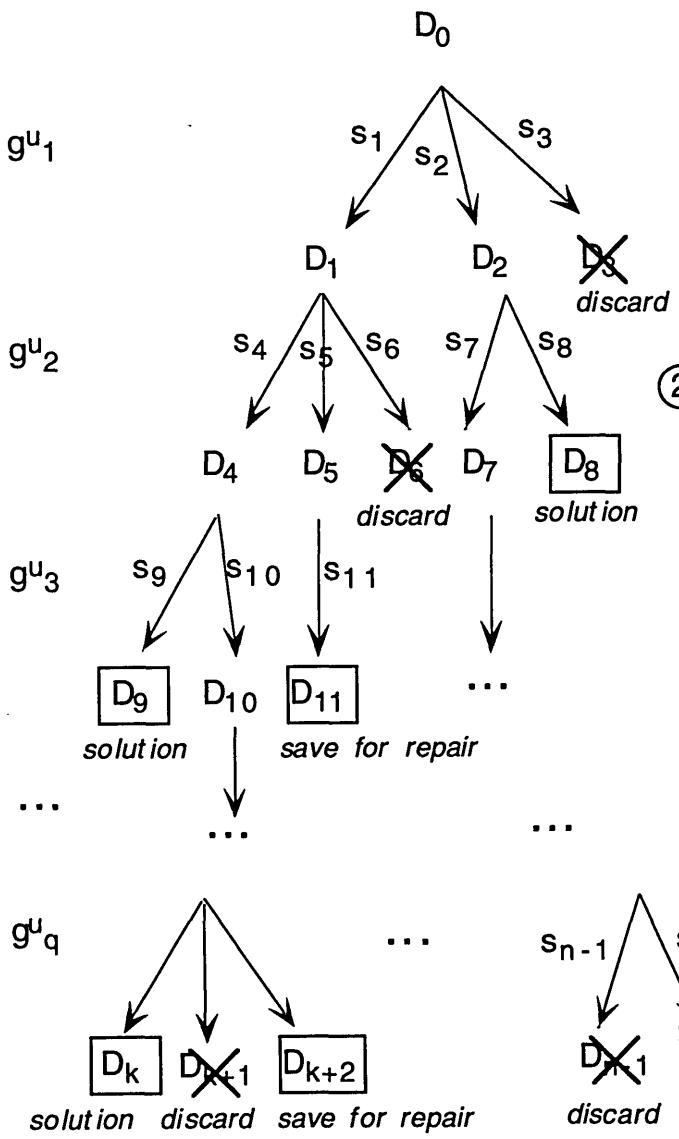
11. or as a decreaser on `opacity-of-elements-btw`.

design creation: it requires a lookup of design modification methods in the knowledge base and a simple matching of expression arguments to the current context; design creation requires copying and modifying designs. The small cost of proposing suggestions is offset by the benefit that the lookahead mechanism offers in decreasing the number of designs generated.

### 7.3.2 Sequential

In the sequential control structure, TAC proposes suggestions and creates new designs for one goal at a time, as the sequential-with-lookahead control structure does. TAC doesn't check suggestions against other goals, however, so it doesn't spot conflicts or synergy opportunities. Instead, it creates new designs for all suggestions. As a result, TAC running with this control structure may search a larger portion of the design space, and may exhibit more looping behavior, as one goal's modification undoes the effect of another goal's modification. The control structure for sequential design generation was shown in Figure 7.1 and is shown again here in Figure 7.18. The suggestion proposal mechanism for the sequential control structure (without lookahead) is summarized in Figure 7.19.

Given design $D_0$, goals $G = G^u \cup G^s = \{g^u_1\ g^u_2\ g^u_3 \ldots\} \cup \{g^s_1\ g^s_2\ g^s_3 \ldots\}$:




① Starting with $D_0$, propose suggestions $s_i$ for $g_{u1}$ and create new designs. Check designs for uniqueness, satisfied $g_{u1}$, satisfied $G^u$, satisfied G. Save as solutions those that satisfy G; save for repair those that satisfy $G^u$; discard those that are not unique or do not satisfy $g^u_1$.

② Starting with new designs $D_1$, $D_2$ that have not been saved or discarded, propose suggestions for $g^u_2$ and create new designs. Save or discard designs as in step above (checking $g^u_2$ satisfaction instead of $g^u_1$).

③ For each of remaining $g^u_i$, continue proposing suggestions, creating and checking new designs. With last goal $g^u_q$, save designs for repair that satisfy $g^u_q$ (rather than $G^u$).

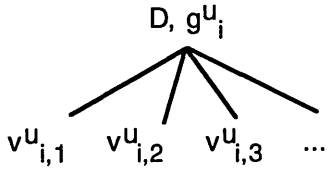
④ After new designs have been created for last goal $g^u_q$ and checked as above, repair designs that were saved for repair.

⑤ After all designs have been repaired, return solutions.

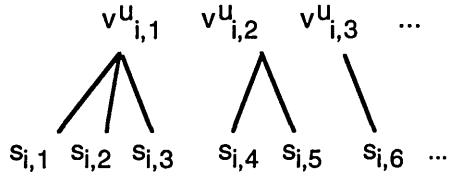
**Figure 7.18:** Sequential control structure.

Given design D, goals $G = G^u \cup G^s = \{g^u_1\ g^u_2\ g^u_3 \ldots\} \cup \{g^s_1\ g^s_2\ g^s_3 \ldots\}$, and goal $g^u_i$:

1. Propose value $v^u_{i,n}$ suggestions for $g^u_i$:




2. Propose design element suggestions $s_{i,o}$ for value suggestions $v^u_{i,n}$:




3. Result is set of design element suggestions for goal $g^u_i$: $\{s_{i,1}\ s_{i,2}\ s_{i,3}\ s_{i,4} \ldots\}$

**Figure 7.19:** Suggestion proposal for the sequential control structure.

As shown in the above figure, suggestion proposal is very straightforward: TAC proposes value suggestions for a goal, then design suggestions (not shown in the figure), then design element suggestions. It does no pruning on the set of suggestions, but creates one new design for each suggestion, checking the designs and saving or discarding them as described for the sequential-with-lookahead control structure. (See Figure 7.18.)

The examples below, discussed in the previous section, illustrate the sequential control structure and how it differs from sequential-with-lookahead.

Example 1:

Let's again say that we want the Living territory in the Tomek house to be visually open from the Dining territory, and we want one fireplace in the Living territory. We specify two goals:

```
<goal: (visually-open Living from Dining) true>
<goal: (fireplace-count in Living) 1>
```

The first goal is not satisfied; the second one is. TAC proposes suggestions for the unsatisfied `visually-open` goal. The suggestions are the same as with the sequential-with-lookahead control structure.[12]

12. In this and subsequent examples in this section, only design element suggestions will be shown.

Remove or puncture the existing fireplace, Fireplace1:

```
(or (remove Fireplace1)
    (puncture Fireplace1))
```

TAC creates new designs, one for each suggestion. Tomek#1 has the fireplace removed; Tomek#2 has the fireplace punctured at a default location. TAC finds the intended `visually-open` goal satisfied for each new design, so it proceeds with checking the remaining `fireplace-count` goal. Tomek#1 no longer has one fireplace, so it is saved for repair. Tomek#2 does have one fireplace, so both goals are satisfied, and Tomek#2 is a solution.

TAC proceeds to repair Tomek#1, proposing suggestions for increasing the fireplace count to one from zero:

```
(add fireplace to Living)
```

TAC carries out this suggestion, creating a new design Tomek#1#1 with a fireplace added to a default location in the Living territory. TAC reaches a solution quickly if the add routine happens to locate the new fireplace in such a way that the `visually-open` goal remains satisfied, or if subsequently puncturing the fireplace in its new location causes that goal to be satisfied. Alternately, if the add routine locates the new fireplace in the original fireplace's location, TAC will notice that it has seen the design before and will stop. Finally, if the add routine places the fireplace in a new location which causes the `visually-open` goal to become unsatisfied, TAC may loop removing the fireplace, adding it back, removing it again, etc. The looping results when a design modification, adding a fireplace in this case, can produce slightly different designs. The slight difference can result, for example, from an intervening modification that splits an edge into two edges. Recall that our definition of "same-as" is very strict and requires designs to have exactly the same edges and design elements. Since TAC doesn't measure degrees of sameness, slight differences between designs may cause TAC to continue its repair cycle. The issue of looping is discussed further in Section 7.4.

The sequential control structure produces the same solution for the above example as the sequential-with-lookahead control structure. Depending on how the add routine is implemented, it may also produce several other solutions with fireplaces in different locations. (Note that if the knowledge base had included moving the fireplace as a means of increasing visual openness, then the sequential-with-lookahead control structure also would have produced designs with fireplaces in the new locations.)

Example 2:

Again let's say that we want the Living territory to be visually open from the Dining territory in the Tomek house, and we want the center of the Living territory to be visible from the center of the Dining territory. We specify the two goals:

```
<goal: (visually-open Living from Dining) true>
<goal: (visible-center Living from Dining) true>
```

As with the sequential-with-lookahead control structure, TAC determines that both goals are unsatisfied and proposes satisfying the first goal by removing or puncturing the fireplace in order to increase the visual openness value:

```
(or (remove Fireplace1>)
    (puncture Fireplace1))
```

TAC creates a new design for each of these suggestions. Tomek#1 has the fireplace removed; Tomek#2 has the fireplace punctured at a default location. Both designs satisfy the `visually-open` goal, so TAC proceeds with each of these designs, checking whether the next unsatisfied goal, namely the `visible-center goal,` is satisfied. Removing the fireplace has satisfied this goal, so Tomek#1 is a solution. Puncturing the fireplace may or may not have caused this goal to be satisfied, depending on the default puncture location. If the puncture location happens to coincide with the line of sight between territory centers, the Living territory center will be visible from the Dining territory center, and Tomek#2 will be a solution. TAC will stop at this point returning Tomek#1 and Tomek#2 as solutions. If the puncture location does not coincide with the line of sight, then TAC will attempt to repair Tomek#2. It will propose removing the fireplace or puncturing the fireplace along the line of sight. If the fireplace is removed, the resulting design is the same as Tomek#1, so TAC discards the newly created design. If the fireplace is punctured (e.g. by widening the first puncture, or puncturing the fireplace in a second place), then TAC returns the new design as a solution.

Thus, in this example, the sequential control structure produces the same two designs as sequential-with-lookahead, possibly also producing a third design with a wider puncture or two punctures in the fireplace.

Example 3:

Returning to the Horner house, let's again specify two goals: a fireplace on an interior edge of the Living territory, and the Living territory visually open from the Dining territory.

```
<goal: (fireplace-on-interior-edge of Living) true>
<goal: (visually-open Living from Dining) true>
```

As before, TAC proposes suggestions for the unsatisfied `fireplace-on-interior-edge` goal. It checks its knowledge base for ways to satisfy the goal, finds that it can move the existing fireplace to an interior edge or add a new fireplace. It then checks the design, finding five candidate interior edges and proposes:

```
(or (move Fireplace1 to <edge: 18.75...>)
    (move Fireplace1 to <edge: 15.75...>)
    (move Fireplace1 to <edge: 29.06...>)
    (move Fireplace1 to <edge: 31.69...>)
    (move Fireplace1 to <edge: 15.75...>)
    (add fireplace to Living such that on-interior-edge))
```

TAC creates new designs for these suggestions, producing the same six designs as the sequential-with-lookahead control structure. As before, the second and third designs are solutions, the fourth and fifth are discarded because the intended goal is not satisfied, and the first and last designs are saved for repair because the `visually-open` goal has become unsatisfied. These designs to be repaired are shown in Figure 7.20.

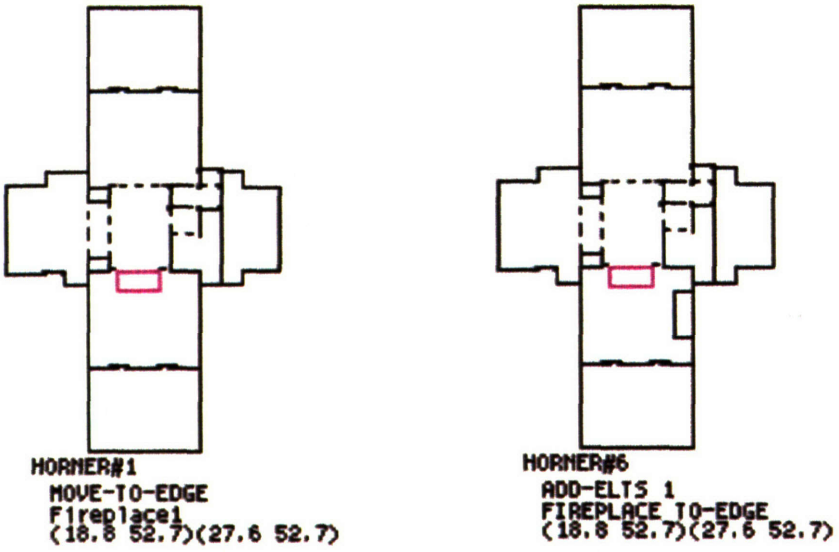



**Figure 7.20:** Designs to be repaired so that `visually-open` goal is satisfied.

To repair Horner#1, TAC proposes to increase the visual openness by:

```
(or (remove Fireplace1)
    (remove Bookcase1)
    (remove Stair)
    (puncture Fireplace1)
    (puncture Stair)
    (and (remove Fireplace1) (remove Stair))
    (and (puncture Fireplace1) (puncture Stair)))
```

TAC creates five new designs, one for each of the first five suggestions. (As before, the last two compound suggestions do not produce unique designs because the first modification satisfies the intended `visually-open` goal and produces the same designs as the first and fourth suggestions.) All but the first design, in which the fireplace has been removed, are solutions. The first design, Horner#1#1, is saved for repair. The sequential-with-lookahead control structure did not generate this design because it was able to detect that removing the fireplace would conflict with keeping a fireplace on an interior edge.

To repair Horner#1#1, TAC proposes adding a fireplace to an interior edge. The resulting design is the same as one of the designs created by moving the original fireplace to an interior edge, so TAC ends the repair cycle for Horner#1#1.

At this point, TAC has finished its repair of Horner#1 and has found four solutions.

TAC now focuses its repair efforts on Horner#6. Repair of Horner#6 proceeds as in the sequential-with-lookahead control structure. TAC proposes the same suggestions:

```
(or (remove Fireplace2)
    (remove Bookcase1)
    (remove Stair)
    (puncture Fireplace2)
    (puncture Stair)
    (and (remove Fireplace2) (remove Stair))
    (and (puncture Fireplace2) (puncture Stair)))
```

As in the sequential-with-lookahead control structure, the first suggestion yields the original design, the second through fourth suggestions yield solutions, and the last two suggestions do not yield unique designs.

In this example, TAC produced the same solutions as the sequential-with-lookahead control structure, but created two extra designs that were ultimately discarded.

Example 4:

Returning to a design discussed in the previous example, Horner#1, we specify goals for one fireplace, a fireplace on an exterior edge, and the Living territory visually open from the Dining territory:

```
<goal: (fireplace-count in Living) 1>
<goal: (fireplace-on-exterior-edge of Living) true>
<goal: (visually-open Living from Dining>) true>
```

As before, TAC finds that the last two goals are unsatisfied and proposes suggestions for the first unsatisfied goal, having a fireplace on an exterior edge: move the fireplace to an exterior edge or add a fireplace to an exterior edge.

```
(or (move Fireplace1 to <edge: (15.75...>)
    (move Fireplace1 to <edge: (31.69...>)
    (add fireplace to Living such that on-exterior-edge))
```

Carrying out these suggestions, TAC creates three new designs (Horner#1, Horner#2, Horner#3), one more than the sequential-with-lookahead control structure, which pruned the add suggestion. The first two designs are solutions. The third design, shown below, is saved for repair because it now has more than one fireplace. [13]

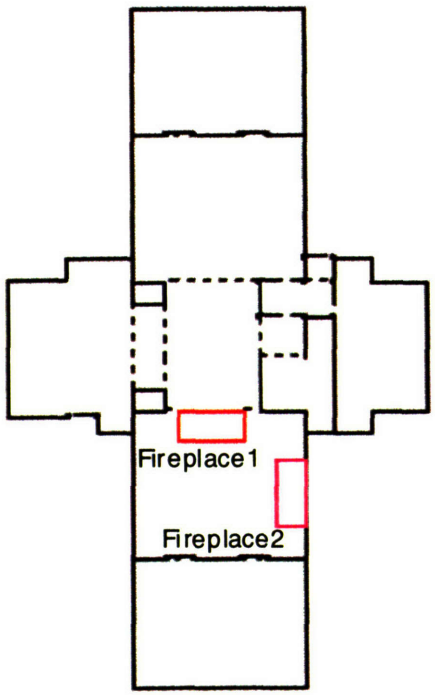



**Figure 7.21:** Design to be repaired to have one fireplace.

To repair the above design, TAC proposes removing either of the two fireplaces. Neither of the resulting designs are unique: removing Fireplace1 produces a design equivalent to Horner#2; removing Fireplace2 produces a design equivalent to the original design.

Thus, in this example, TAC produced the same solutions as with sequential-with-lookahead, but created three extra designs that did not lead to solutions.

13. Recall that the current version of the add routine produces one design, rather than all possible designs.

Benefits and Costs

In the examples above, the sequential control structure generated more designs than the sequential-with-lookahead control structure. As with sequential-with-lookahead, designs are generated breadth-first, with a branching factor of the number of suggestions and a depth of the number of unsatisfied goals. Without lookahead, however, the branching factor is not decreased. In addition, the number of unsatisfied goals is not deliberately decreased, though it may be decreased due to TAC's "getting lucky" with discovered synergies. Detecting these synergies is dependent on goal order (discussed in Section 7.4). As shown in Example 1, whether synergies exist or not can depend on the design modification routines; this is true for all the control structures tested. (Recall that if the default location for adding a fireplace happened to be an exterior edge, the visual openness goal was also satisfied when a fireplace was added to satisfy the fireplace count goal.)

As discussed in more detail in Section 7.4, the sequential control structure can exhibit looping behavior, as one goal's modification undoes the effect of another goal's modification. The sequential-with-lookahead control structure can exhibit this behavior also, but does so less often because its lookahead routine is able to detect and prune conflicting suggestions.

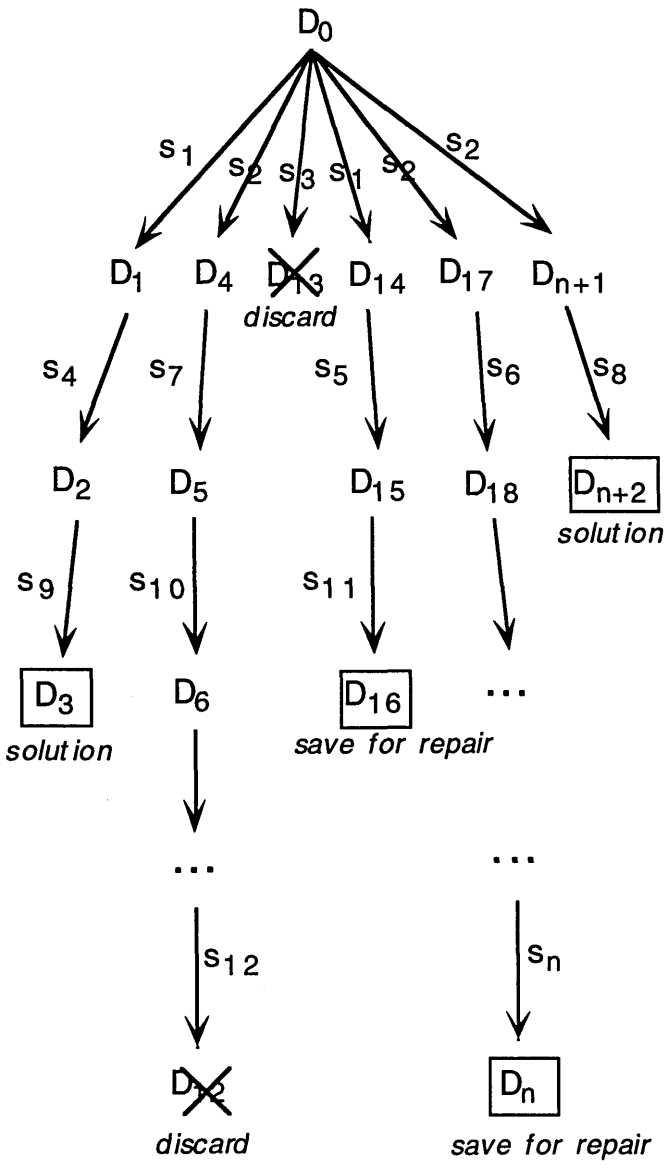
Not performing lookahead has a benefit, however: Because the sequential control structure generates more new designs, it can sometimes find more solutions than the sequential-with-lookahead control structure. As we saw in the Example 2, most of the additional solutions did not differ significantly from other solutions: one had a wider puncture through the fireplace, one an extra puncture through the fireplace. Once presented with a solution that contained a punctured fireplace, the designer could easily vary the characteristics of the puncture.

The solutions that differed from those found with sequential-with-lookahead had a fireplace in a new location. They resulted from TAC's removing the fireplace and adding a new one at a different location, as discussed in Example 1. Because TAC does no lookahead with the sequential control structure, it is allowed to violate a goal, that of having one fireplace, in order to satisfy another goal, making the Living visually open from the Dining. In this example, the combination of remove and add operators produces the result of a third operation, namely a move, which satisfies both goals. As illustrated here, operators may combine in unexpected ways to produce solutions. The sequential-with-lookahead control structure has fewer opportunities for unexpected operator combinations because it prunes more suggestions.

### 7.3.3 Concurrent

The concurrent control structure proposes suggestions for all goals at the same time in the context of the original design. The suggestions are compound suggestions, each of which represents a plan for producing a design intended to satisfy all unsatisfied goals. Each such design is created by carrying out a compound suggestion's proposed modifications sequentially on a copy of the original design. If a new design satisfies some of the goals, but not all of them, TAC attempts to repair the design. Figures 7.22 and 7.23 summarize the concurrent control structure.

Given design $D_0$, goals $G = G^u \cup G^s = \{g^u_1\ g^u_2\ g^u_3 \ldots\} \cup \{g^s_1\ g^s_2\ g^s_3 \ldots\}$:




(1) Starting with $D_0$, propose compound suggestions for G. Create new designs by sequentially carrying out simple suggestions $s_i$ in each compound suggestion $C_j$. After each $s_i$, check design for uniqueness, satisfied intended goal $g^u_i$, satisfied $G^u$, satisfied G. If unique and satisfies G, save as solution; if unique and satisfies $G^u$, save for repair.

After first $s_i$ in each $C_j$, if design does not satisfy $g^u_i$, discard. After all other $s_i$, if design does not satisfy $g^u_i$, skip $s_i$. After last $s_i$, if design is unique and not a solution, save for repair; if not unique, discard.

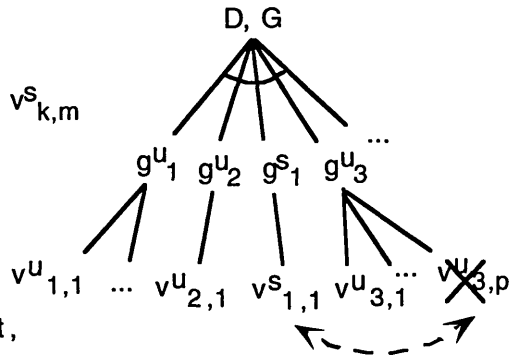
(2) After new designs have been created for last compound suggestion and checked as above, repair designs that were saved for repair.

(3) After all designs have been repaired, return solutions.

**Figure 7.22:** Concurrent control structure

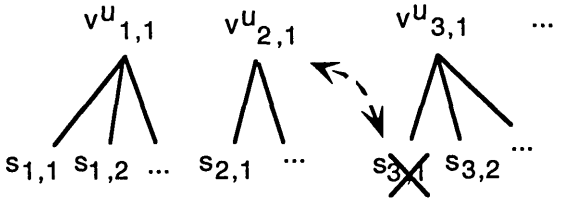
Given design $D_0$ and goals $G = G^u \cup G^s = \{g^u_1\ g^u_2\ g^u_3 \ldots\} \cup \{g^s_1\ g^s_2\ g^s_3 \ldots\}$:

1. Propose value suggestions for all goals:
   - for $g^s_k$ in $G^s$ propose keep-value suggestions $v^s_{k,m}$
   - for $g^u_i$ in $G^u$ propose value suggestions $v^u_{i,n}$

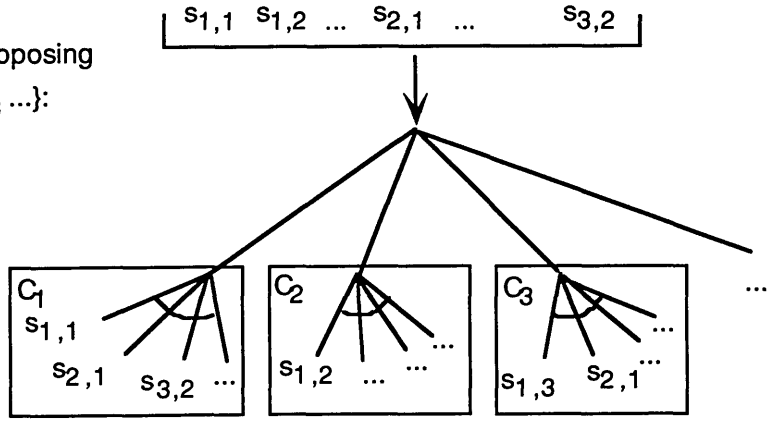



2. Check value suggestions for synergy and conflict, pruning as necessary:

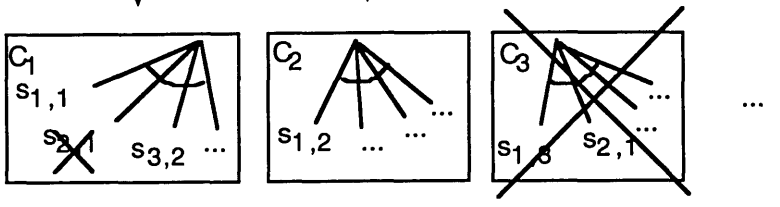
3. Propose design element suggestions $s_{i,o}$ for each value suggestion $v^u_{i,n}$; check each $s_{i,o}$ for conflict with any value suggestion $v^x_{j,n}$:




4. Put suggestions $s_{i,o}$ in DNF, proposing compound suggestions $\{C_1\ C_2\ C_3 \ldots\}$:




5. Check each compound suggestion $C_j$ for synergy and conflict within $C_j$:
   - $s_{1,1}$+ $s_{2,1}$: synergy, $s_{2,1}$ pruned
   - $s_{1,3}$ +$s_{2,1}$: conflict, $C_3$ pruned




6. Result is set of compound design element suggestions for all goals in $G^u$: {C1, C2, ... }

**Figure 7.23:** Concurrent-lookahead

As with the other two control structures, TAC starts with a design and a set of goals and proposes repair suggestions for unsatisfied goals. The concurrent control structure, however, proposes suggestions for all goals, conjoining the suggestions to produce compound suggestions. Using a mechanism similar to the lookahead mechanism described earlier, TAC checks for interaction between the simple suggestions within each compound suggestion, pruning as necessary when it detects conflict or synergy. The resulting compound suggestions are intended to satisfy all unsatisfied goals.

As shown in Figure 7.23, TAC begins its suggestion proposal by creating value suggestions for each goal. (Recall that keep-value suggestions are proposed for satisfied goals; increase-, decrease-, or set-value suggestions for unsatisfied goals.) It conjoins the value suggestions and checks them for conflict and synergy, pruning conflicting suggestions and combining synergistic ones. TAC proposes design suggestions, then design element suggestions, and checks for interaction. It compares each suggestion with the keep-value suggestions proposed in the previous step, pruning when it detects conflict. It then looks for conflict or synergy among the remaining suggestions: It puts the suggestions in disjunctive normal form, which results in a set of compound suggestions, each of which contains one simple suggestion for each unsatisfied goal. It checks for synergy or conflict among the simple suggestions, combining simple suggestions that exhibit synergy and pruning compound suggestions that contain conflicts.

For each surviving design element suggestion TAC creates a new design. As with the other two control structures, TAC checks the design to see if the design is a solution or if it's been seen before. If neither of these situations is found, TAC checks intended goals, either discarding the design or saving it for repair.

As shown in Figure 7.22, creating a new design and checking intended goals for a compound suggestion are a bit different than for a single simple suggestion. A compound suggestion can be thought of as a plan for creating a solution; each simple suggestion in the compound suggestion is a step in the plan. With a compound suggestion, TAC starts with a copy of the original design, then sequentially carries out each of its simple suggestions. Each simple suggestion results in a new design which is checked as in the previous control structures, to see if it should be saved as a solution, saved for repair, or discarded. If all goals are satisfied, any subsequent proposed modifications are unnecessary; synergy has been discovered. If the design has not been seen before, it is saved as a solution. In Figure 7.22, $D_3$ is a solution created from compound suggestion (and $s_1$ $s_4$ $s_9$); $D_{n+2}$ is a solution created from compound suggestion (and $s_2$ $s_8$). [14] If all unsatisfied goals are satisfied, and the design has not been seen before, it is saved for repair (because some of the formerly satisfied goals are now unsatisfied). $D_{16}$ is such an example; it was created from

14. The original compound suggestions may have contained more simple suggestions following $s_8$ and $s_9$, but they were unnecessary so were not carried out and are not shown in the figure.

compound suggestion (and $s_1$ $s_5$ $s_{11}$). The design is discarded if the first simple suggestion's intended goals are not satisfied, as with $D_{13}$. The design is not discarded if later simple suggestions don't satisfy intended goals: These suggestions, though relevant to the original design (the context in which the suggestion was proposed), may not be relevant to the current design. If they do not satisfy their intended goals, TAC assumes they are not relevant and skips them, then continues with subsequent simple suggestions. After the last simple suggestion, if the design is not a solution and has not been seen before, it is saved for repair; e.g. $D_n$ in Figure 7.22.[15]

A final note about creating designs from compound suggestions: Recall that carrying out a simple suggestion results in a new design, and the suggestion that follows in the compound suggestion operates on that design rather than the original design. Because each simple suggestion is originally proposed in the context of the original design, it must be mapped to the appropriate new design context before being carried out. This mapping may or may not be possible, depending on how different the new design is from the original. This issue is discussed in Example 2 below.

The following examples, shown previously, illustrate the concurrent control structure. In each example, we point out how the concurrent control structure's solutions differ from those found by the other two control structures.

Example 1: Predictable conflict

Let's again say that we want the Living territory in the Tomek house to be visually open from the Dining territory, and we want one fireplace in the Living territory. We specify the two goals:

```
<goal: (visually-open Living from Dining) true>
<goal: (fireplace-count in Living) 1>
```

TAC proposes value suggestions for each goal, conjoining them into a compound suggestion that specifies increasing visual openness and keeping the number of fireplaces at one:

```
(and (increase-value of (visual-openness of Living from Dining)
                  until visual-openness greater than 0.6)
     (keep-value of (fireplace-count in Living) 1))
```

It checks for interaction between the suggestions, but finds none, so it proceeds with proposing suggestions for the unsatisfied `visually-open` goal:

```
(or (remove blocking-elements-btw Living and Dining)
    (puncture blocking-elements-btw Living and Dining))
```

15. One further note: If a goal expression is a conjunction, then suggestion proposal proceeds as described here. A goal with the expression `(and (visually-open Living from Dining) (visible-center Living from Dining))` results in the same suggestions as with two separate goals. (See Example 2 that follows.) Fewer solutions may be found, however: the clauses are not reasoned about separately, as goals are, so no intermediate designs are generated that may satisfy one of the clauses but not the other.

It determines that the fireplace blocks the view, so it proposes removing or puncturing it:

```
(or (remove Fireplace1)
    (puncture Fireplace1))
```

As with the sequential-with-lookahead control structure, TAC checks each of these suggestions for interaction with the keep-value suggestion for the `fireplace-count` goal. It finds that removing the fireplace conflicts with keeping the number of fireplaces at one, so it prunes that suggestion and only creates a new design with the fireplace punctured. It finds that the new design satisfies the `visually-open` goal and returns it as a solution. TAC has no designs to repair, so it stops.

In this example, TAC found the same solution as with the sequential-with-lookahead control structure, and its behavior was quite similar.

Example 2: Predictable synergy

Again let's say that we want the Living territory to be visually open from the Dining territory in the Tomek house, and we want the center of the Living territory to be visible from the center of the Dining territory. We specify the two goals:

```
<goal: (visually-open Living from Dining) true>
<goal: (visible-center Living from Dining) true>
```

TAC proposes value suggestions for the goals, conjoining them into a compound suggestion that specifies increasing visual openness and making the center of Living visible from the center of Dining:

```
(and (increase-value of (visual-openness of Living from Dining)
                until visual-openness greater than 0.6)
     (set-value of (visible-from (center of Living) (center of Dining))
                to true))
```

It finds no interaction between the suggestions, so it proposes more specific suggestions, conjoining suggestions for each of the goals:

```
(and
   (or (remove Fireplace1)
       (puncture Fireplace1))
   (or (remove Fireplace1)
       (puncture Fireplace1 at <edge: 111.06...>)))
```

TAC checks each of these suggestions for interactions with the value suggestions above and finds none. It then puts the suggestions into disjunctive normal form to check for conflict and synergy among the suggestions themselves.

In disjunctive normal form, the suggestions are:

```
(or
  (and (remove Fireplace1)
       (remove Fireplace1))
  (and (remove Fireplace1)
       (puncture Fireplace1 at <edge: 111.06...>))
  (and (puncture Fireplace1)
       (remove Fireplace1))
  (and (puncture Fireplace1)
       (puncture Fireplace1 at <edge: 111.06...>)))
```

Checking these compound suggestions, TAC notices that the first compound suggestion contains the same simple suggestions, so it prunes one of them. It notices that the second compound suggestion cannot be completely carried out: a fireplace that has been removed cannot then be punctured, so the second clause in the compound suggestion is pruned. (Rather than say that the puncture suggestion conflicts, we say that it is *inapplicable*.) The third suggestion doesn't contain a conflict (because it's not clear that removing the fireplace will undo the puncture modification's intended goal), but it does contain a wasted operation: If the fireplace is to be removed, then puncturing it first isn't necessary; a remove operation subsumes all others. TAC avoids such wasted operations in a compound suggestion by pruning all simple suggestions involving the design element that appears in a remove suggestion. In this example, TAC prunes the puncture suggestion. Finally, TAC notices a synergy: the last suggestion contains two very similar simple suggestions, with one more specific than the other. It keeps the more specific suggestion, pruning the less specific one which is subsumed. The pruning is summarized in Figure 7.24.

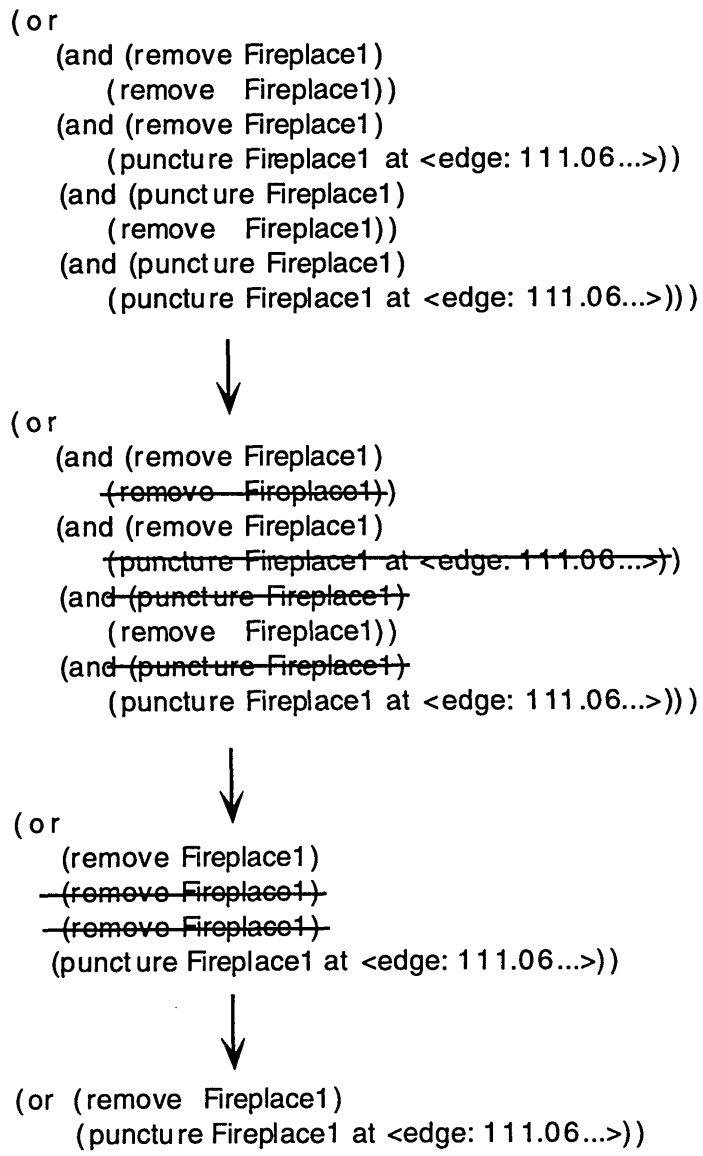



**Figure 7.24:** Pruning of suggestions for `visually-open` and `visible-center` goals.

TAC is now left with the same suggestions it proposed using the sequential-with-lookahead control structure:

```
(or (remove Fireplace1>)
    (puncture Fireplace1 at <edge 111.06...>))
```

The suggestions each have two intended goals, making the Living territory visually open from the Dining territory and making its center visible from the Dining territory's center. TAC finds both goals satisfied and returns Tomek#1 and Tomek#2 as solutions, as it did with the sequential-with-lookahead control structure. The concurrent control structure, however, proposed sugges-

tions that had to be pruned because they contained inapplicable simple suggestions, i.e. suggestions that could not be carried out because a previous suggestion modified the design in such a way as to make them inappropriate or impossible. In the above example, the suggestion to puncture the fireplace after removing it was impossible to carry out. The knowledge necessary for pruning this suggestion must be represented explicitly, or TAC must know to skip modifications it cannot perform. Both the sequential and sequential-with-lookahead control structures avoid inapplicable suggestions because they propose suggestions in the context of a new design rather than in the context of the original design. With a new design, only suggestions appropriate for that particular design are proposed. The sequential and sequential-with-lookahead control structure, however, do perform wasted operations that the concurrent control structure can avoid. They, for example, will puncture the fireplace to satisfy the `visually-open` goal, then subsequently propose removing it to satisfy the `visible-center` goal. Since the concurrent control structure must represent extra knowledge in order to avoid inapplicable suggestions, it can also use that knowledge to avoid wasted operations. This benefit, however, is outweighed by the cost of having to explicitly represent the knowledge it requires.

Example 3: Unpredictable conflict

Returning to the Horner house, consider the two goals: a fireplace on an interior edge of the Living territory, and the Living territory visually open from the Dining territory.

```
<goal: (fireplace-on-interior-edge of Living) true>
<goal: (visually-open Living from Dining) true>
```

Since this example has only one unsatisfied goal and no interactions between suggestions, the concurrent control structure generates the same designs and solutions as the sequential-with-lookahead control structure.

Example 4: Unpredictable synergy

For the Horner#1 design, shown in Figure 7.25, we specify goals for one fireplace in the Living territory, a fireplace on an exterior edge of the Living territory, and the Living territory visually open from the Dining territory:

```
<goal: (fireplace-count in Living) 1>
<goal: (fireplace-on-exterior-edge of Living) true>
<goal: (visually-open Living from Dining>) true>
```

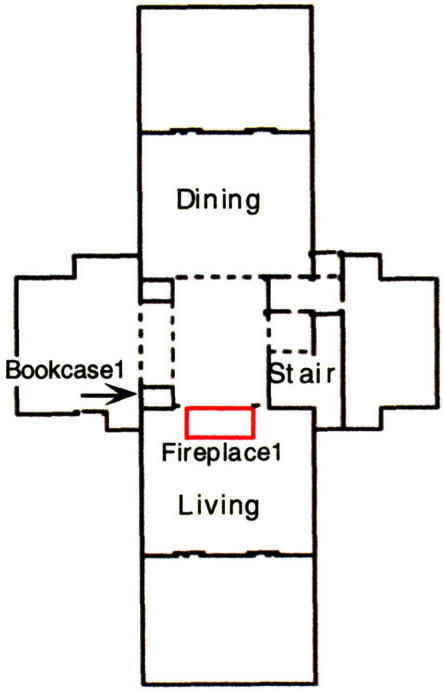



**Figure 7.25:** Territory model for Horner#1.

As before, TAC finds that the last two goals are unsatisfied. It proposes suggestions, conjoining them into a compound suggestion to keep the value of the fireplace count at one, make the design have a fireplace on an exterior edge, and increase the visual openness value:

```
(and
    (keep-value of (fireplace-count in Living) 1)
    (or (set-value of (on-exterior-edge Fireplace1)
                      to true)
        (increase-value of (fireplace-count in Living)
                           to 2 such that on-exterior-edge is true))
    (increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6))
```

It checks for interaction, noticing that increasing the number of fireplaces to satisfy the `fireplace-on-exterior-edge` goal conflicts with the keeping the number of fireplaces at one. So it prunes that suggestion:

```
(and
    (keep-value of (fireplace-count in Living) 1)
    (or (set-value of (on-exterior-edge Fireplace1)
                      to true)
        ~~(increase-value of (fireplace-count in Living)~~
                           ~~to 2 such that on-exterior-edge is true)~~)
    (increase-value of (visual-openness of Living from Dining)
                   until visual-openness greater than 0.6))
```

After pruning:

```
(and  (keep-value of (fireplace-count in Living) 1)
      (set-value of (on-exterior-edge Fireplace1)
                    to true)
      (increase-value of (visual-openness of Living from Dining)
                         until visual-openness greater than 0.6))
```

TAC then proposes more specific suggestions for the last two of the suggestions above:

```
(and
   (move Fireplace1 to any exterior-edges-for Fireplace1)
   (or (remove blocking-elements-btw Living and Dining)
       (puncture blocking-elements-btw Living and Dining)))
```

It checks the design, substitutes exterior edges and blocking design elements into the above suggestions, and proposes:

```
(and
   (or (move Fireplace1 to <edge 15.75...>)
       (move Fireplace1 to <edge 31.69...>))
   (or (remove Fireplace1)
       (remove Bookcase1)
       (remove Stair)
       (puncture Fireplace1)
       (puncture Stair)
       (and (remove Fireplace1) (remove Stair))
       (and (puncture Fireplace1) (puncture Stair))))
```

TAC checks each of these suggestions for interaction with the value suggestions, and finds that removing the fireplace conflicts with keeping the number of fireplaces at one. So it prunes that suggestion:

```
(and
   (or (move Fireplace1 to <edge 15.75...>)
       (move Fireplace1 to <edge 31.69...>))
   (or ~~(remove Fireplace1)~~
       (remove Bookcase1)
       (remove Stair)
       (puncture Fireplace1)
       (puncture Stair)
       ~~(and (remove Fireplace1) (remove Stair))~~
       (and (puncture Fireplace1) (puncture Stair))))
```

It puts the suggestions into disjunctive normal form:

```
(or (and (move Fireplace1 to <edge 15.75...>)
         (remove Stair))
    (and (move Fireplace1 to <edge 31.69...>)
         (remove Stair))
    (and (move Fireplace1 to <edge 15.75...>)
         (remove Bookcase1))
    (and (move Fireplace1 to <edge 31.69...>)
         (remove Bookcase1))
    (and (move Fireplace1 to <edge 15.75...>)
         (puncture Fireplace1))
    (and (move Fireplace1 to <edge 31.69...>)
         (puncture Fireplace1))
    (and (move Fireplace1 to <edge 15.75...>)
         (puncture Stair))
    (and (move Fireplace1 to <edge 31.69...>)
         (puncture Stair))
    (and (move Fireplace1 to <edge 15.75...>)
         (puncture Fireplace1)
         (puncture Stair))
    (and (move Fireplace1 to <edge 31.69...>)
         (puncture Fireplace1)
         (puncture Stair)))
```

TAC then checks for interactions within each compound suggestion and finding none, creates ten designs, one for each compound suggestion. Only two of the designs are unique, however. In each compound suggestion, the first simple suggestion (to move the fireplace) satisfies both goals, so the second simple suggestion is not carried out. The first and second compound suggestions generate two new designs which are solutions. The designs created by the remaining compound suggestions are the same as one of these two designs and are discarded.

As noted above, TAC found no interactions within the compound suggestions. A stricter definition of conflict might have disallowed compound suggestions that proposed moving the fireplace, then puncturing it, or puncturing a fireplace, then moving it. Recall that TAC prunes compound suggestions which contain conflicting simple suggestions, i.e. suggestions that clobber one another's intended goals. Chances are that in a compound suggestion proposing both puncturing and moving a design element, the move will keep the puncture from accomplishing its intended goal, e.g. of making a territory center visible. Rather than assume these modifications will conflict, TAC takes a conservative approach: It only prunes compound suggestions containing predictable conflicts (e.g. adding a fireplace while attempting to keep the number of fireplaces constant), opting to repair the resulting new design if necessary. After repair of a design whose fireplace has been punctured then moved, however, the fireplace will contain a puncture that may or may not be relevant to any eventual solutions.

Note that if the puncture suggestion contains a specific location, rather than simply specifying a default, a potential conflict with a move suggestion would be more easily predicted (because the puncture location would no longer intersect the fireplace in its new location). TAC, however, would have to check the design to detect this conflict, so the suggestion still could not be pruned ahead of time.

In this example, the concurrent control structure found the same two solutions as the sequential-with-lookahead control structure, but it generated eight designs that were discarded. The extra designs were the result of TAC's proposing suggestions for all goals in the context of the original design: the modifications for the first goal (moving the fireplace) changed the design in such a way that suggestions for other goals were no longer relevant. Note that different orderings of simple suggestions in the compound suggestions would not have produced this result. Simple suggestion order reflects goal order, an issue that is discussed in the next section.

Benefits and Costs

As shown in the examples above, the concurrent control structure proposes suggestions once for all goals in the context of the original design, rather than proposing suggestions multiple times as the sequential-with-lookahead and sequential control structures do. Prior to creating designs, it checks those suggestions for interactions, producing a set of compound suggestions, each of which represents a plan for producing a design intended to satisfy all unsatisfied goals. Checking the suggestions for interactions, i.e. conflict and synergy opportunities, reduces the number of designs generated with this control structure.

These two benefits, however, are offset by the costs of the concurrent control structure. Even though the concurrent control structure can limit search by detecting conflicts and synergies, it often generates more intermediate designs than the sequential-with-lookahead control structure. Viewing design generation as search, the compound suggestions represent paths through the search tree to designs at leaf nodes. The tree is searched depth-first, one compound suggestion at a time, with intermediate designs created after each simple suggestion in a compound suggestion. The extra intermediate designs are due to a large branching factor at the root of the design tree. The worst case branching factor is the maximum number of compound suggestions, $s^u$, where s is the number of suggestions per goal, and u is the number of unsatisfied goals. (Recall that the branching factor for the sequential-with-lookahead and sequential control structures is s.) The branching factor after the first tree level is one, since compound suggestions are carried out a single simple suggestion at a time. The maximum depth of the search tree is u, as with the other two control structures. In the worst case then, the tree size is $s^u \times u$. With only one unsatisfied goal, as in Example 3, the number of designs generated is the same as with the other two control structures, or $s^1 \times 1 = s$.

The size of the design tree is usually not the worst case size. The concurrent control structure decreases the branching factor by using a mechanism similar to lookahead, and decreases the tree depth by checking each new design to see if it should be discarded or saved as a solution. Nevertheless, when there is more than one unsatisfied goal, the concurrent control structure often generates more intermediate designs than the sequential-with-lookahead control structure, and many of those designs are discarded as duplicates. As shown in Example 4, eight designs out of ten were discarded as duplicates because the first proposed modification in each compound suggestion satisfied both goals, rendering subsequent modifications unnecessary. The concurrent control structure usually generates fewer designs than the sequential control structure without lookahead, however, since its lookahead-like suggestion proposal decreases the likelihood of looping with repair cycles. Design generation for each control structure is discussed further in Section 7.6.

Additional costs result from proposing compound suggestions in the context of the original design. Beginning with a copy of the original design, the first simple suggestion is carried out, producing a new intermediate design; the next simple suggestion is carried out on that design, producing another new intermediate design; and so on. The simple suggestions must be mapped to the new design contexts before they are carried out. A small cost is associated with performing this mapping. A larger cost is incurred when the mapping is not possible and suggestions are inapplicable, i.e. irrelevant or impossible to carry out in the new design context. Inapplicability arises because with each modification the new design differs more from the original design, and as a result, suggestions that were appropriate for the original design may no longer be applicable to the new design. When this situation arises, TAC skips the suggestions, which usually results in designs that must be either discarded as duplicates or saved for repair.

Some inapplicable suggestions can be avoided if operator interactions are known ahead of time. The remove operator, for example, renders impossible subsequent operators that attempt to act on the removed object. TAC was able to prune a puncture fireplace suggestion in Example 2, for instance, because that suggestion followed a remove fireplace suggestion. The knowledge needed to prune inapplicable suggestions is not necessary with the sequential and sequential-with-lookahead control structures because these control structures do not propose inapplicable suggestions. Having to represent extra knowledge does have one benefit, however: the knowledge also can be used to avoid wasted operations. If the remove operator, for example, is represented as subsuming all other operators, rendering them impossible if they follow or moot if they precede, TAC can avoid modifying a design element, then removing that design element. The concurrent control structure is able to do this, as shown in Example 2; the other two control structures are not. Having to represent extra knowledge incurs a cost, however: as more interactions are explicitly represented, the knowledge base becomes large and difficult to maintain.

Finally, a small cost is incurred by the concurrent control structure's search strategy. Since the

design space is searched depth-first, identical designs generated later in the search may be reached via fewer modifications from the original design than designs generated earlier. Since TAC prefers solutions that are minimal, it substitutes the later designs for earlier ones in repair and solution sets. Substitution of this sort is unnecessary with sequential design generation because designs are generated breath-first.

## 7.4 Goal Order

Goal order affects the efficiency of search through the design space and the number and quality of solutions found. We measure a design's quality by the number of modification steps from the original design, and by how many of those steps were actually necessary for satisfying the specified goals.

All three control structures are sensitive to goal order, which determines the order in which suggestions are proposed, and therefore, the order in which modification operators are carried out. Different operator orders can produce different designs because: (1) synergies may be evident in some goal orders, but not others, and (2) operators in this domain are not necessarily commutative. The ability to detect some synergies ahead of time, as in the sequential-with-lookahead and concurrent control structures, decreases goal order sensitivity. Example 1 illustrates goal order influence on the sequential-with-lookahead control structure and discusses optimal and nonoptimal goal orders; the other control structures behave similarly. Example 2 discusses goal orders for the set of goals used in the experiments described at the end of this chapter.

Example 1

Returning to the Horner#1 design, which is shown below, let's again say that we want the center of the Living territory to be visible from the center of the Dining territory, and we want the Living territory to be visually open from the Dining territory:

```
<goal: (visible-center Living from Dining) true>
<goal: (visually-open Living from Dining) true>
```

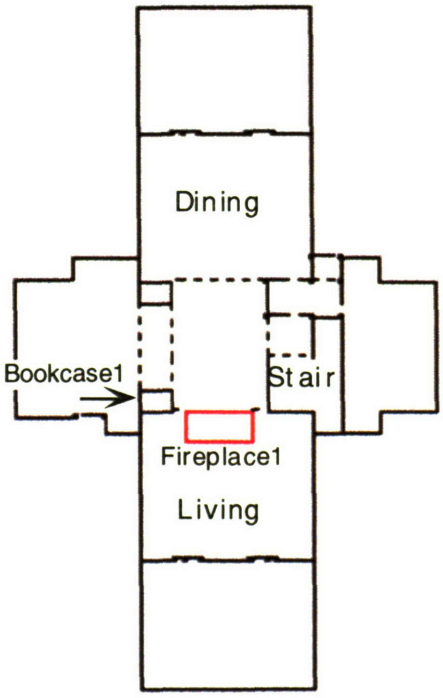



**Figure 7.26:** Territory model for Horner#1.

Given the two goals above, TAC proposes suggestions for the first goal, the `visible-center` goal:

```
(or (remove Fireplace1>)
    (puncture Fireplace1 at <edge: 111.06...>))
```

It performs lookahead, notes no interactions that change the suggestions, then creates one design for each suggestion. It makes sure that the `visible-center` goal is satisfied, then finds that the new designs also satisfy the `visually-open` goal. It proposes no further suggestions and returns the two designs as solutions.

With the goals in reverse order, TAC first proposes suggestions for the `visually-open` goal:

```
(or (remove Fireplace1)
    (remove Bookcase1)
    (remove Stair)
    (puncture Fireplace1)
    (puncture Stair)
    (and (remove Fireplace1) (remove Stair))
    (and (puncture Fireplace1) (puncture Stair)))
```

Performing lookahead, TAC proposes for the `visible-center` goal:

```
(or (remove Fireplace1)
    (puncture Fireplace1 at <edge: 111.06...>))
```

It notes a synergy between the puncture fireplace suggestions, and refines its suggestions for the `visually-open` goal:

```
(or (remove Fireplace1)
    (remove Bookcase1)
    (remove Stair)
    (puncture Fireplace1 at <edge: 111.06...>)
    (puncture Stair)
    (and (remove Fireplace1) (remove Stair))
    (and (puncture Fireplace1 at <edge: 111.06...>) (puncture Stair)))
```

It creates seven designs, one for each suggestion. As in previous examples, the last two compound suggestions produce duplicates. The new designs, which include two solutions, are:

Horner#1#1 with fireplace removed; solution

Horner#1#2 with bookcase removed

Horner#1#3 with stair removed

Horner#1#4 with fireplace punctured along line of sight between territory centers; solution

Horner#1#5 with stair punctured

Horner#1#6 with fireplace removed; duplicate of Horner#1#1, discarded

Horner#1#7 with fireplace punctured along sight line; duplicate of Horner#1#4, discarded

TAC then proposes suggestions for the `visible-center` goal for each of the three designs that are not solutions or duplicates (Horner#1#2, Horner#1#3, Horner#1#5):

```
(or (remove Fireplace1>)
    (puncture Fireplace1 at <edge: 111.06...>))
```

It carries out these suggestions and creates six more designs, two of which are solutions:

Horner#1#2#1 with fireplace removed; duplicate of Horner#1#1, discarded

Horner#1#2#2 with fireplace punctured along line of sight; solution

Horner#1#3#1 with fireplace removed; duplicate of Horner#1#1, discarded

Horner#1#3#2 with fireplace punctured along line of sight; solution

Horner#1#5#1 with fireplace removed; duplicate of Horner#1#1, discarded

Horner#1#5#2 with fireplace punctured along line of sight; solution

TAC returns five designs as solutions, having generated a total of 13 designs. Since TAC prefers minimal solutions, i.e. those that are a minimum number of modification steps from the original design, Horner#1#1 and Horner#1#4 are considered the best solutions. The other solutions contain extra modifications that we know from the existence of the two minimal solutions are not necessary for satisfying both goals.

Not coincidentally, the two minimal solutions are identical to the solutions found with the goals in the reverse order. This order—`visible-center` goal, followed by `visually-open` goal—is an optimal goal order because the goal with synergistic operators, i.e. that will satisfy more than one goal, precedes the goal with which its operators interact.

TAC's behavior would be less sensitive to goal order in this example if its lookahead mechanism pruned all the `visually-open` suggestions except for the two synergistic ones, removing the fireplace and puncturing it along the line of sight. The possibility of unpredictable synergies, however, means that TAC's behavior still can depend on goal order.

Example 2

For the Horner design, let's say that we'd like the center of Living territory visible from the center of the Dining territory, the Living territory visually open from the Dining territory, a fireplace on an interior edge of the Living territory, one fireplace in the Living territory, and one fireplace in the entire design:

```
(<goal: (visible-center Living Dining) true>
 <goal: (visually-open Living Dining) true>
 <goal: (fireplace-on-interior-edge of Living) true>
 <goal: (fireplace-count in Living) 1>
 <goal: (fireplace-count in Horner) 1)
```

The above goals are in an optimal order because goals with synergistic operators precede goals with which they interact. Puncturing the fireplace to satisfy the `visible-center` goal also may satisfy the `visually-open` goal. If the design has no fireplace, TAC will satisfy the first fireplace goal by adding a fireplace to an interior edge of the Living territory; all three fireplace goals will be satisfied then. With or without lookahead, one modification can satisfy the three fireplace goals.

The following goal order is nonoptimal because goals with synergistic operators follow goals with which they interact:

```
(<goal: (visually-open Living Dining) true>
 <goal: (visible-center Living Dining) true>
 <goal: (fireplace-count in Horner) 1>
 <goal: (fireplace-count in Living) 1>
 <goal: (fireplace-on-interior-edge in Living) true>)
```

With this goal order, TAC will satisfy the `visually-open` goal first, generating some designs that also satisfy the `visible-center` goal and some that do not. The designs that don't satisfy both goals, e.g. with the bookcase removed, are then modified to satisfy the `visible-center` goal. If the previous modifications for the `visually-open` goal were not necessary for satisfying both goals, the resulting designs contain unnecessary modifications and, therefore, are not minimal solutions. Puncturing the fireplace, for example, satisfied both goals so removing the bookcase was unnecessary. Also with this goal order, if the design has no fireplace, TAC will attempt to satisfy the first fireplace goal of having one fireplace in the entire design. If TAC performs lookahead, it figures out that all three fireplace goals can be satisfied by adding a fireplace to an interior edge of the Living territory. Without lookahead, TAC will add a fireplace to a random location in the design. It then will add a fireplace to the Living territory to satisfy the second fireplace goal, then move the fireplace to an interior edge of the Living territory to satisfy the third fireplace goal, then remove the first fireplace to resatisfy the first fireplace goal—four modifications instead of one.

As both these examples have illustrated, there is an optimal goal order for producing minimal solutions. This order, however, is difficult to determine apriori because, as noted previously, the effects of operators in this domain are very difficult to predict. The issue of goal ordering is discussed further in Chapter 10.

## 7.4 Termination

As discussed earlier, TAC stops iteration with a particular design when the design is a solution or when the intended goal for the design has not been satisfied. TAC also stops iteration when it encounters a design it has seen before or when it reaches an iteration limit.

### 7.4.1 Identifying Equivalent Designs

It's important when searching a design space, or any space for that matter, to be able to recognize when the search has returned to a place it's already been. In TAC, not recognizing when a design has been seen before can lead to looping as the system continues attempting to repair the design.

The system can encounter the same design because there may be many paths to it, i.e. different combinations of operators may have the same effects. When operators commute, for example, the same design may be reached via the same operators in different orders. Alternatively, different operators may result in the same changes to a design object. Shortening a wall, for example, may have the same effect as widening an adjacent doorway. The two designs below, generated in experiments on the Horner house, show a more complex example of TAC's reaching the same design via different modification steps.

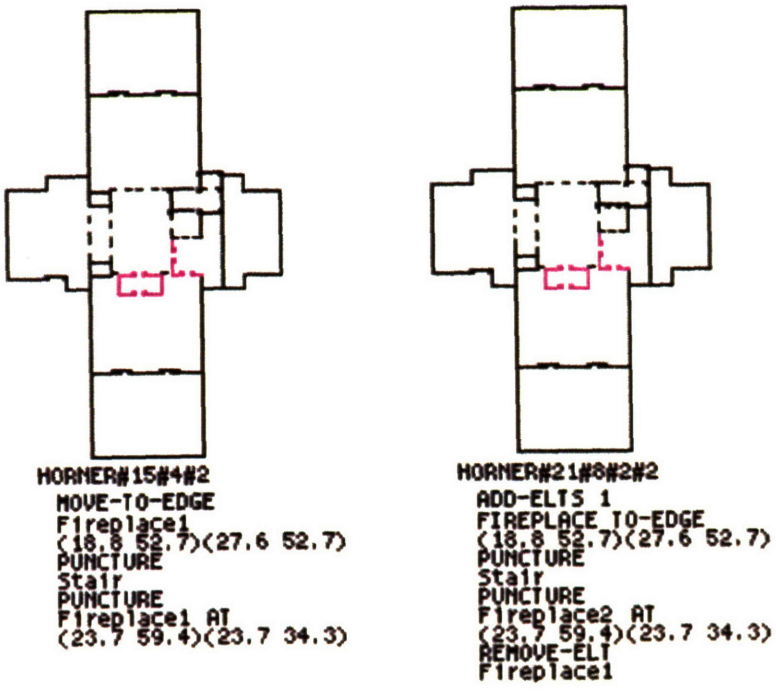



**Figure 7.27:** Same designs generated via different modification steps.[1]

1. In the experiments described in this chapter, the stair `screenify` operator has been replaced by `puncture`.

Recognizing that two designs are the same is difficult because designs that differ very slightly are often still considered the "same", even though they are not strictly identical. Slight differences can arise when design objects have real-valued attributes.[2] The designs shown below, for example, differ only in that the stairs are punctured in slightly different locations.

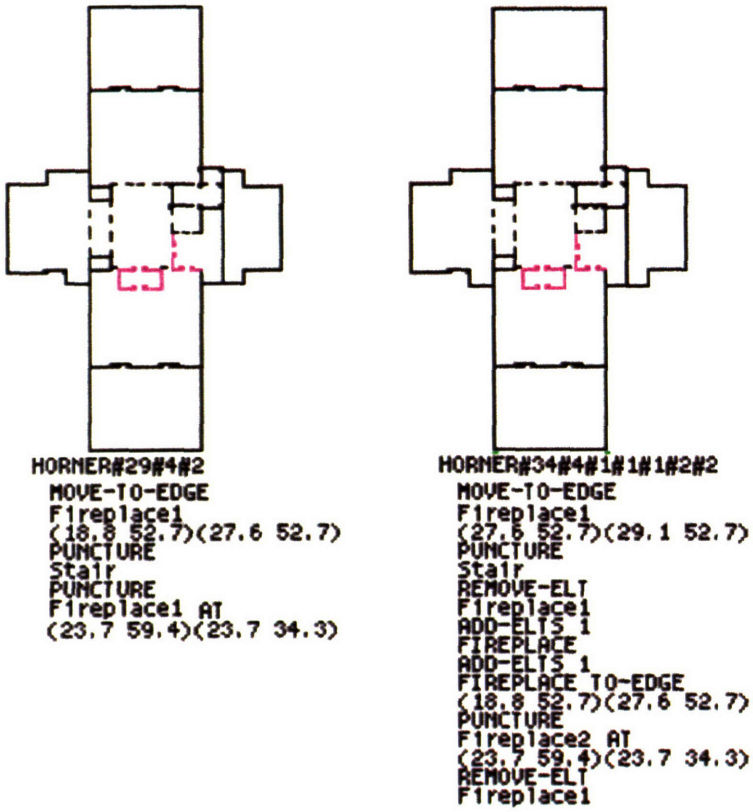



**Figure 7.28:** Comparison of solutions with stair punctured in slightly different locations.

To consider these two designs the same, we must define equivalence relations. Two designs might be the same, for example, if their edges are the same; two edges might be the same if their endpoints are within a specified $\varepsilon$ of each other. Defining these equivalence relations is a nontrivial knowledge engineering problem, though even simple relations will improve program performance.

Given how difficult it is to recognize equivalent designs, it is tempting to look for equivalent suggestions instead: If a suggestion is proposed that is the same as one in a design's derivation history, assume the design is equivalent to a predecessor and prune the suggestion.[3] We tried this simple lookbehind mechanism and found that it did cut down on extraneous nonminimal solutions. In general, however, such a lookbehind mechanism is not a good idea because (1) defining equivalence for suggestions is just as difficult as for designs, and (2) the design context in

2. By "attribute" we mean a physical characteristic of a design element, e.g. size, location, materials.
3. A design's derivation history is a list of the repairs carried out to generate the design from the original design.

which the second suggestion is proposed may be genuinely different, and pruning the suggestion may cause a solution to be missed. (See experiment 5 in Section 7.6 for lookbehind results.)

### 7.4.2 Limiting Iteration

Being able to recognize when search has returned to a previously generated design is sufficient to guarantee termination if the design space is finite. With real-valued attributes, however, the design space is infinite. Two methods ensure that a system such as TAC terminates its search of such a space: discretizing and bounding the space, or employing an iteration limit.

The space of designs can be discretized at design equivalence-testing time, for example, by defining equivalence relations, as suggested earlier, for real-valued design object attributes. The space also can be discretized at design generation time by discretizing the design operators. A design element to be added at a particular location can be centered on the nearest point in a predefined grid, for example.

The space of designs can be bounded by placing bounds on design object attributes and by insisting that operators keep attribute values within those bounds. The square footage for a design could be bounded, for example, by specifying that it not change by more than a given amount.

Discretizing and bounding a space of designs will ensure termination. In lieu of both of these, however, nontermination may be a problem. In the current version of TAC, the design space is bounded, but not discretized. TAC's set of operators bound the space by never changing the footprint of a design, but do not discretize the space. They position design elements on currently existing edges in the design. If intervening operators do not change the edges in a design, subsequent positioning or repositioning of design elements will occur at the same locations. Under these circumstances the space appears discretized, but intervening operators can change the edges in a design. TAC's equivalence testing also does not discretize the space: two designs are the same if their edges are identical and if their design elements have identical types, sizes, and locations. Because TAC's search is bounded but not discretized, it relies on an iteration limit to guarantee termination.[4]

Two examples below show designs that had to rely on an iteration limit because the combinations of add, remove, move, and puncture operators modified edges in such a way that the designs varied slightly from previously generated designs. As a result, TAC did not recognize the designs as duplicates and continued with its repair cycles.

4. An iteration limit also may be pragmatic when searching a finite, but very large space.


**Figure 7.29:** Two designs stopped at iteration limit with sequential control structure.

Long modification sequences proved more of an issue for the sequential control structure than for the other two control structures. This situation arises because, without pruning of conflicting suggestions, more modifications are carried out and those modifications change the size and location of a design's edges. As mentioned above, without equivalence relations that discretize edge sizes and locations, very similar designs will not be regarded as the same, and TAC will continue repairing each of them. When the sequential control structure can take advantage of synergies in an optimal goal order, long modification sequences and possible nontermination are not issues because the edge-changing modifications are not carried out.

## 7.5 Summary

Sequential-with-lookahead combines the benefits of the concurrent and sequential control structures: It can spot conflict and synergy opportunities, as in the concurrent control structure, and it proposes context appropriate suggestions for each new design, as in the sequential control structure. Its lookahead mechanism decreases potential looping behavior and sensitivity to goal order. The sequential-with-lookahead control structure, thus, is the most efficient of the three in searching the suggestion and design spaces. It does not, however, necessarily find the most complete solution set: Because the sequential control structure is allowed to temporarily violate goals while searching, it may find additional solutions. For this reason, the best control structure for generating solutions when goals interact is the sequential-with-lookahead control structure with an option for relaxing lookahead when desired. If nonminimal solutions are a priority, relaxing lookahead should be accompanied by ordering goals to take advantage of operator synergies when possible.

The table below summarizes the features of the control structures.

| | Sequential-with-lookahead | Sequential | Concurrent |
|---|---|---|---|
| detects synergy & conflict | ✓ | × | ✓ |
| proposes suggestions once per goal | × | × | ✓ |
| proposes only applicable suggestions | ✓ | ✓ | × |
| has small branch factor at root | ✓ | ✓ | × |
| finds minimal solutions without substitution (breadth- vs depth-first) | ✓ | ✓ | × |
| finds extra solutions (not necessarily minimal) | × | ✓ | × |

**Table 7.1:** Comparing control structure features.

## 7.6 Experiments

Since control structure behavior was not analytically determinable, we ran a series of experiments for each of the three control structures: sequential-with-lookahead, sequential, and concurrent. We chose a set of interacting goals, then varied goal order and sets of design modification operators. For each control structure, goal order, and operator set, we compared efficiency and solution sets. We compared efficiency with a measure of relative efficiency uniform across all the control structures: how many suggestions were proposed and how many were pruned, and how many designs were generated. We compared solution sets by looking at number, identity, and quality of solutions generated.[1] Finally, as mentioned in Section 7.4.1, we tested each control structure with a simple lookbehind mechanism. Below is a summary of the experiments.

Design

We used the Horner design.

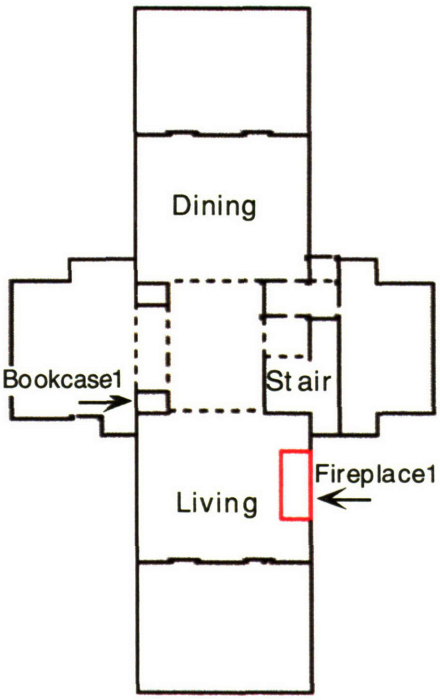



**Figure 7.30:** Territory model for Horner.

1. Recall that we measure the quality of a design by how many modification steps it is from the original design; the fewer the steps, the better the design.

Goals

We chose a set of interacting goals:

```
(<goal: (visible-center Living Dining) true>
 <goal: (visually-open Living Dining) true>
 <goal: (fireplace-on-interior-edge of Living) true>
 <goal: (fireplace-count in Living) 1>
 <goal: (fireplace-count in Horner) 1)
```

Four of the six kinds of interaction occur in this goal set. There are predictable synergies between puncture operators for the `visible-center` and `visually-open` goals; predictable conflicts between removing a fireplace for the `visually-open` and `visible-center` goals and keeping fireplace count at one; unpredictable conflict between the `visually-open` goal and having a fireplace on an interior edge. There also is interaction between the goals that deal with the fireplace: predictable conflict between adding a fireplace to an interior edge and maintaining one fireplace in the Living territory and in the entire design; obvious synergy between having one fireplace in the Living territory and one in the entire design.

As discussed in Section 7.4, the above goals are in an optimal order because goals with synergistic operators, i.e. that will satisfy more than one goal, precede goals with which they interact.

Also as discussed in Section 7.4, the following goal order is nonoptimal because goals with synergistic operators follow goals with which they interact:

```
(<goal: (visually-open Living Dining) true>
 <goal: (visible-center Living Dining) true>
 <goal: (fireplace-count in Horner) 1>
 <goal: (fireplace-count in Living) 1>
 <goal: (fireplace-on-interior-edge in Living) true>)
```

Operators

We specified three modification operators for increasing visual openness between two things: puncture blocking elements, remove blocking elements, or move blocking elements to exterior edges of their territories. To simplify initial experiments 1 and 2, TAC employed only the first two of these operators. The operator that moves a design element to an exterior edge introduces an unpredictable synergy (between satisfying `visible-center` and `visually-open` goals) and an unpredictable conflict (between `visually-open` and `fireplace-on-interior-edge` goals). This conflict could be predictable if the relationship between interior and exterior edges were represented in the knowledge base. In experiments 3 and 4, we studied the affect of introducing this operator.

We defined the fireplace addition operator as follows: a fireplace is added to a territory on one of the territory's interior edges; a fireplace is added to a design without a specified location (i.e. the fireplace is added to the design's set of design elements, but not given a location).

Below is a summary of the parameters for each experiment, followed by summaries of each experiment.

| Experiment | control structure | goal order | visual-openness operators |
|---|---|---|---|
| 1a | sequential+look | optimal | puncture, remove |
| 1b | sequential | optimal | puncture, remove |
| 1c | concurrent | optimal | puncture, remove |
| 2a | sequential+look | nonoptimal | puncture, remove |
| 2b | sequential | nonoptimal | puncture, remove |
| 2c | concurrent | nonoptimal | puncture, remove |
| 3a | sequential+look | optimal | puncture, remove, move |
| 3b | concurrent | optimal | puncture, remove, move |
| 3c | sequential | optimal | puncture, remove, move |
| 4a | sequential+look | nonoptimal | puncture, remove, move |
| 4b | concurrent | nonoptimal | puncture, remove, move |
| 4c | sequential | nonoptimal | puncture, remove, move |
| 5a | sequential+look + lookbehind | nonoptimal | puncture, remove, move |
| 5b | sequential + lookbehind | nonoptimal | puncture, remove, move |
| 5c | concurrent + lookbehind | nonoptimal | puncture, remove, move |

**Table 7.2:** Summary of experiment parameters.

## Experiment 1

We ran each control structure with the optimal goal order and the puncture and remove operators for increasing visual openness.

| Experiment 1 | a sequential+look | b sequential | c concurrent |
|---|---|---|---|
| # solutions | 4 | 4 | 4 |
| # designs | 8 | 16 | 14 |
| # duplicate designs | 0 | 5 | 6 |
| # suggestions (+ lookahead) | 11 (+ 22) | 16 | 27 |
| # suggestions pruned | 3 | 0 | 10 |
| % suggestions pruned | 27.3 | 0 | 37.0 |
| # repair cycles | 3 | 5 | 3 |

**Table 7.3:** Results of experiments with optimal goal order.

The concurrent control structure generated extra designs due to its large branching factor at the start of the search, i.e. extra compound suggestions. See Section 7.3.3 for a discussion of this issue. The sequential control structure generated extra designs because it violated goals, then repaired the resulting designs. See Section 7.3.2.

Below are the four solutions found by each control structure.

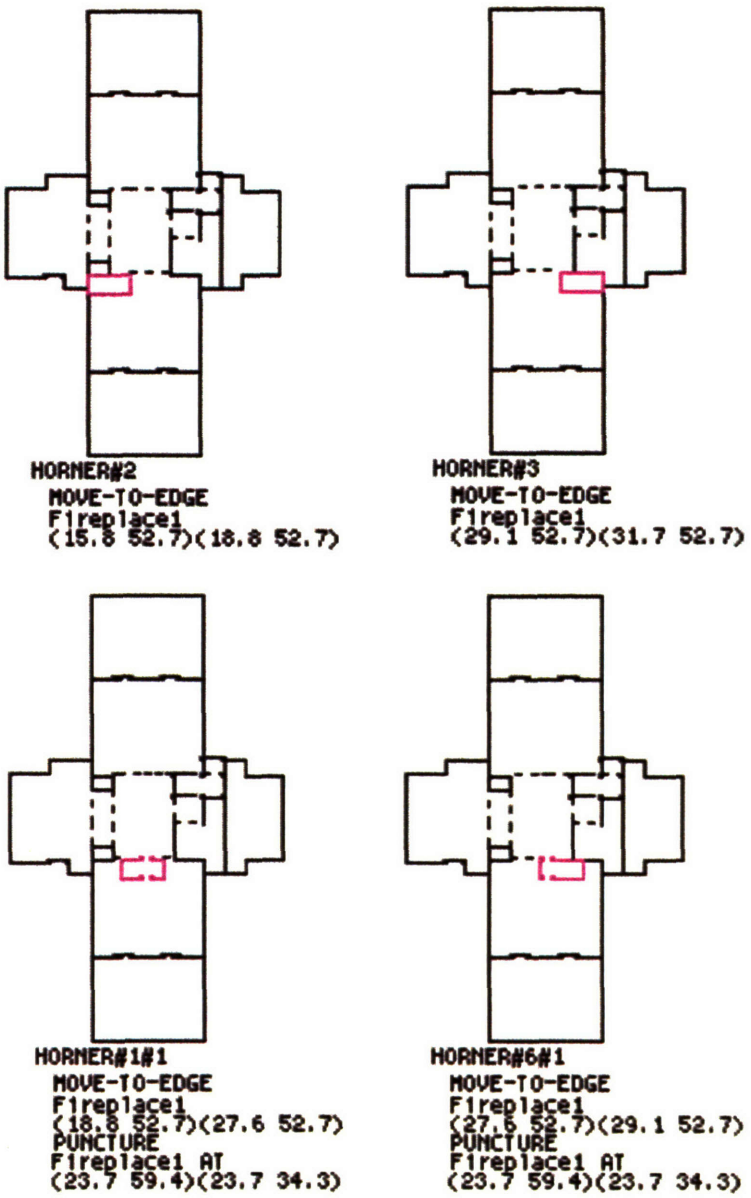



**Figure 7.31:** Four solutions for the control structures with optimal goal order.

## Experiment 2

We ran each control structure with the nonoptimal goal order and the puncture and remove operators for increasing visual openness.

| Experiment 2 | a sequential+look | b sequential | c concurrent |
|---|---|---|---|
| # solutions | 9 | 19 | 9 |
| # designs | 21 | 128 | 21 |
| # duplicate designs | 2 | 47 | 2 |
| # suggestions (+ lookahead) | 31 (+ 32) | 128 | 27 |
| # suggestions pruned | 10 | 0 | 10 |
| % suggestions pruned | 32.3 | 0 | 37.0 |
| # repair cycles | 3 | 23 | 3 |

**Table 7.4:** Results of experiments with nonoptimal goal order.

The sequential-with-lookahead and concurrent control structures produced the same four solutions as with the optimal goal order, plus five extra solutions which are shown below. As discussed in Section 7.4, these extra solutions are not minimal solutions.

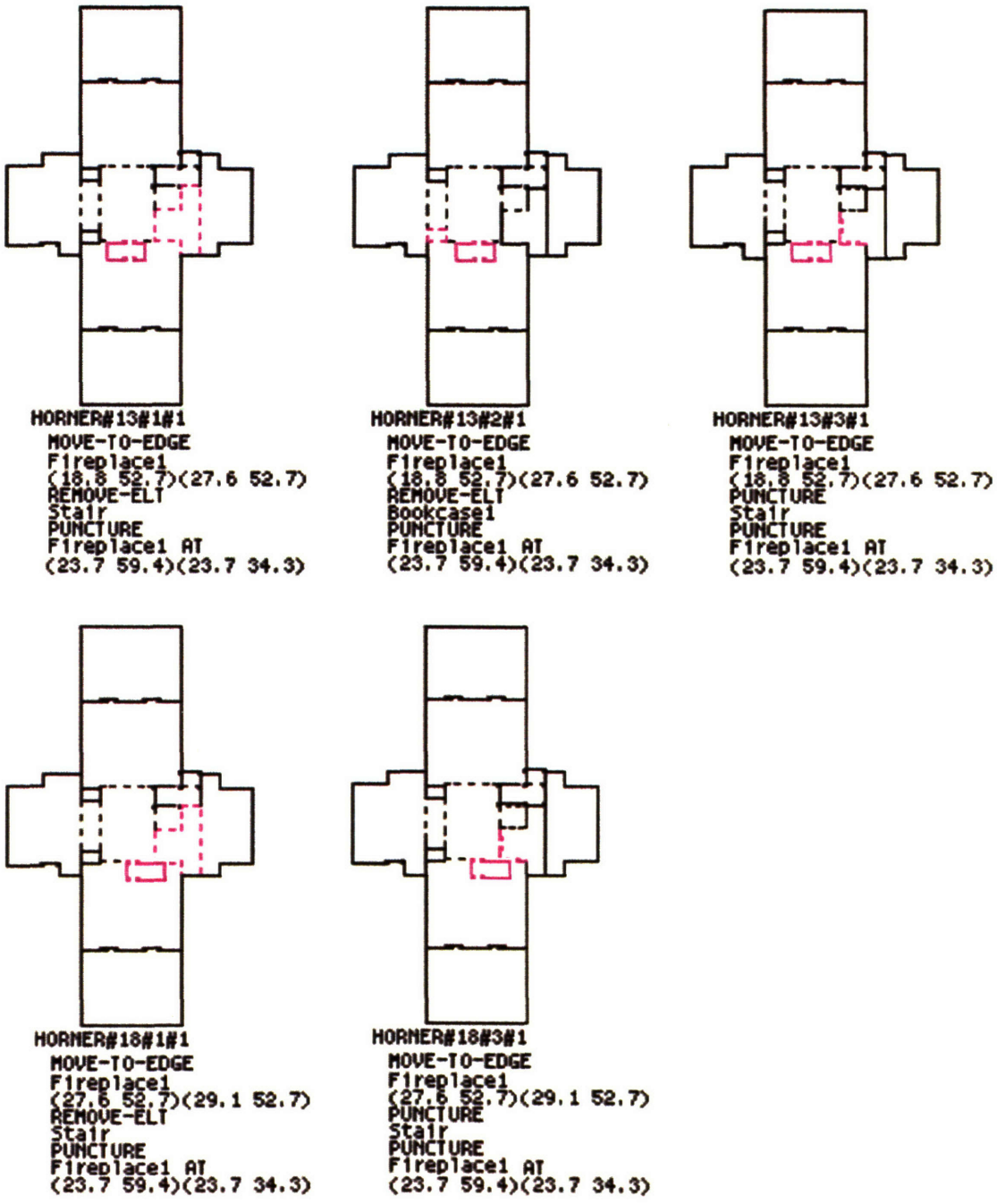



**Figure 7.32:** Five extra solutions found for sequential-with-lookahead and concurrent control structures with nonoptimal goal order.

The sequential control structure produced virtually the same nine solutions as the other two control structures (one has a fireplace moved and punctured in two places rather than one). It also produced ten extra solutions. Four of those are genuinely new designs, similar to two other solutions, but with extra (unnecessary) modifications carried out first. See Figures 7.33 and 7.34. The six additional solutions are not significantly different from other solutions: they have slightly different fireplace locations, slightly different stair puncture locations, slightly different edge extensions, slightly different new territory boundaries. These solutions result from the difficulty of spotting the "same" designs when using real-valued positions for operators. Two such solutions were shown in Figure 7.28. See Section 7.4.1 for discussion of design equivalence.

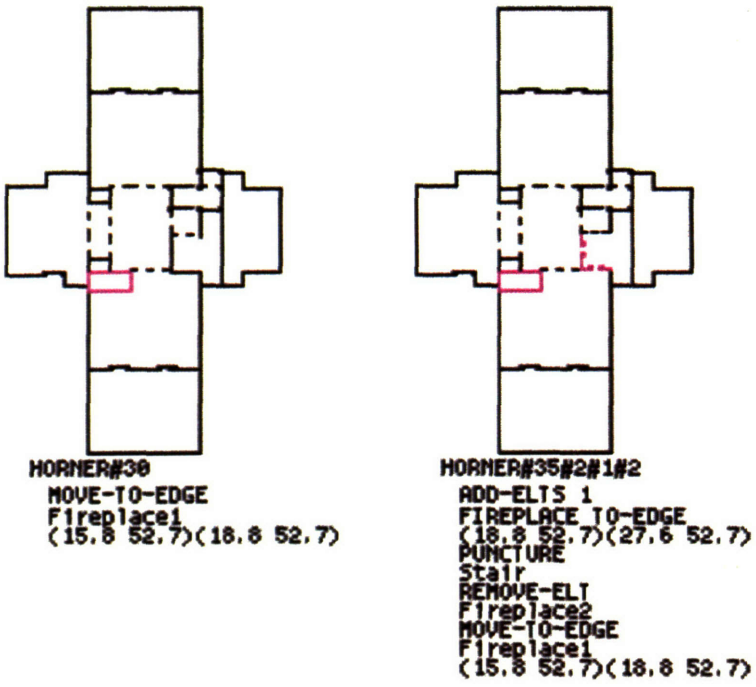



**Figure 7.33:** Comparison of two solutions for sequential control structure:
one on left was also found by other two control structures;
one on right has extra unnecessary modifications.

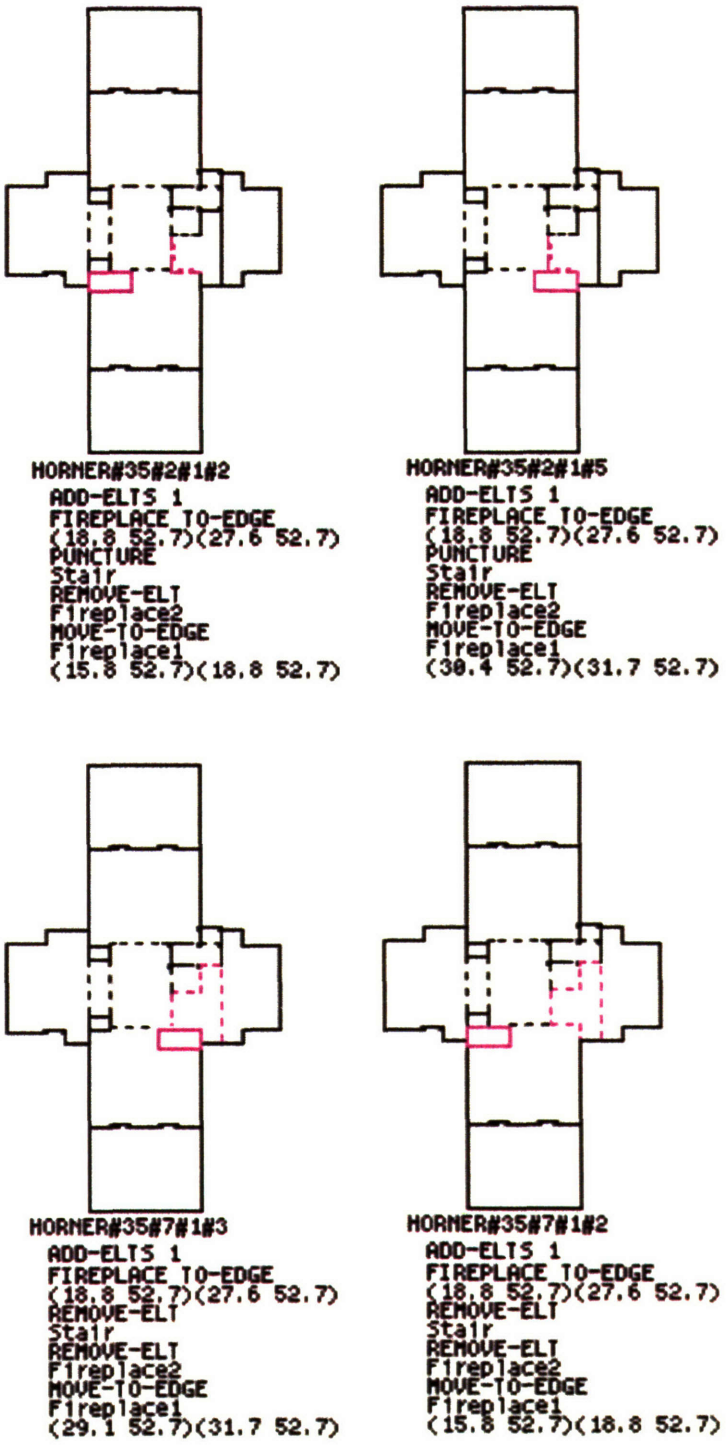



**Figure 7.34:** Extra new solutions for sequential control structure with nonoptimal goal order.

## Experiment 3

We introduced an operator that causes both unpredictable synergy and unpredictable conflict: move a design element to an exterior edge as a method for increasing visual openness. The table below summarizes the results of running each control structure with the additional move operator and an optimal goal order.

| Experiment 3 | a<br>sequential+look | b<br>sequential | c<br>concurrent |
|---|---|---|---|
| # solutions | 4 | 4 | 4 |
| # designs | 8 | 16 | 18 |
| # duplicate designs | 0 | 5 | 10 |
| # suggestions (+ lookahead) | 11 (+ 26) | 0 | 31 |
| # suggestions pruned | 3 | 0 | 10 |
| % suggestions pruned | 27.3 | 5 | 32.3 |
| # repair cycles | 3 | | 3 |

**Table 7.5:** Results of adding move to exterior edge as means of increasing visual openness; optimal goal order.

The results are the same for the sequential-with-lookahead and sequential control structures running without the move operator: With an optimal goal order, the `visually-open` goal is always satisfied by the `visible-center` operators, so no suggestions are proposed for it. The concurrent control structure generated more designs in the presence of the move operator because it proposed compound suggestions that included moving the fireplace to an exterior edge after modifying the design to satisfy the `visible-center` goal. The resulting designs were duplicates because the first modification satisfied both goals, rendering the move modification unnecessary.

## Experiment 4

We included the move operator and ran the control structures with the nonoptimal goal order. Extra solutions and designs are due to looping between moving the fireplace to an interior edge and moving the fireplace to an exterior edge to satisfy the `visually-open` goal. See Section 7.4 for a discussion of looping and termination issues.

| Experiment 4 | a<br>sequential+look | b<br>sequential | c<br>concurrent |
|---|---|---|---|
| # solutions | 11 | 37 | 22 |
| # designs | 47 | 339 | 96 |
| # duplicate designs | 10 | 122 | 22 |
| # suggestions (+ lookahead) | 62 (+ 148) | 339 | 92 |
| # suggestions pruned | 15 | 0 | 33 |
| % suggestions pruned | 24.2 | 0 | 35.9 |
| # repair cycles | 5 | 48 | 11 |

**Table 7.6:** Results of adding move to exterior edge as means of increasing visual openness; nonoptimal goal order.

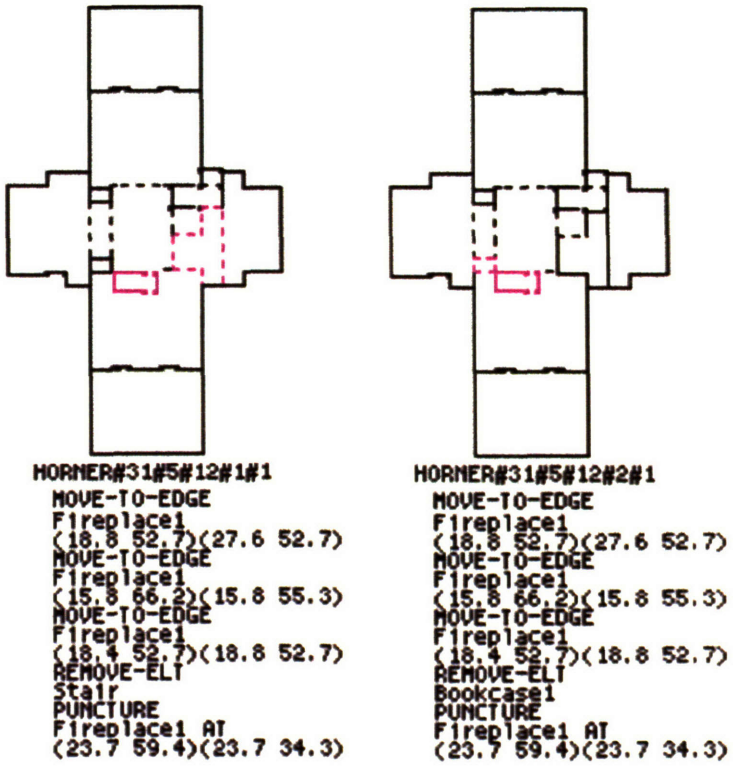



**Figure 7.35:** Two extra solutions for control structures with move operator; nonoptimal goal order.

An additional eleven solutions for the concurrent control structure resulted from a large branching factor at the beginning of the search, combined with the difficulty of identifying designs that are the "same". The additional solutions, two of which are shown below, were all nonminimal solutions very similar to other solutions.

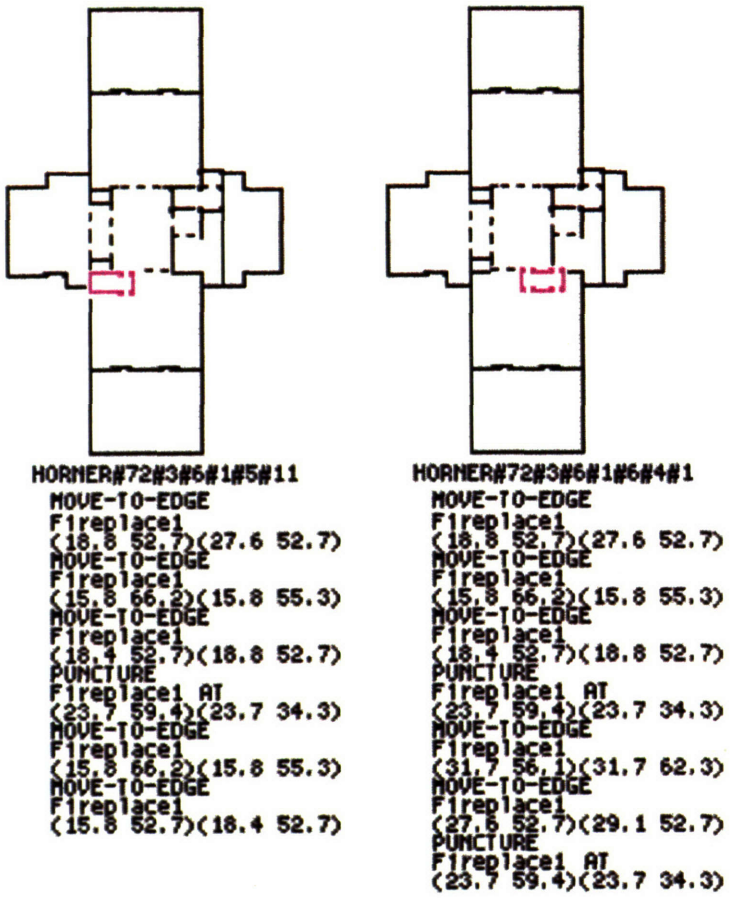



**Figure 7.36:** Examples of extra solutions for concurrent with move operator.

Finally, the additional solutions found by the sequential model differed only slightly from those found by the two other control structures. They had many extra modifications, however. Two are shown below in Figure 7.37. In addition, ten repair cycles were terminated when an iteration limit was reached without a solution or duplicate design. See Section 7.4.2 for a discussion of termination issues.

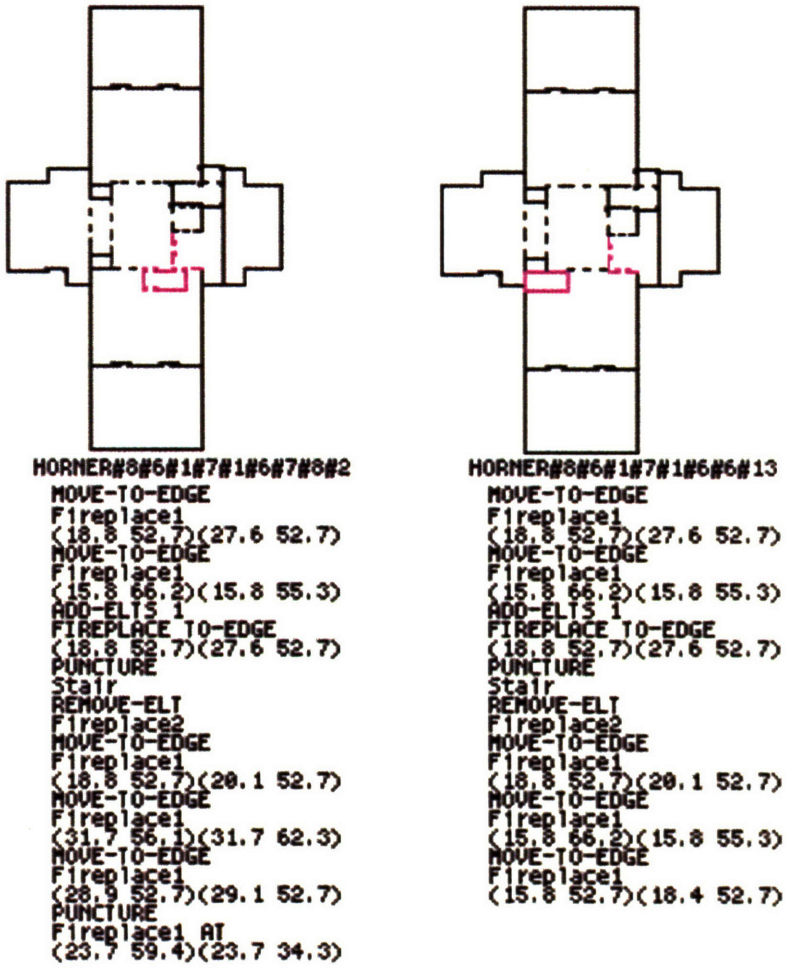



**Figure 7.37:** Examples of extra solutions for sequential with move operator.

## Experiment 5

We added a simple lookbehind mechanism to each of the control structures. Proposed suggestions were pruned if they were identical to suggestions in the design's modification history, i.e. if the identical suggestion had been carried out already. By "identical" we mean that the suggestions proposed the same modification to the same design object.

| Experiment 5 | a<br>sequential+look | b<br>sequential | c<br>concurrent |
|---|---|---|---|
| # solutions | 11 | 29 | 16 |
| # designs | 21 | 253 | 59 |
| # duplicate designs | 8 | 67 | 10 |
| # suggestions (+ lookahead) | 62 (+134) | 276 | 66 |
| # suggestions pruned | 17 | 23 | 23 |
| % suggestions pruned | 27.4 | 8.3 | 34.8 |
| # repair cycles | 5 | 40 | 8 |

**Table 7.7:** Results of experiments with lookbehind and nonoptimal goal order.

Adding lookbehind did not change the set of solutions for the sequential-with-lookahead control structure. It did reduce the number of designs and the number of duplicates. With the sequential and concurrent control structures, it reduced the number of nonminimal solutions, the number of designs, and the number of duplicates.

One of the nonminimal solutions generated by the sequential control structure but avoided when lookbehind was employed is shown in Figure 7.38. See discussion in Section 7.4 about why lookbehind is not a good technique for limiting search of a design space.

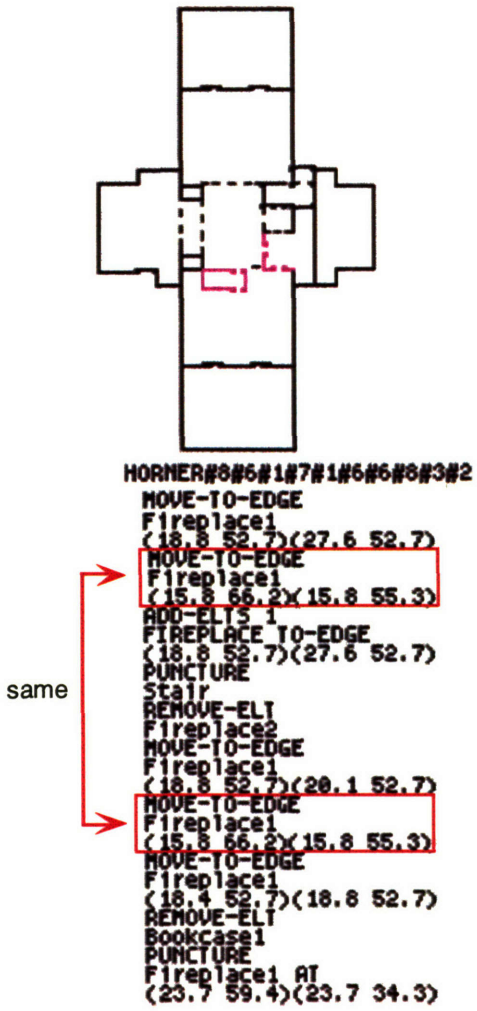



**Figure 7.38:** Nonminimal solution avoided by sequential with lookbehind.
Identical suggestions are boxed; lookbehind pruned second suggestion.

# Chapter 8

# Exercises in Architecture

## 8.1 Design: Chatham House

Let's return to the Chatham house introduced in Section 5.4 and shown below.

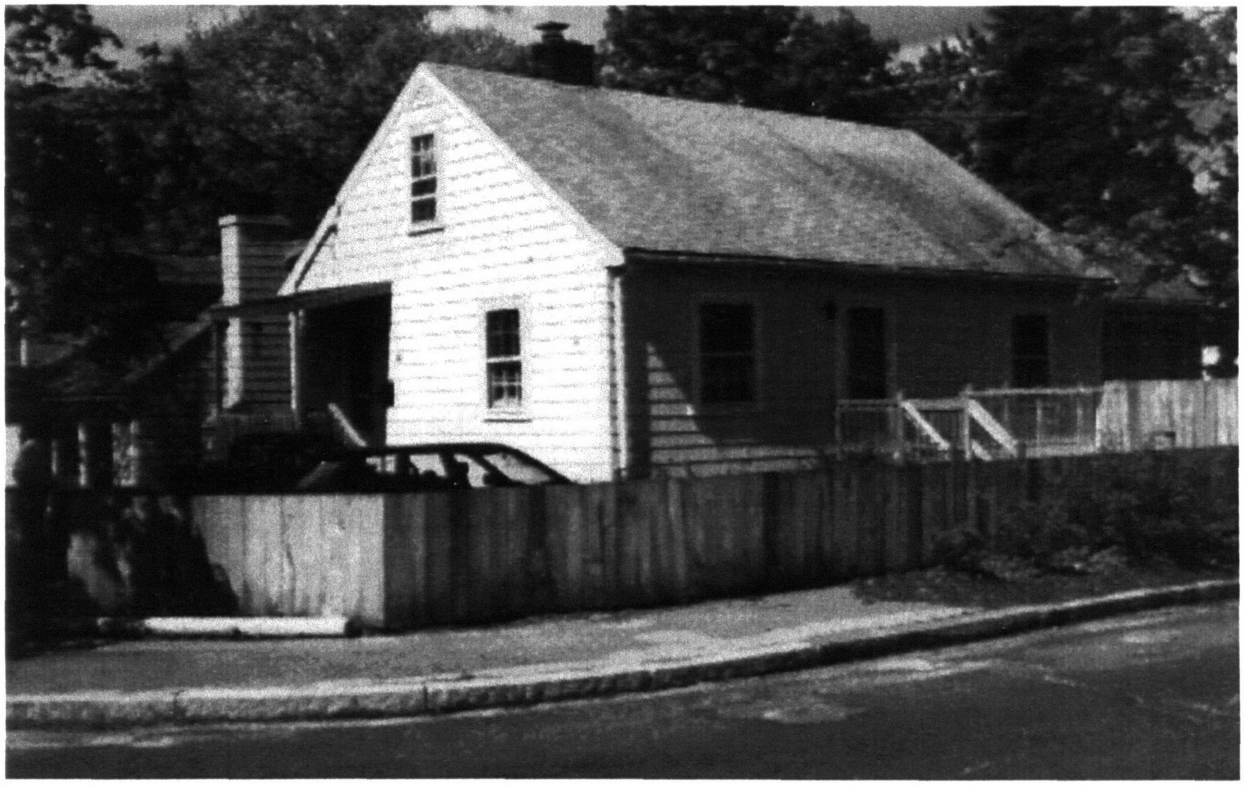


**Figure 8.1:** The Chatham house.

TAC was given a model of this house and a set of design goals defined by the owners and their architects, and in response proposed new designs. The designs, along with TAC's reasoning, are presented here to show that TAC finds plausible solutions to a real architectural design problem, and that it does so with breadth and generality. At the end of this section, several of TAC's solutions are compared to those proposed by the architects.

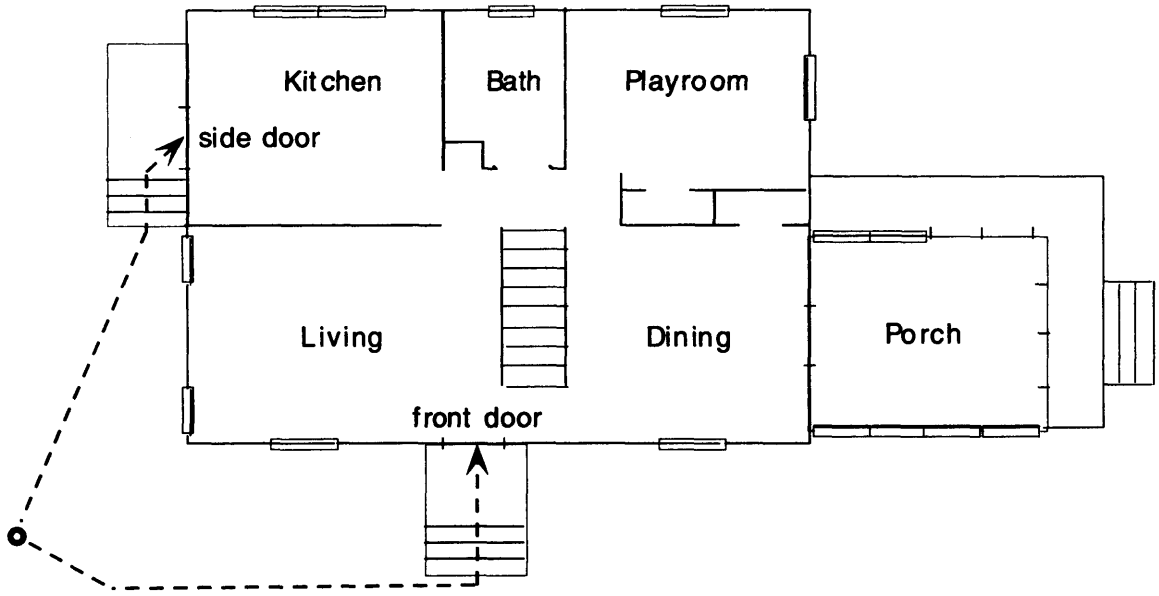



**Figure 8.2:** The Chatham house first floor and approach paths to exterior doors. The usual approach point is marked by **O**.

Four problems were identified:

- site: visitors approaching the house are not sure which door to use
- entry: the living room is not very private with respect to the front door
- territories: the rooms feel very isolated from one another
- use: the kitchen is too far from the dining room

Rephrasing the problems in terms of goals for physical access and visual openness:

- site: have one perceived main entry
- entry: have the living room visually semi-open and physically semi-accessible from the perceived main entry (i.e. have the living room only partially visible and reached via a somewhat crooked path)
- territories: have the rooms (main living spaces in particular) visually open from one another
- use: have the kitchen next to the dining room

Photographs included here illustrate the above problems. The two possible entries to the Chatham house are visible in the photograph shown in Figure 8.1. The photograph in Figure 8.3 shows the lack of privacy of the living room from the front door: a visitor at the front door sees the entire living room and walks directly into it. The photograph in Figure 8.4 illustrates the lack of visual openness between the living and dining rooms: the stair blocks most of the view between the rooms. The floorplan in Figure 8.2 shows that the kitchen is not next to the dining room.

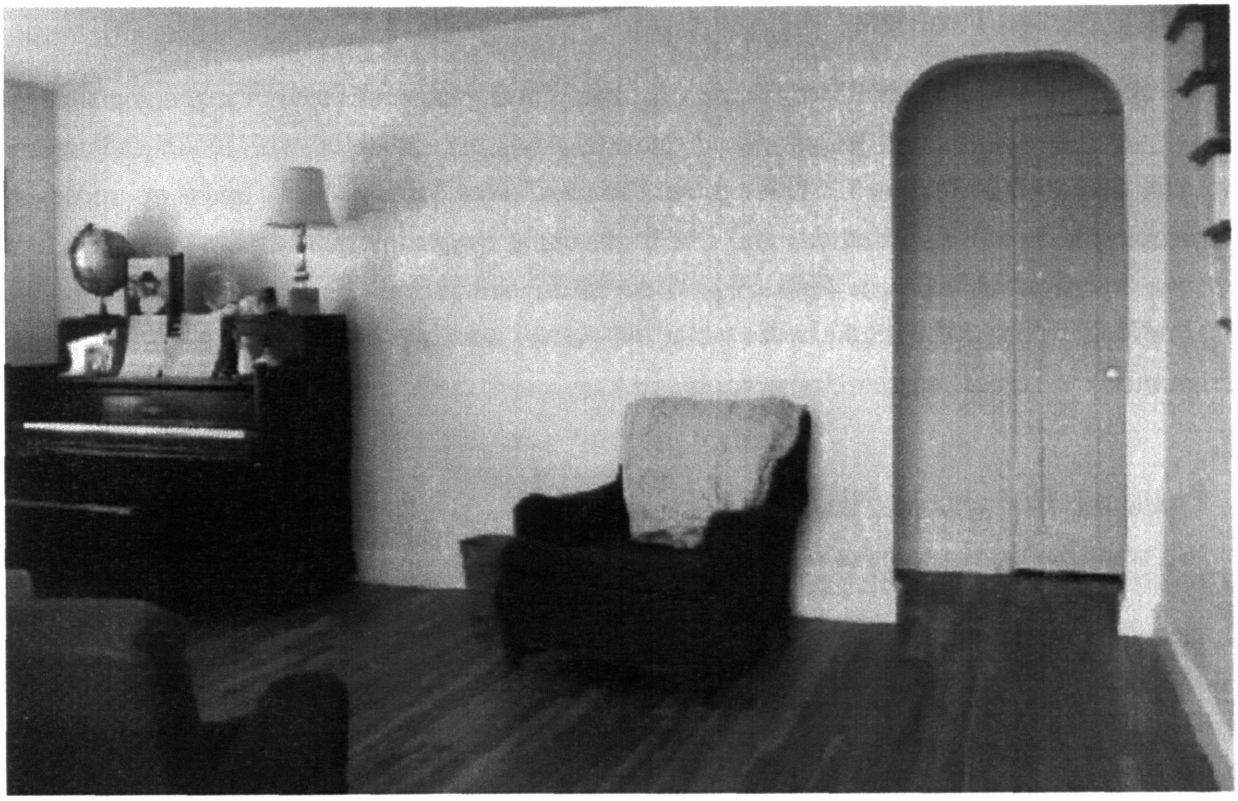


**Figure 8.3:** View from front door into living room.

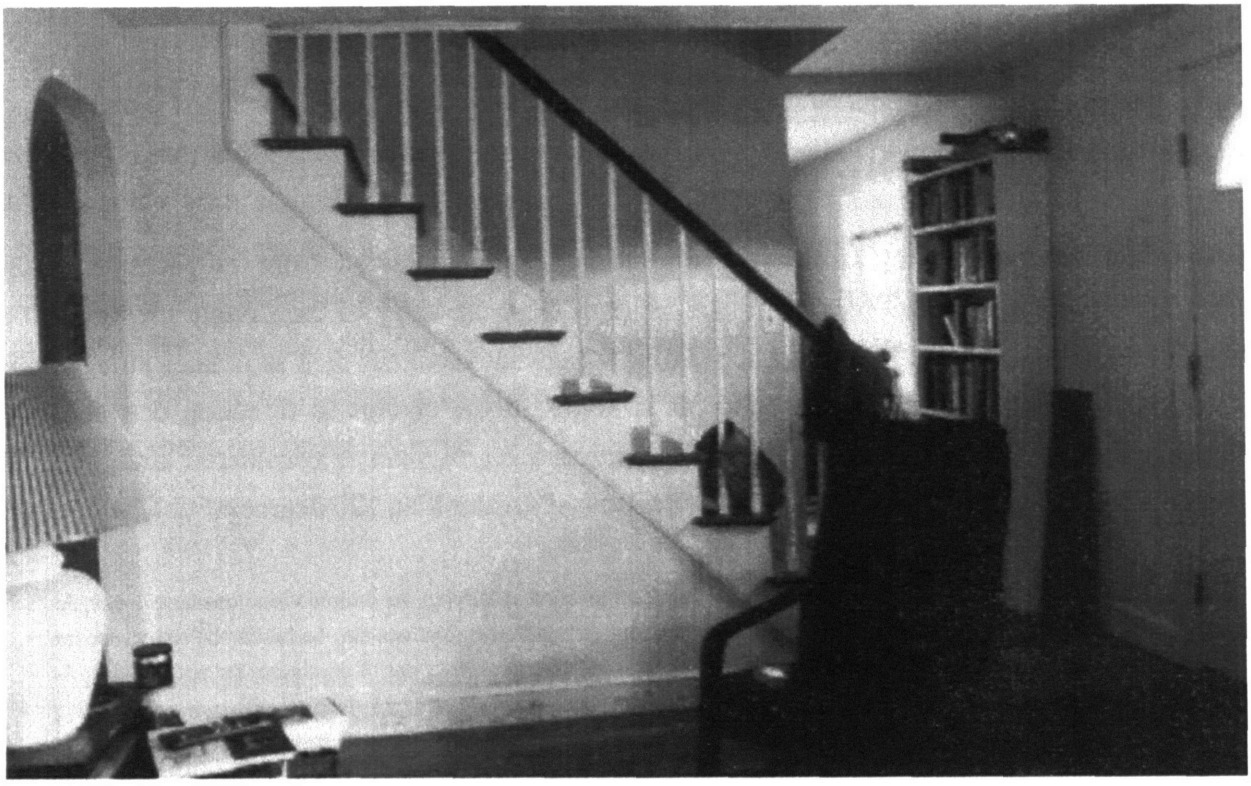


**Figure 8.4:** View from living room to dining room.

## Defining Design Problems

We gave TAC a design problem for the Chatham house which consisted of seven goals representing the above problems. TAC produced solutions using the sequential-with-lookahead control structure described in Section 7.3.1. We present some of those solutions here, and for purposes of exposition, we discuss the solutions and TAC's reasoning by grouping the design goals into four separate design problems—one for each problem mentioned above. At the end of this section we describe TAC's behavior with the seven goals and discuss the trade-offs between running a single design problem versus separate design problems.

Problem 1: One perceived main entry

We want the design to have one perceived main entry, so we specify the goal as in Section 5.4:

```
<goal: (one-perceived-main-entry Chatham) true>
```

Problem 2: Living visually semi-open, physically semi-accessible from perceived main entry

We want the Living territory to be visually semi-open, i.e. we want to see part but not all of it, from the perceived main entry. In particular, let's say that we'd like to see more than 30% but less than 80% of the Living territory. We represent this visual openness range by specifying these two goals:[1]

```
<goal:(gt (visual-openness Living from (perceived-main-entry Chatham) 0.3)
      true>
<goal:(lt (visual-openness Living from (perceived-main-entry Chatham) 0.8)
      true>
```

We also want the Living territory to be physically semi-accessible from the perceived main entry, by which we'll mean that we'd like the path from the entry to the Living territory to be "somewhat crooked". We'll say that a path is "somewhat crooked" if it is neither straight nor circuitous, and we represent these concepts in terms of the change in direction between two things, a design characteristic introduced in Section 3.2.1. A path is considered straight if its change in direction is less than 10 degrees, circuitous if greater than 120 degrees. For Chatham we

1. A single goal could be defined whose expression is a conjunction specifying the desired visual openness range. As noted in Section 7.3.3, however, if a goal expression is a conjunction, fewer solutions may be found: the clauses are not reasoned about separately, so no intermediate designs are created that may satisfy one of the clauses but not the other. As we'll see in this example, a solution would be missed. Finally, since the design characteristic `visual-openness` is invertible (see discussion in Section 5.2.2), the operators that increase or decrease it cannot guarantee a resulting value within a specified range. Representing upper and lower bounds separately allows the possibility of a solution found in two steps rather than one (i.e. a design satisfies one of the bounds, then is repaired to satisfy the other also).

specify the goals:[2]

```
<goal:(gt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          10) true>
<goal:(lt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          120) true>
```

Problem 3: Dining territory visually open from Living territory

We want the Dining territory to be visually open from the Living territory. We specify a goal using the design characteristic `visually-open` introduced in Section 3.2.1:[3]

```
<goal: (visually-open Dining from Living) true>.
```

Problem 4: Kitchen adjacent to Dining

We introduce the design characteristic, `use-adjacent`. Recall from Section 3.1 that TAC represents territories and their functional uses separately. So a territory named "Kitchen" does not carry along information about activities that take place in a Kitchen; instead the information is represented by a use space model which pairs territories and activities. The design characteristic `use-adjacent` takes two activities and checks a use space model for adjacency of territories with those activities. By saying that we want Kitchen adjacent to the Dining, we're actually saying that we want kitchen activities adjacent to dining activities, so we define this goal for Chatham:

```
<goal: (use-adjacent kitchen dining) true>.
```

We gave TAC the above goals in the order presented here, which is an order proposed by the architects: They thought about the larger issue of site entry, then house entry, then territory boundaries, ending with assigning uses to territories. Though they did not work on these problems linearly throughout the design process, initially ordering the problems in this way allowed them to narrow their search for solutions by fixing early in the process those aspects they deemed most important. For example, choosing a particular perceived main entry prior to considering the Living territory's visual openness from the entry meant that they could consider fewer alternative designs for the visual openness issue.

---

2. The range is represented with two goals rather than one for the same reasons as discussed for a visual openness range.
3. Recall that a territory is visually open from another territory if more than 60% of it is visible. In the same way that the design characteristic `visually-open` abstracts this visual openness threshold, a sophisticated user could define design characteristics for other thresholds. The goals in the second design problem above then could be specified in terms of such abstractions; e.g. `visually-closed` (defined as a visual openness value of less than 0.3) and `visually-very-open` (defined as a visual openness value of greater than 0.8).

## Evaluating Design Goals, Proposing and Performing Repair Suggestions

TAC's reasoning and selected solutions for each of the above design problems are presented here. Figure 8.5 shows the territory model TAC is given for the first problem. Also shown are the paths from the usual approach point to the front door and side door. TAC found the paths by searching the design's circulation model.

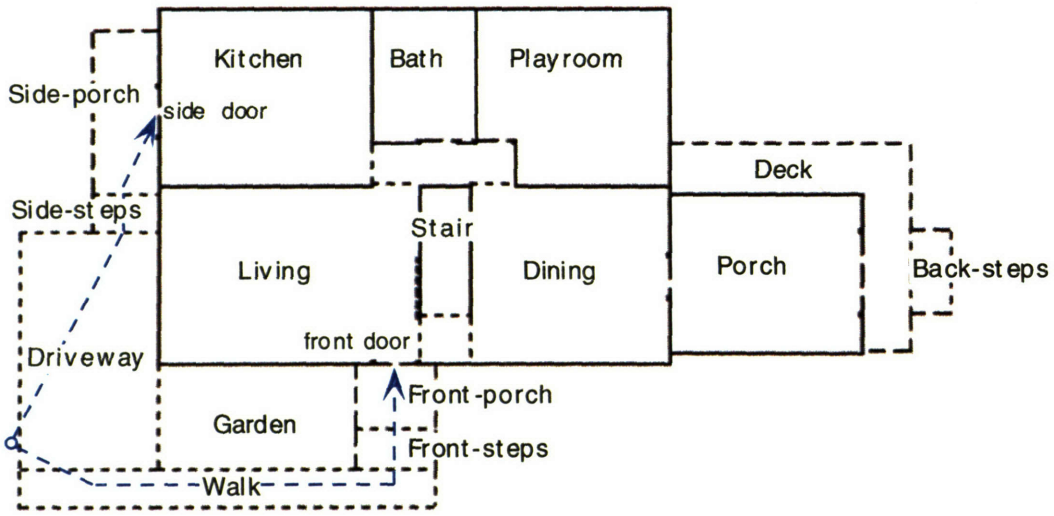



**Figure 8.5:** Territory model and approach paths for Chatham.

We begin by describing the design problem of having one perceived main entry, then proceed to subsequent design problems, using solutions to this first design problem as starting designs.

Problem 1: One perceived main entry

```
<goal: (one-perceived-main-entry Chatham) true>.
```

TAC finds that the `one-perceived-main-entry` goal is not satisfied: neither the front door nor the side door has a `perceived-main-entryness` value that dominates. As discussed in Section 5.4, TAC suggests making the front door the perceived main entry by moving it closer to the usual approach point and making the path to it straighter, replacing the side door with a window or a wall, removing the path to the side door, or making the side door not visible from the usual approach point. Several of these new designs are shown in Figure 8.6.[4] TAC also suggests making the side door the perceived main entry by moving it further from the usual approach point

4. Designs will be shown without the porch, deck, or back steps in order to simplify the figures.

making the side door the perceived main entry by moving it further from the usual approach point and making the path to it less straight, replacing the front door with a window or wall, removing the path to the front door, or making the front door not visible from the usual approach point. Several of these designs are shown in Figure 8.7. All the designs are shown in Appendix G.

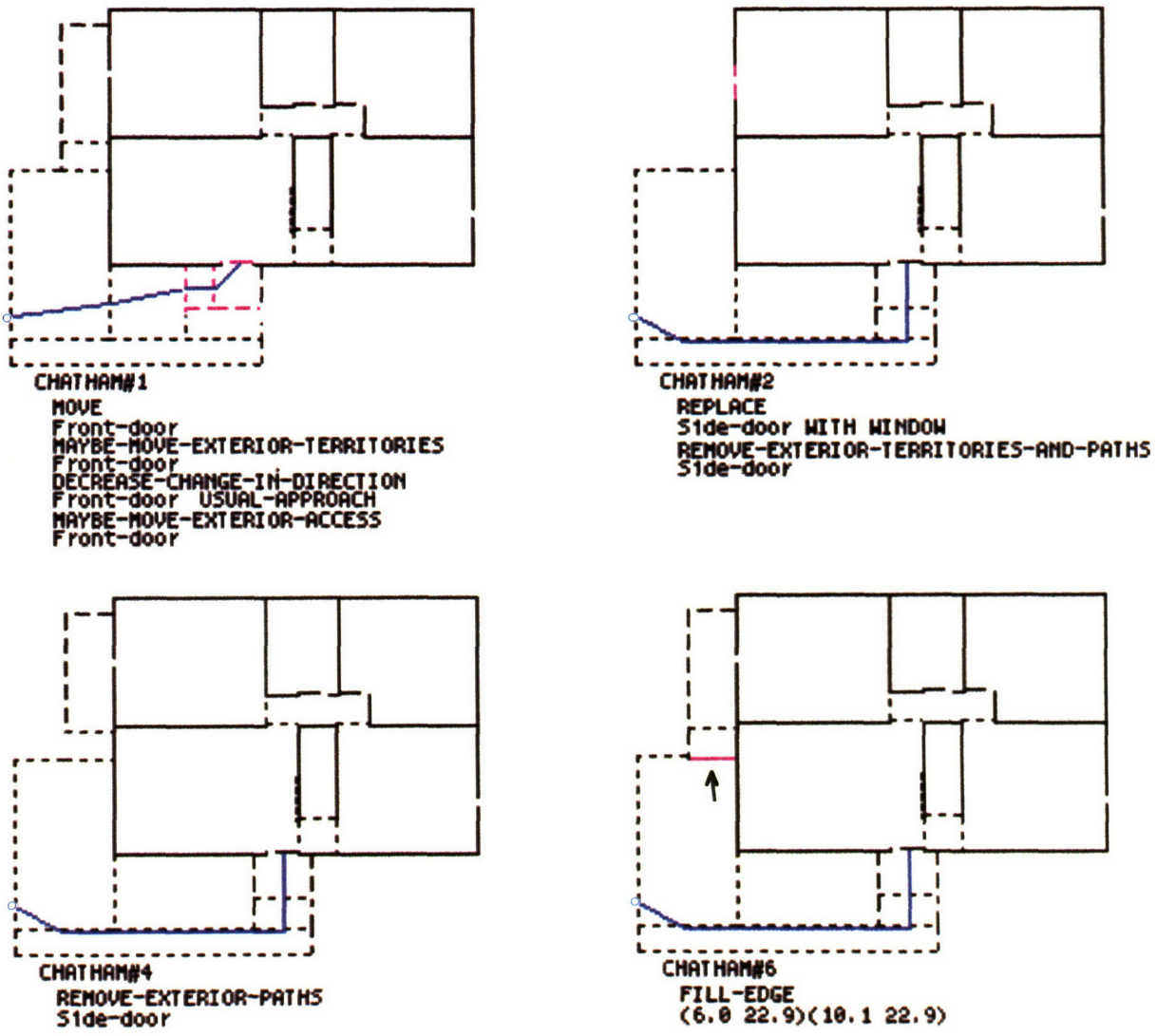



**Figure 8.6:** Designs TAC proposes with front door as perceived main entry. Path is from usual approach point (o) to front door; arrow in #6 shows new wall.

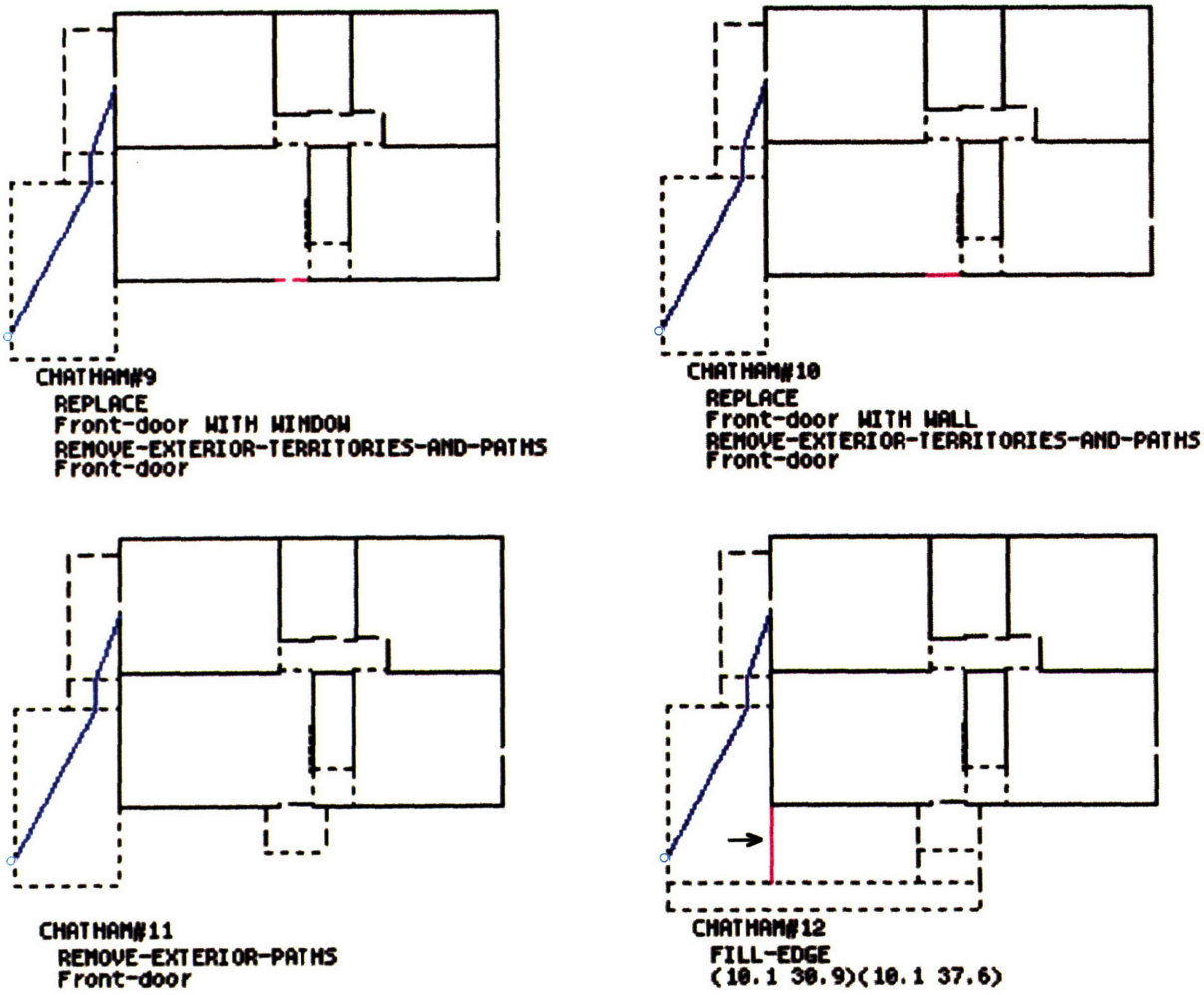



**Figure 8.7:** Designs TAC proposes with side door as perceived main entry.
Path is from usual approach point (o) to side door; arrow in #12 shows new wall.

Problem 2: Living visually semi-open, physically semi-accessible from perceived main entry

```
<goal:(gt (visual-openness Living from (perceived-main-entry Chatham) 0.3)
      true>
<goal:(lt (visual-openness Living from (perceived-main-entry Chatham) 0.8)
      true>
<goal:(gt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          10) true>
<goal:(lt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          120) true>
```

We illustrate TAC's reasoning on this design problem by showing its behavior on two very different solutions for the previous design problem. Chatham#2 has the front door as the perceived main entry; the side door has been replaced with a window. Chatham#10 has the side door as the perceived main entry; the front door has been replaced with a wall section.

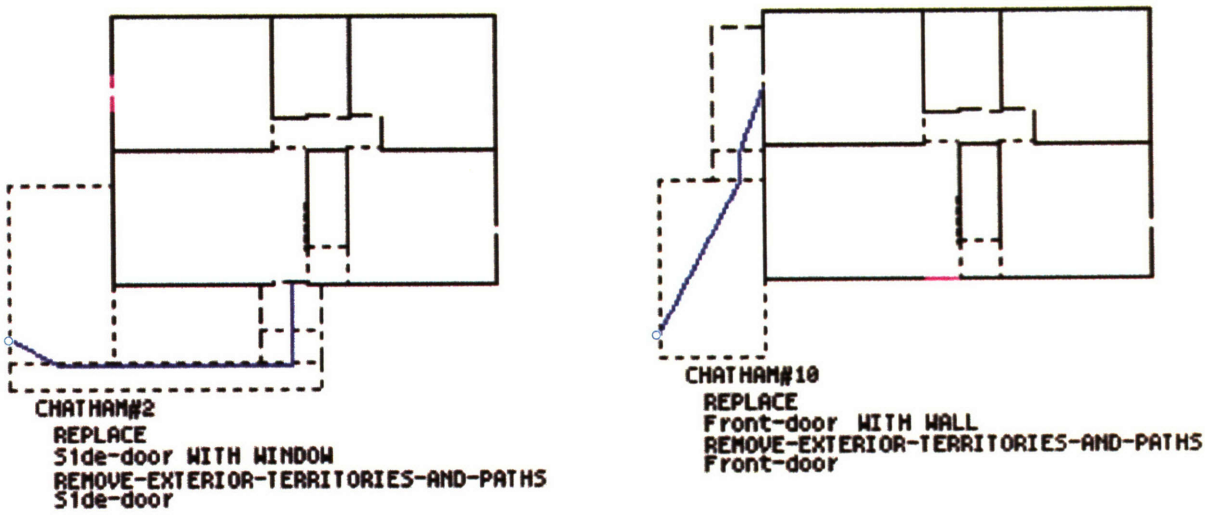



**Figure 8.8:** Two designs: perceived main entry is front door (left) or side door (right).

TAC evaluates the visual openness and change in direction goals for the Living territory with respect to the perceived main entry in each design, then proposes repair suggestions and creates new designs for the suggestions. It then repairs any designs that do not satisfy all goals.

Chatham#2

Step 1: Evaluating goals

TAC finds that two of the goals are not satisfied for Chatham#2: the Living territory is too visually open from the front door (value is 1.0), and the path to it from the front door is too straight (change in direction is 0 degrees).

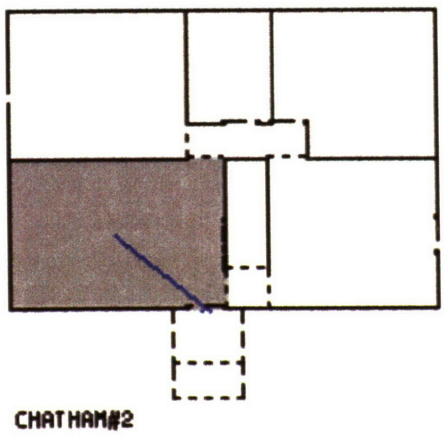



**Figure 8.9:** Chatham#2 visual openness and accessibility of Living from front door:[5] Shaded area is visible; * is a viewpoint; path is from front door to center of Living.

5. Designs will be shown without the driveway and front walk.

Step 2: Proposing and carrying out suggestions for visual openness to be less than 0.8

Since the visual openness of the Living territory is too high, TAC proposes decreasing it:

```
(decrease-value of (visual-openness of Living from Front-door)
                until visual-openness less than 0.8)
```

Seeking a way to accomplish this suggestion, TAC follows influence links which lead it to suggest increasing the opacity of any design elements between the Living territory and front door:

```
(increase-value of (opacity-of-elements-btw Living and Front-door)
                until visual-openness less than 0.8)
```

By checking its knowledge base, TAC finds that it can increase design element opacity by filling in any open edges with walls or screens:

```
(or (fill any open-edges-btw Living and Front-door)
    (screenify any open-edges-btw Living and Front-door))
```

It identifies four open edges between the Living territory and front door and proposes adding walls (filling) and screens (screenifying) at those locations:

```
(or (fill <edge: 26.44...>)
    (fill <edge: 23.56...>)
    (fill <edge: 23.56...>)
    (fill <edge: 23.56...>)
    (screenify <edge: 26.44...>)
    (screenify <edge: 23.56...>)
    (screenify <edge: 23.56...>)
    (screenify <edge: 23.56...>))
```

TAC has thus translated a goal stated in terms of the experiential quality visual openness into design modifications stated in terms of specific locations in the design.

TAC creates eight new designs, one for each suggestion above, adding walls and screens for the four open edges between the Living territory and the front door.

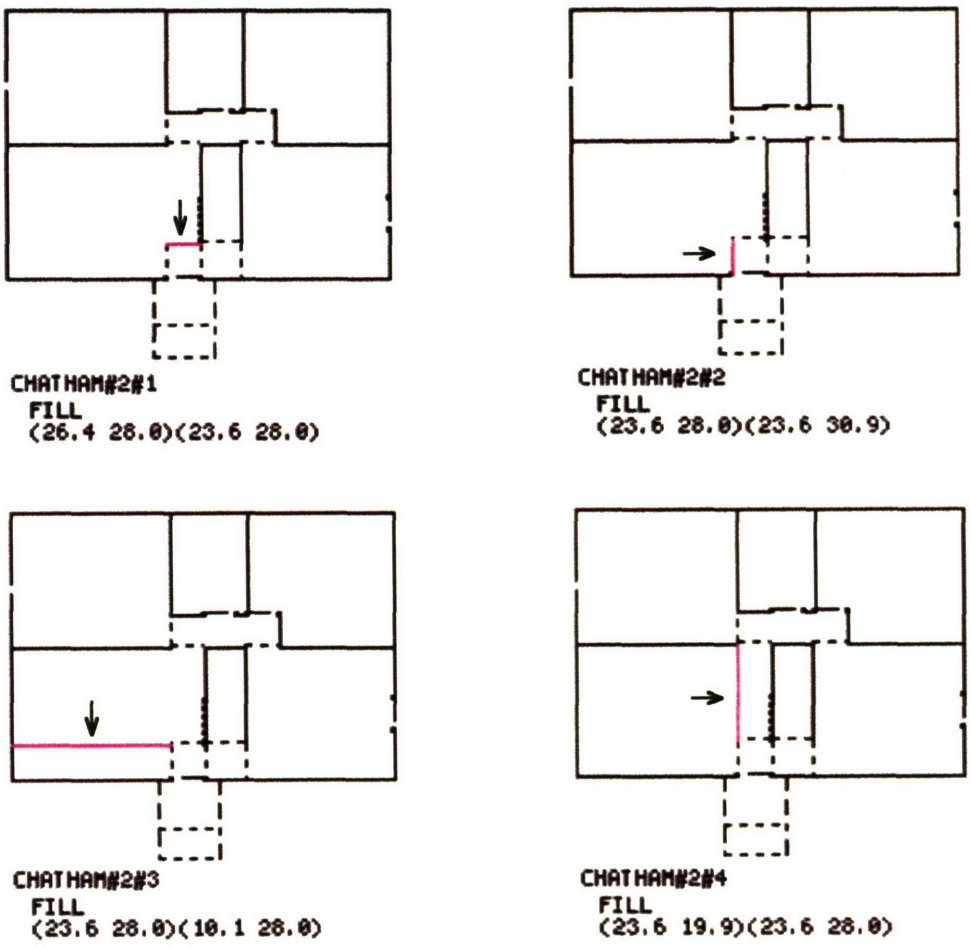



**Figure 8.10:** Decreasing visual openness by replacing open edges with walls. Arrows show locations of new walls.

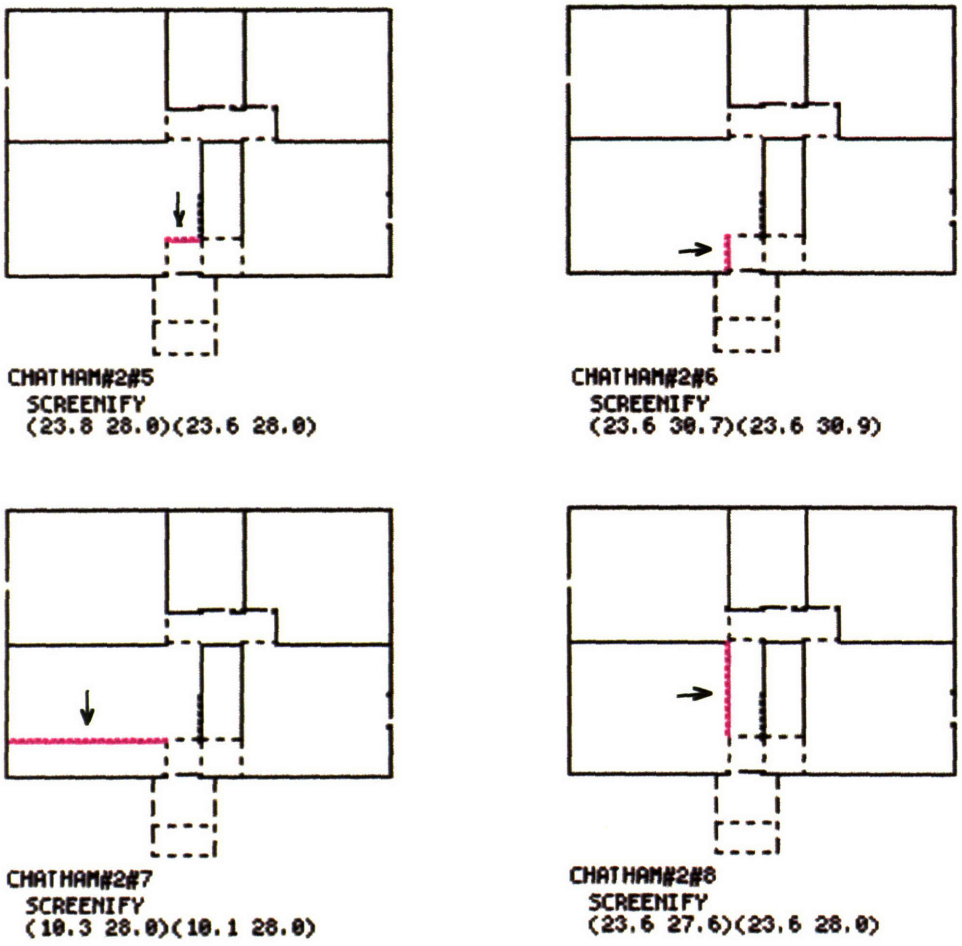



**Figure 8.11:** Decreasing visual openness by replacing open edges with screens. Arrows show locations of new screens.

Checking the visual openness values for the new designs, TAC finds that modifications yielding designs #4, #5, and #8 did not have the intended effect: the designs do not have visual openness values less than 0.8, so they are discarded.

Step 3: Proposing and carrying out suggestions for path change in direction to be greater than 10 degrees

Having created new designs for the first unsatisfied goal, TAC now proposes suggestions for the second unsatisfied goal: having the path from the front door to the Living territory not straight (i.e. with a change in direction of greater than 10 degrees). At this point, TAC is working with the five designs shown in Figure 8.12.

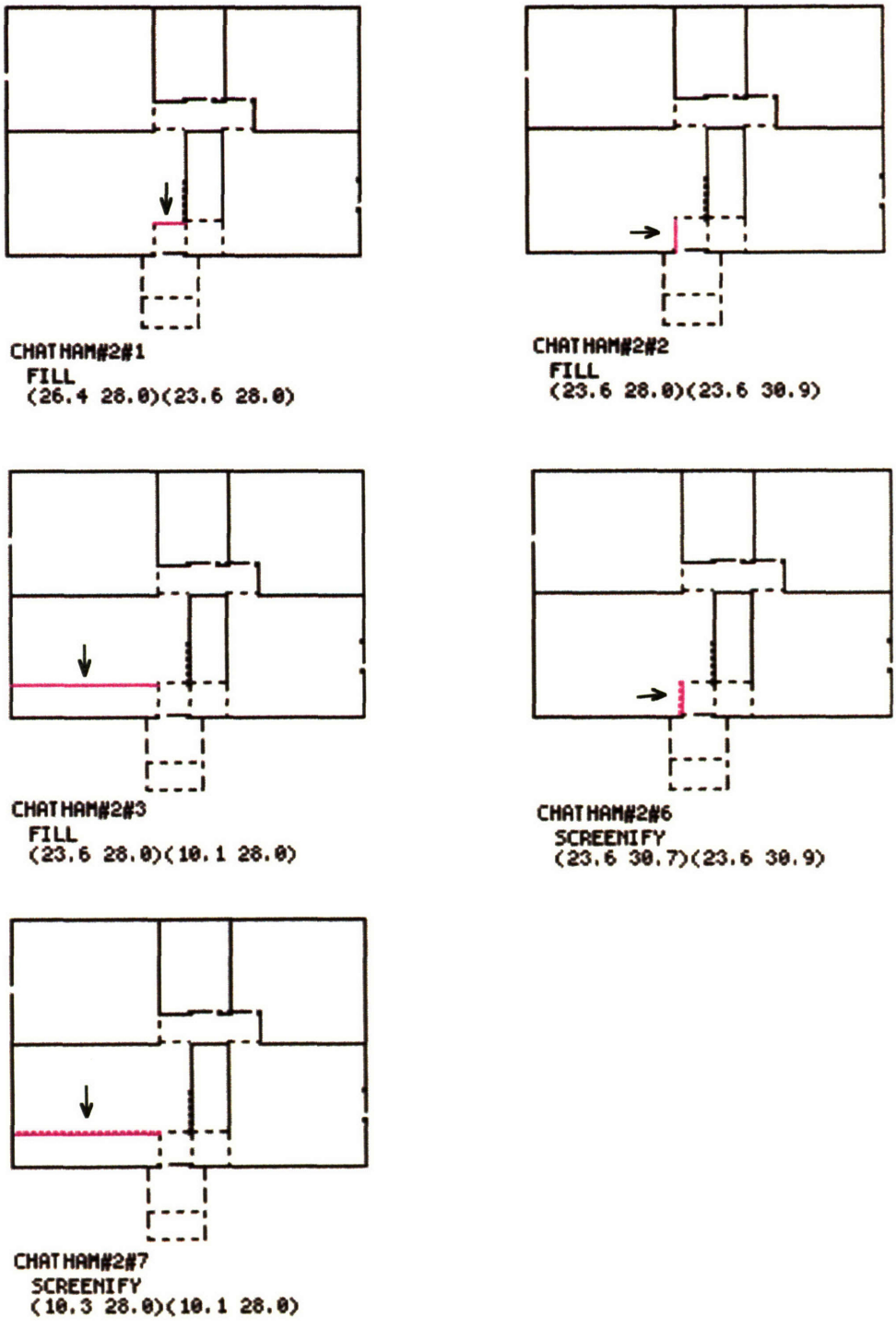



**Figure 8.12:** Designs with visual openness repaired to be less than 0.8. Arrows show locations of new walls or screens.

TAC checks the straightness of the path from front door to Living territory in these designs and finds the path now crooked enough in four of them (#2, #3, #6, #7). TAC has encountered a synergy: each modification to increase visual openness also has increased the path change in direction. In design #1, however, the path is unchanged; its change in direction is not large enough. TAC checks its knowledge base and finds that it knows how to increase a path's change in direction.[6] It suggests this change, then creates the design shown in Figure 8.13. This new design is the same as a previous design (Chatham#2#2), so it is discarded.

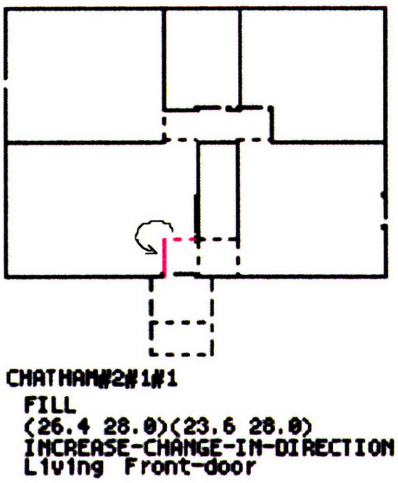



**Figure 8.13:** Design derived from #1, path change in direction has been repaired. Arrow shows movement of previously added wall to new location.

Four of the designs shown in Figure 8.12—Chatham#2#2, Chatham#2#3, Chatham#2#6, Chatham#2#7—satisfy the two previously unsatisfied goals.

Step 4: Evaluating all goals

TAC has now created designs that it expects will satisfy all goals, but has to make sure that its modifications did not cause the previously satisfied goals to become unsatisfied. For each of the current four designs, TAC checks that the visual openness of Living from the front door is greater 0.3 and less than 0.8, and that the change in direction from front door to Living is greater than 10 degrees and less than 120 degrees.

Chatham#2#2: visual openness value (0.23) is less than 0.8, but not greater than 0.3; its path is ok (change in direction is 68.7 degrees); it saves this design for repair.

---

6. The change in direction modifier takes a path and sequentially moves path nodes, which correspond to open edges, to other edges in the same territory as the node. When the change in direction is as desired, it switches the original edge and the new open edge, as in this example. If the desired change in direction cannot be achieved, it returns the starting design. This modifier operates a bit differently from other TAC modifiers: it does not operate on a single design element but rather on a set of design elements which it computes from paths between two design objects. Because of the complexity of this operator, we chose to make it opaque. This opacity, however, means that TAC is not able to reason about conflict or synergy between this modifier and others.

Chatham#2#3: visual openness value (0.12) is less than 0.8, but not greater than 0.3; its path is ok (change in direction is 68.7 degrees); it saves this design for repair.

Chatham#2#6: all goals are ok; this design is a solution.

Chatham#2#7: all goals are ok; this design is a solution.

TAC thus finds that adding screens at the specified locations has produced solutions. Adding walls has decreased the visual openness too much so it will attempt repair.

Step 6: Repairing designs

TAC has two designs to repair. For the first one, Chatham#2#2, TAC checks its knowledge base for ways to increase visual openness and proposes:

```
(or (remove blocking-elements-btw Living and Front-door)
    (puncture blocking-elements-btw Living and Front-door)
    (screenify blocking-elements-btw Living and Front-door)
    (rotate blocking-elements-btw Living and Front-door
          through angles-that-minimize-projection-btw)
    (move blocking-elements-btw Living and Front-door
          to any exterior-edges-for each element))
```

Examining Chatham#2#2, TAC locates the wall it added in the previous step to block the view to the Living territory from the front door. Applying the above suggestions to the wall, it proposes:

```
(or (remove Wall-17200)
    (puncture Wall-17200)
    (screenify Wall-17200))
```

Notice that it did not propose rotating the wall or moving it to an exterior edge; it knows that those modifications are not appropriate for walls.

Now examining each suggestion before carrying it out, TAC recognizes that removing the wall conflicts with having just added it in the previous step, so it prunes that suggestion.[7] It then creates new designs for the remaining two suggestions. (See Figure 8.14.)

7. A simple lookbehind mechanism checked for a suggestion in conflict with an immediately preceding suggestion.

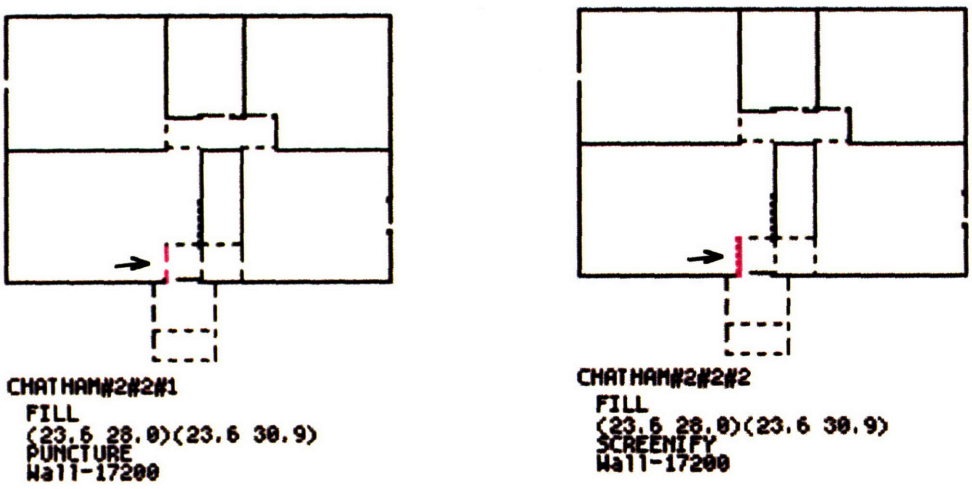



**Figure 8.14:** New designs for Chatham#2#2 after repairing for visual openness > 0.3.

TAC now checks whether the modifications had the intended effect, i.e. checks that the visual openness of Living from front door is greater than 0.3. It finds that puncturing the wall (adding a floor-to-ceiling opening 1.5 feet wide) has increased the visual openness sufficiently; the value is now 0.69. Replacing the wall with a screen yields a duplicate of a previous design (Chatham#2#6) so this new design is discarded.

Continuing with its repair step, TAC now focuses on the second design in need of repair, Chatham#2#3. It again checks its knowledge base, proposing the same initial five suggestions stated in terms of blocking elements, then identifies the blocking element as the wall it has added to decrease visual openness. It prunes irrelevant suggestions to rotate the wall or move it to an exterior edge, and proposes removing, puncturing, or screenifying the wall:

```
(or (remove Wall-17201)
    (puncture Wall-17201)
    (screenify Wall-17201))
```

Again, TAC notices a conflict with removing the wall it has just added, so it prunes the first suggestion, then creates the two new designs shown in Figure 8.15.

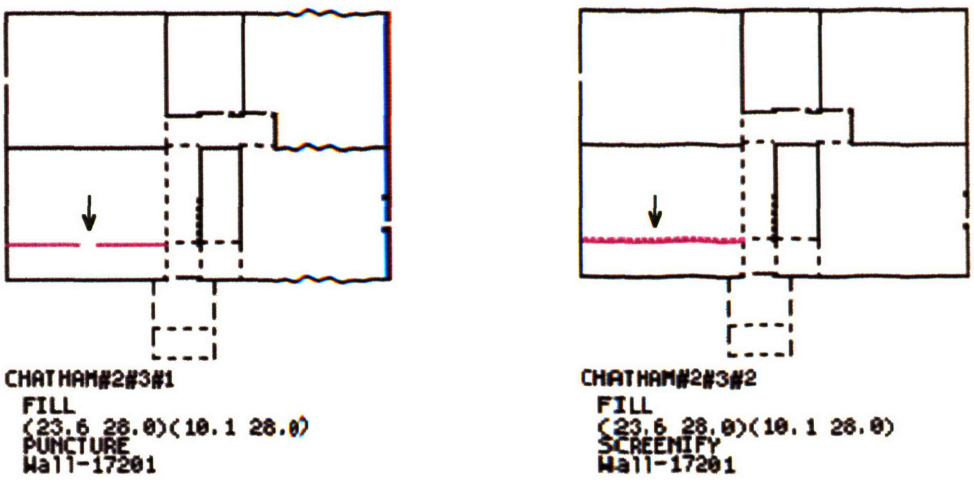



**Figure 8.15:** New designs for Chatham#2#3 after repairing for visual openness > 0.3.

Checking the effectiveness of the modifications, TAC finds that puncturing the wall does not increase the visual openness enough, so it discards this design. (The value is 0.19). Replacing the wall with a screen results in a duplicate of Chatham#2#7, so this design is discarded.

At this point TAC has completed its repair step, which produced one potential solution and three designs that were discarded—one whose intended goal was not met and two that were duplicates of previous designs.

Step7: Evaluating all goals

TAC finds that all goals are satisfied for the potential solution, Chatham#2#2#1 which has a punctured wall between the front door and the Living territory.

TAC returns the three solutions shown in Figure 8.16. The visual openness values for the three designs—Chatham#2#6, Chatham#2#7, and Chatham#2#2#1—are 0.76, 0.51, and 0.63, respectively. The path change in direction is 68.7 degrees for each design.

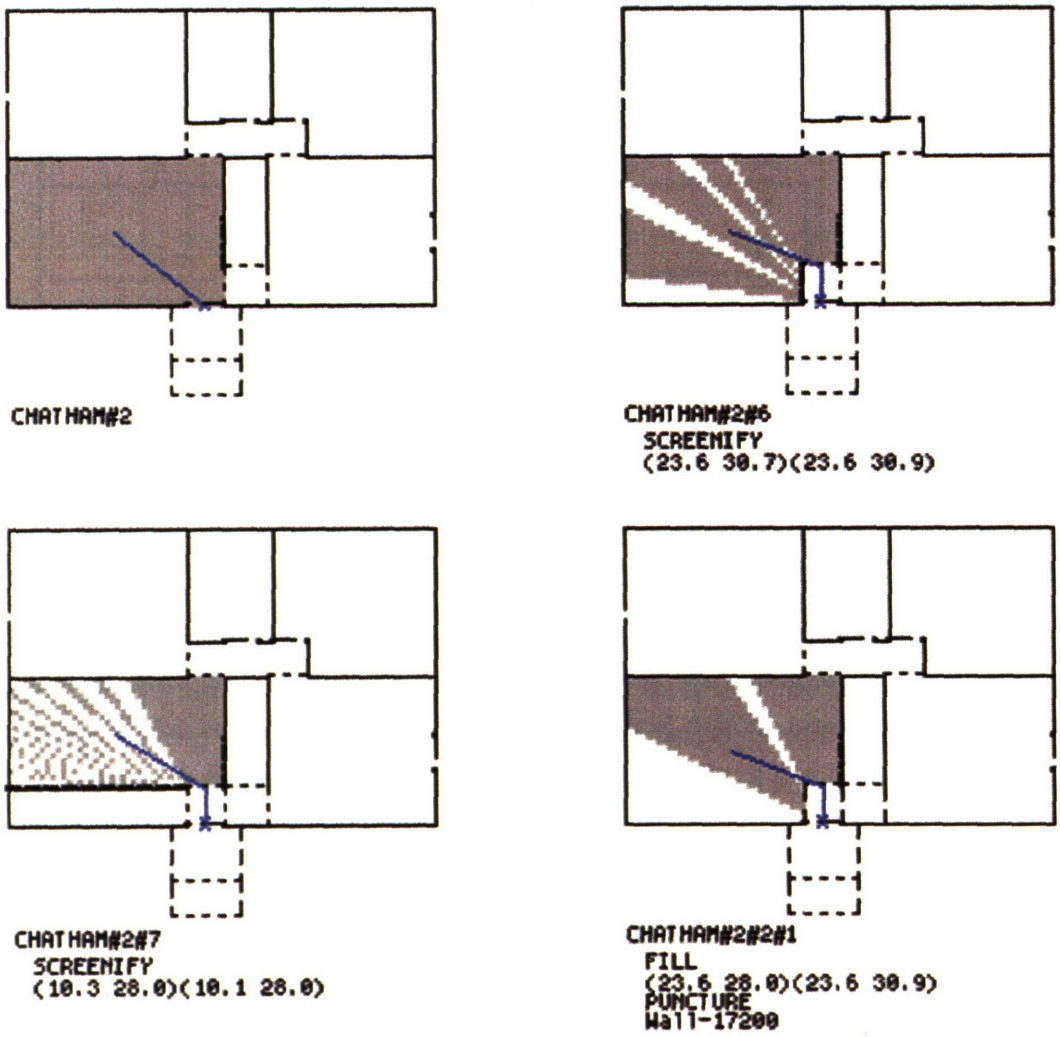



**Figure 8.16:** Starting design (#2), solutions for Living visual openness and path from front door: 0.3 < visual openness < 0.8, and 10 degrees < path direction change < 120 degrees. Shaded areas are visible; * is viewpoint; path is from front door to center of Living.

Notice that two of TAC's solutions seem better than the third: adding a screen the full width of the Living territory (in Chatham#2#7), thereby making the Living territory smaller, is probably not something an architect would propose. In order for TAC to judge the quality of solutions, it must have defined goodness criteria, e.g. leave the territories as large as possible.

A final note: Had we specified a single goal for the desired visual openness range, rather than two goals, the solution Chatham#2#2#1 would have been missed. The first modification step for this design, namely filling an open edge with a wall, resulted in a visual openness value lower than desired, and the design (Chatham#2#2) would have been discarded. With upper and lower bounds

for visual openness specified separately, the design satisfied the upper bound requirement and was not discarded. A subsequent modification (a puncture) repaired it to also satisfy the lower bound.[8]

Chatham#10

We now describe TAC's behavior with the same design problem—visual openness and physical accessibility of the Living territory—when given a very different solution to the first design problem. Chatham#10 has the side door as the perceived main entry; the front door has been replaced by a wall section, and the front porch, steps, and paths have been removed.

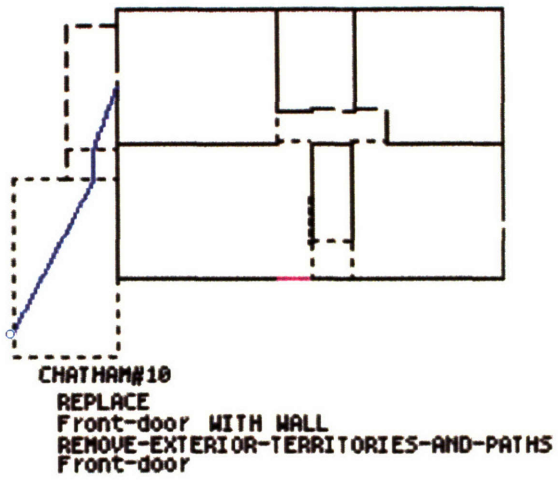



**Figure 8.17:** Starting design with side door as perceived main entry.
Path is from usual approach (o) to side door.

Step 1: Evaluating goals

Recall that we'd like the following goals to be satisfied:

```
<goal:(gt (visual-openness Living from (perceived-main-entry Chatham) 0.3)
      true>
<goal:(lt (visual-openness Living from (perceived-main-entry Chatham) 0.8)
      true>
<goal:(gt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          10) true>
<goal:(lt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          120) true>
```

TAC finds that two of the goals are not satisfied for Chatham#10. As shown in Figure 8.18, the visual openness of the Living territory from the side door is not greater than 0.3, and the path change in direction is not less than 120 degrees.

8. Had the knowledge base included an additional means for decreasing visual openness, namely an operator that was a composite of filling a wall and puncturing it, the solution would have been found in a single modification step. We chose to keep the operator set small and rely on the repair mechanism.

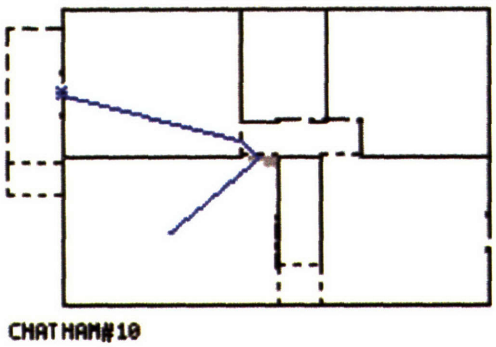



**Figure 8.18:** Chatham#10 visual openness and accessibility of Living from side door: Shaded area is visible; * is a viewpoint; path is from side door to center of Living. Visual openness value is 0.08; path change in direction is 126.8 degrees.

Step 2: Proposing and performing suggestions for visual openness greater than 0.3

TAC checks its knowledge base for ways to increase visual openness, and, as in previous examples, finds that it should decrease the opacity of design elements that block the view. It knows to decrease opacity by removing elements, puncturing them, replacing them with screens, rotating them, or moving them to exterior edges.

Examining the design, TAC finds that three wall sections block the view between the Living territory and the side door. Again noticing that it does not rotate walls or move them to exterior edges, TAC proposes removing, puncturing, or screenifying each of the walls:[9]

```
(or (remove Wall-1k1)
    (remove Wall-1k2)
    (remove Wall-1k3)
    (puncture Wall-1k1)
    (puncture Wall-1k2)
    (puncture Wall-1k3)
    (screenify Wall-1k1)
    (screenify Wall-1k2)
    (screenify Wall-1k3))
```

TAC then creates the nine new designs shown in Figures 8.19 and 8.20.

9. Three walls are represented rather than one because of the presence of a chimney that vents a basement furnace to the outside through the roof. It is represented by the middle wall section.

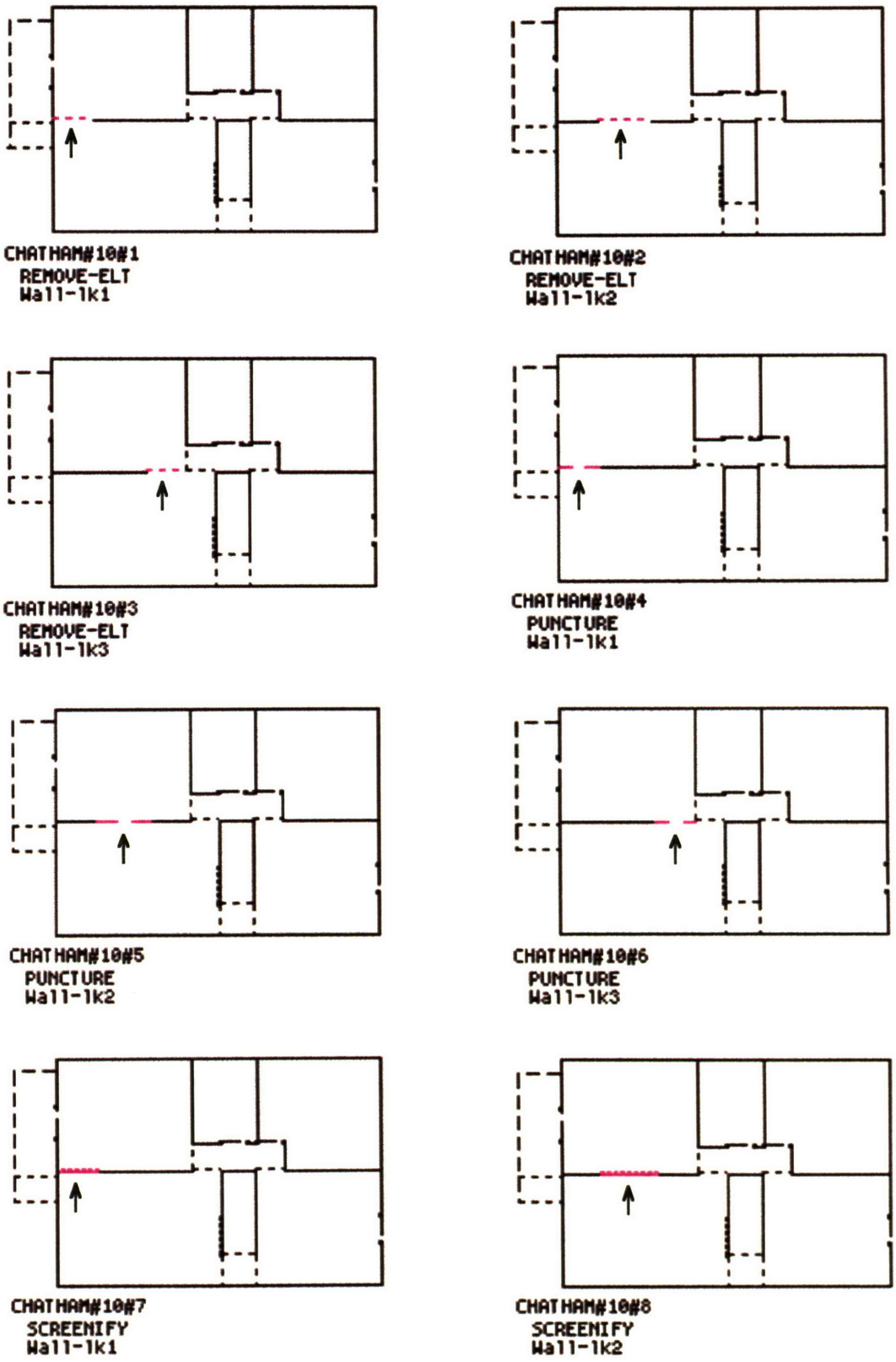



**Figure 8.19:** New designs with visual openness increased between Living and side door.

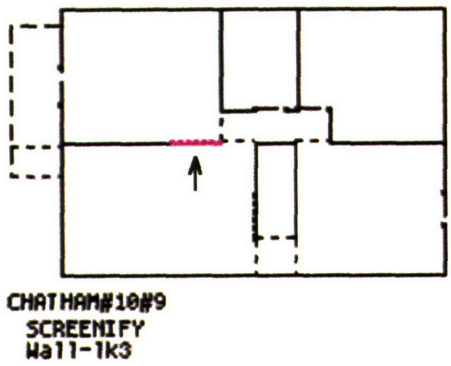



**Figure 8.20:** Another design with visual openness increased between Living and side door.

Checking the effectiveness of the modifications, TAC finds that designs #3, #4, #5, #6, and #9 do not increase the visual openness enough as intended, so it discards those designs. It then is left with the four designs shown below.

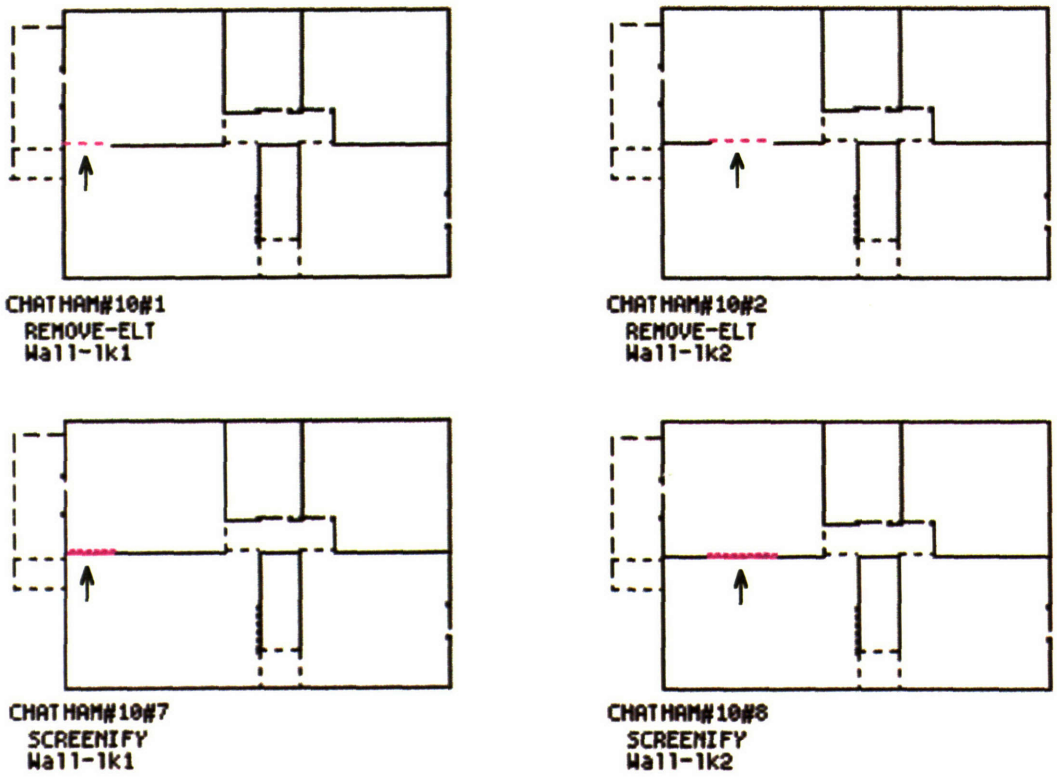



**Figure 8.21:** New designs left after pruning those whose intended goal was not met.

Step 3: Proposing and carrying out suggestions for path change in direction to be less than 120 degrees

TAC proposes suggestions for each of its current designs for the second unsatisfied goal, namely having a less crooked path between the Living territory and side door (change in direction of less than 120 degrees). It finds that designs #1 and #2 are ok: removing wall sections has increased visual openness and decreased the path change in direction as well—visitors can now look through and walk through the new opening to the Living territory. TAC finds that designs #7 and #8 are not ok; the path from the side door to the Living territory has remained unchanged and its change in direction is still too large. For each of these designs, it proposes to decrease the change in direction and creates the new designs shown below.

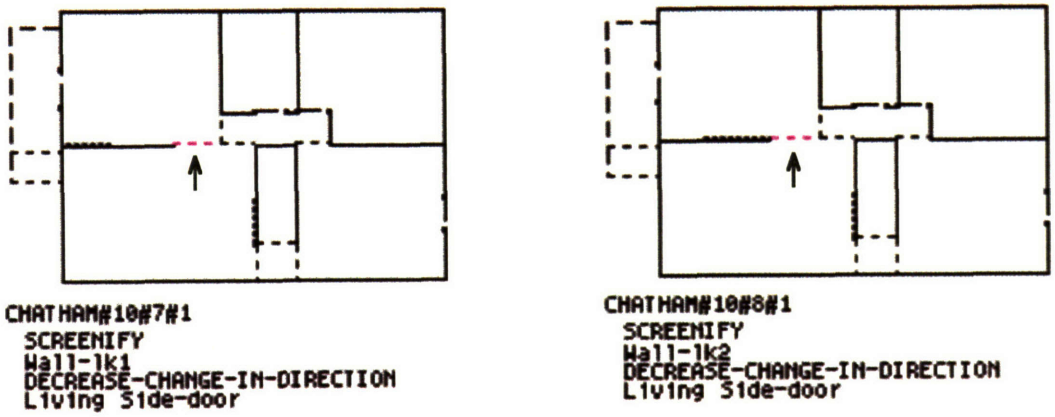



**Figure 8.22:** Decreasing change in direction between Living territory and side door. Arrows show newly opened wall sections.

Checking the intended goal, TAC finds that the change in direction from side door to Living has been sufficiently decreased in both new designs.

Step 4: Evaluating all goals

TAC finds all goals satisfied for each if its current designs:

Chatham#10#1: visual openness (0.60) is ok; path is ok (direction change is 25.1 degrees).

Chatham#10#2: visual openness (0.47) is ok; path is ok (direction change is 40.5 degrees).

Chatham#10#7#1: visual openness (0.44) is ok; path is ok (direction change is 98.6 degrees).

Chatham#10#8#1: visual openness (0.38) is ok; path is ok (direction change is 98.6 degrees).

TAC returns these four solutions, which are shown in Figure 8.23.

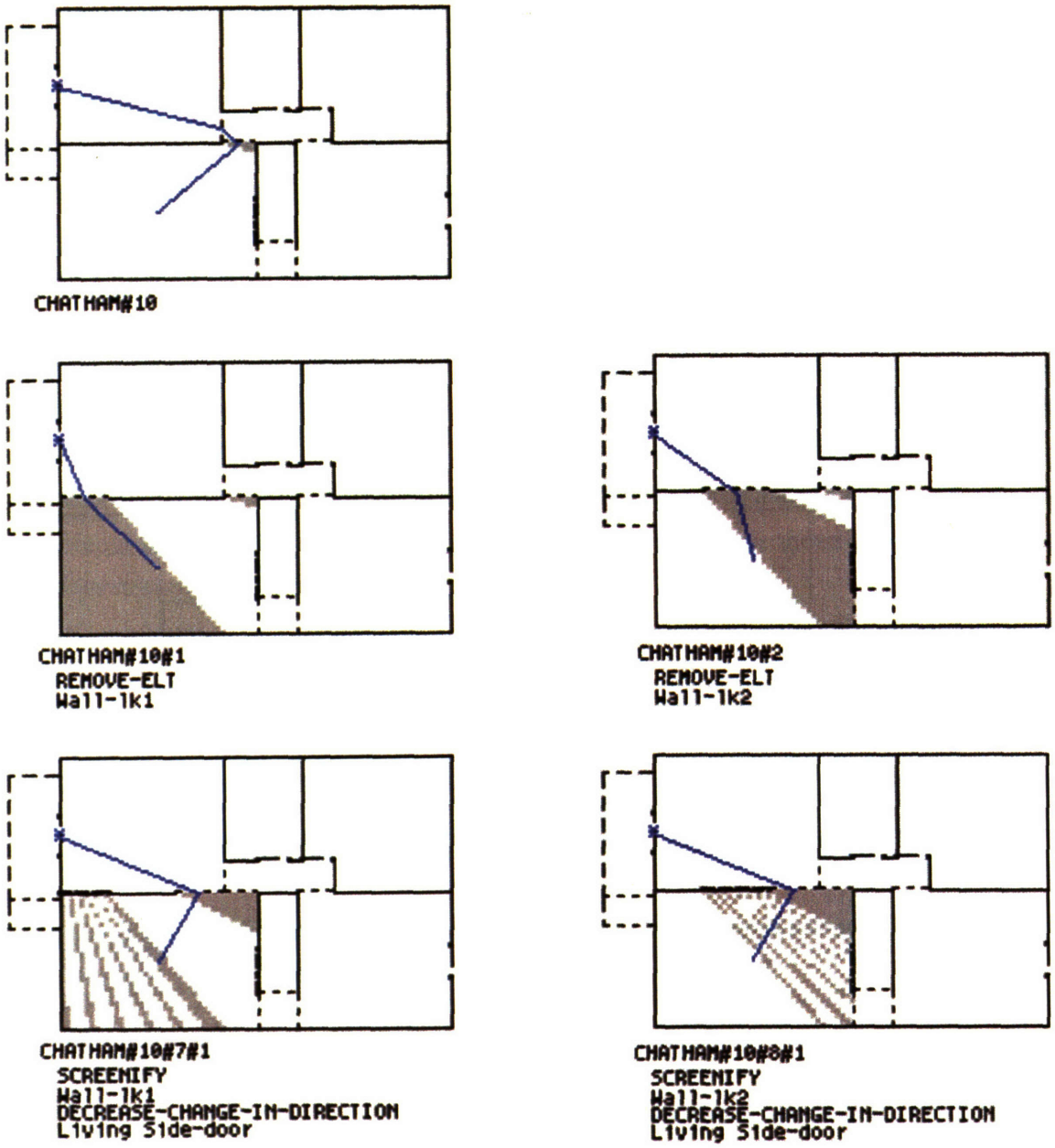



**Figure 8.23:** Starting design (top), solutions for Living visual openness and path from side door: 0.3 < visual openness < 0.8, and 10 degrees < path direction change < 120 degrees. Shaded areas are visible; * is viewpoint; path is from side door to center of Living.

Problem 3: Dining territory visually open from Living territory

```
<goal: (visually-open Dining from Living) true>.
```

Continuing with the Chatham design example, we now use a solution from the previous design problem, Chatham#10#1, to illustrate TAC's reasoning about the Dining territory's visual openness from the Living territory.

Step 1: Evaluating goals

TAC finds that the Dining territory is not visually open from the Living territory; its visual openness value is not greater than 0.6.

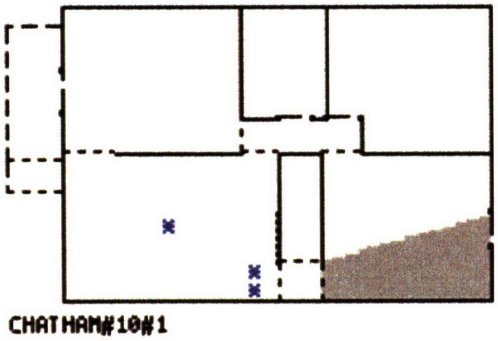



**Figure 8.24:** Visual openness of Dining territory from Living territory; value is 0.42. Shaded area is visible; * represents a viewpoint.

Step 2: Proposing and carrying out suggestions for visually open Dining

As we've seen in previous examples, TAC proposes increasing the value of visual openness by decreasing the opacity of intervening design elements. Checking its knowledge base, it proposes to accomplish this decrease in opacity by:

```
(or (remove blocking-elements-btw Dining and Living)
    (puncture blocking-elements-btw Dining and Living)
    (screenify blocking-elements-btw Dining and Living)
    (rotate blocking-elements-btw Dining and Living
          through angles-that-minimize-projection-btw)
    (move blocking-elements-btw Dining and Living
          to any exterior-edges-for each element))
```

TAC determines that the stair in this design blocks the view between Dining and Living. It finds that each of the above suggestions, except puncturing, is relevant for the stair, so it proposes:

```
(or (remove Stair)
    (screenify Stair)
    (rotate Stair 90)
    (rotate Stair 270)
    (move Stair to <edge: 42.11...>)
    (move Stair to <edge: 18.22...>)
    (move Stair to <edge: 10.11...>))
```

A note about rotating the stair: TAC computes the orientations for the stair that would minimize the projection of the stair in the direction of the sight line between territory centers. In these orientations, the stair would "cast the smallest possible shadow" so more of the Dining territory would be visible from the Living territory.

For the above suggestions, TAC creates the new designs shown in Figures 8.25 and 8.26. In each design, the Dining territory is visually open from the Living territory.

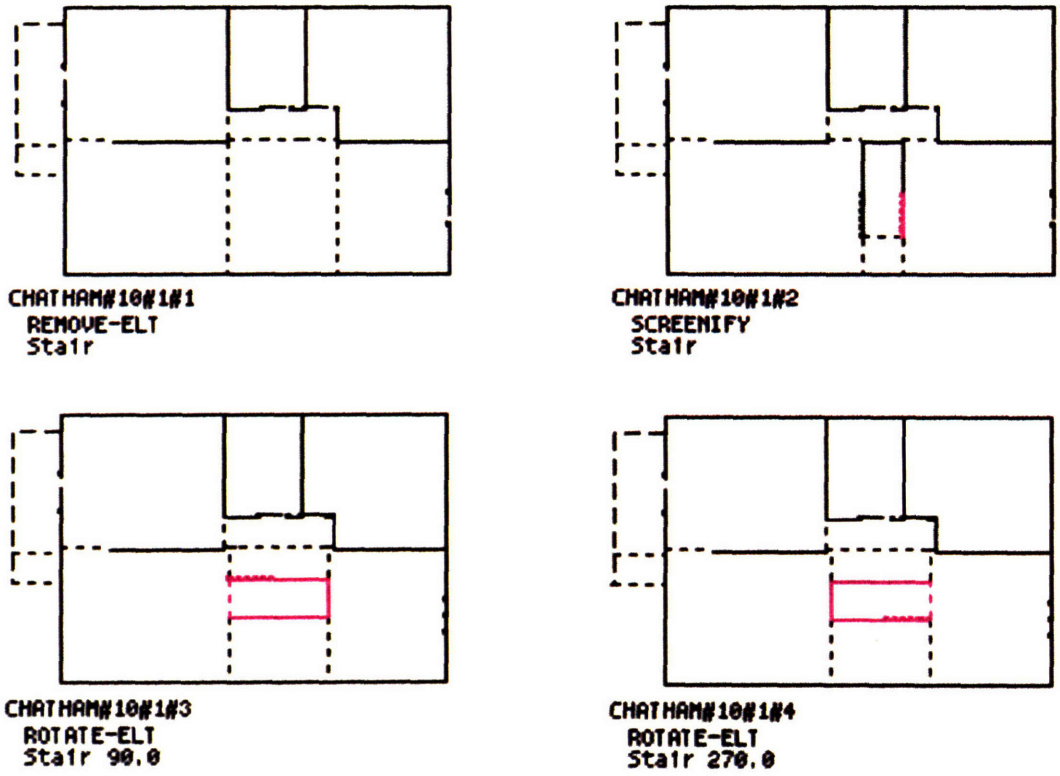



**Figure 8.25:** Designs that increase visual openness of Dining from Living.

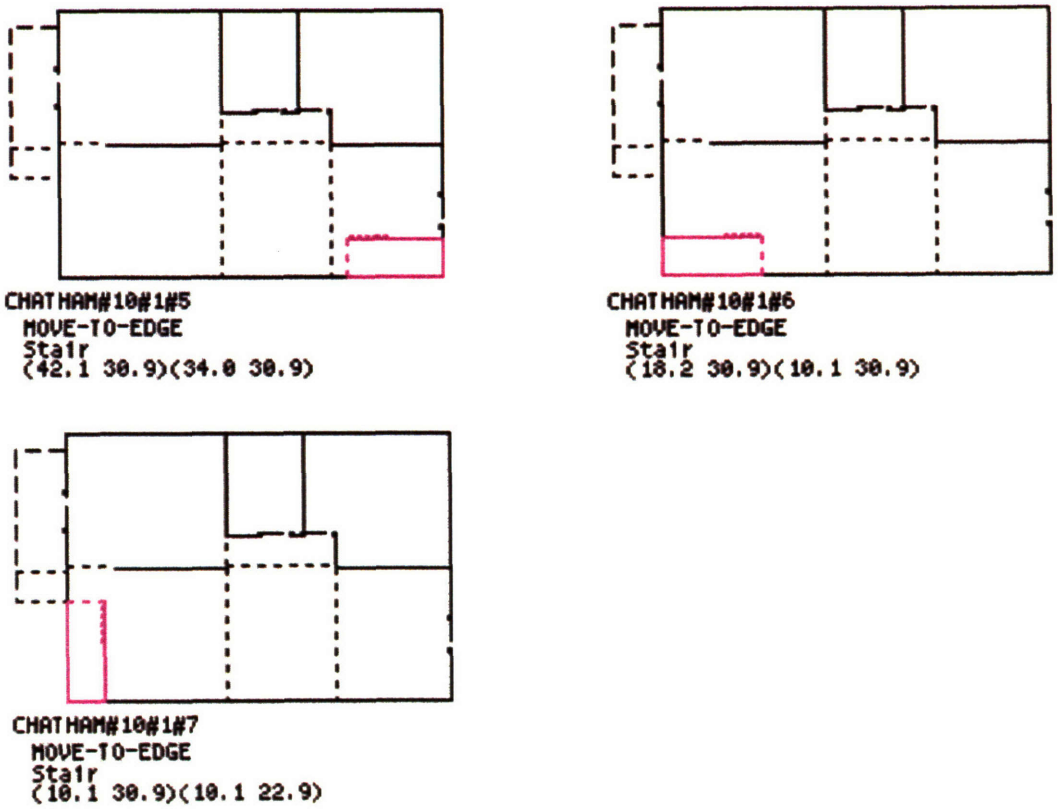



**Figure 8.26:** More designs that increase visual openness of Dining from Living.

Note that when moving the stair in this example, TAC has not taken into consideration other stair-related goals. In particular, we did not tell it that the design was to have a stair, so it proposed removing the stair. We discuss in Chapter 10 how TAC might specify (with the designer's approval) "common" goals related to a particular building type. In this way, TAC could decrease the number of goals a designer must explicitly specify.

TAC returns all seven new designs as solutions. TAC's solution Chatham#10#1#2, which corresponds to the solution chosen by the owners, is shown in the photograph in Figure 8.27.

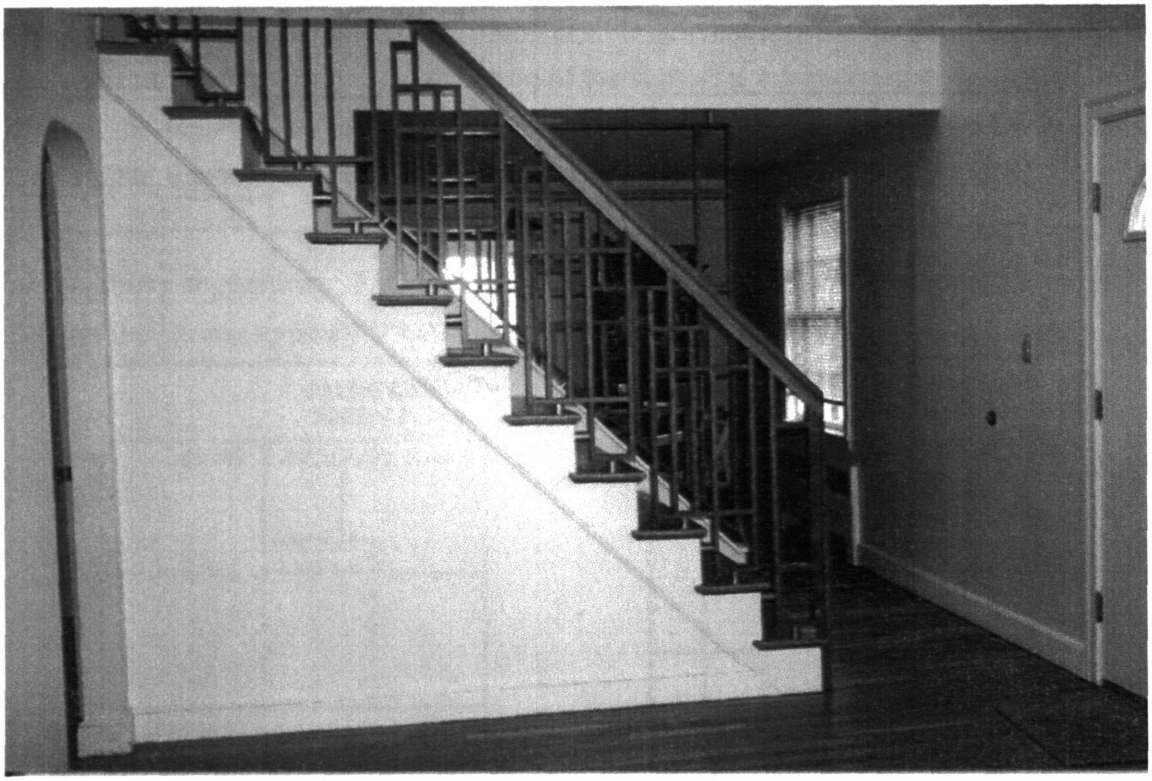


**Figure 8.27:** Screenifying the stair: a wall has been replaced with a screen in the Chatham house.

Problem 4: Kitchen adjacent to Dining

```
<goal: (use-adjacent kitchen dining) true>
```

This design problem is akin to a simple architectural space-planning problem: labels on territories change, but the physical form does not. We illustrate this design problem by describing how TAC accomplishes the goal above for a solution to the previous design problem, Chatham#10#4#1. TAC works with the use space model for this design.

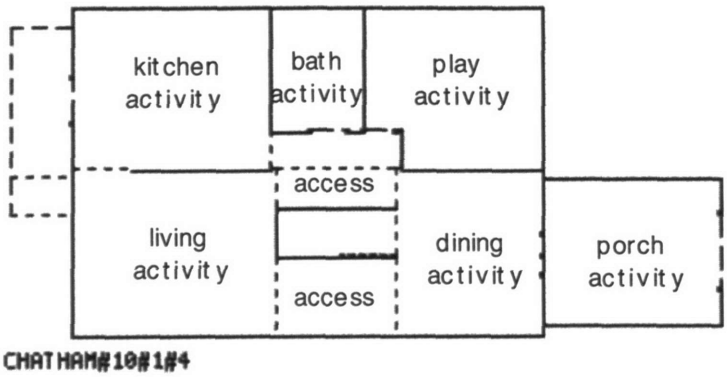



**Figure 8.28:** Use space model for Chatham#10#1#4: territories carry indication of intended use.

Step 1: Evaluating goals

Examining the use space model shown above, TAC finds that the kitchen activity is not adjacent to the dining activity.

Step 2: Proposing and carrying out suggestions for kitchen adjacent to dining

TAC checks its knowledge base and finds that it knows how to set the value of the design characteristic in the goal expression to true so it suggests:

```
(set-value of use-adjacent for kitchen and dining to true)
```

TAC then proposes moving the kitchen activity to another use space:

```
(exchange-use kitchen with any use-spaces-adjacent-to dining)
```

It then proposes moving the kitchen activity to particular use spaces:

```
(or (exchange-use kitchen with play)
    (exchange-use kitchen with porch))
```

Note that TAC does not propose moving the kitchen activity to either of the use spaces labeled "access" in Figure 8.28; the territories associated with these use spaces were deemed too small. It also does not propose building a new territory for the kitchen activity, even though this might satisfy the goal, because TAC's current operators do not change a design's footprint.

Carrying out its suggestions, TAC exchanges the kitchen activity with the play and porch activities and creates the following two designs, both of which are returned as solutions.

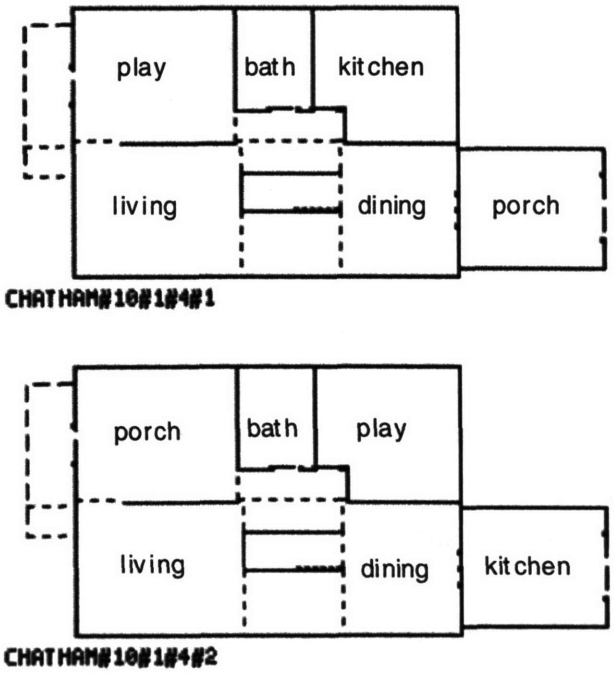



**Figure 8.29:** New designs with kitchen activity adjacent to dining activity.
Activity labels are abbreviated, e.g. play = play activity.

The designs shown in Figure 8.29 are solutions to all goals in the four design problems: they have one perceived main entry; the Living territory (which we'll assume coincides with the territory labeled with the living activity) is visually semi-open and physically semi-accessible from the perceived main entry, as defined by our visual openness and path change in direction goals; the Dining territory is visually open from the Living territory; and the kitchen activity is adjacent to the dining activity.

## The Architects' Designs

Several of TAC's solutions are very similar to the architects' two designs. Given that one of the architects contributed to TAC's knowledge of architecture, the similarities are not that surprising. The similarities do show, however, that TAC knows something of architecture, namely, how to modify a design so that it exhibits particular experiential qualities. As shown in the figures at the end of this section, TAC's designs have features in common with the architects' designs. TAC created one perceived main entry by removing the front porch, steps and walk, turning the current side door into a new front door. It increased visual openness between the Living territory and the new front door by removing a section of wall, a modification that it finds also makes the Living territory more easily accessible. It turned the stair to increase visual openness between the Dining and Living territories. It exchanged the playroom and kitchen activities so that the kitchen activity would be adjacent to the dining activity.

The similar designs show some differences, however. Many of the differences result from the architects' working with a larger goal set than TAC. These goals were not given to TAC because some of them would not have illustrated new TAC behavior, e.g. making the Kitchen territory more visually open from the Dining territory. Other goals were outside the scope of TAC's current operators, which do not change a design's footprint, e.g. enlarging the entry porch. Other differences between TAC's designs and the architects' are due to both unspecified goals and lack of information in TAC's knowledge base. When the Dining territory became smaller as a result of turning the stair, for example, TAC did not enlarge the territory at the expense of the Porch as the architects did. (See Figure 8.30.) Desired sizes for the territories were not specified, but there was an implicit assumption made by the architects and the clients that the Dining territory would not be smaller. Even if the sizes had been specified, however, TAC currently does not know that a territory can be made larger by making a neighboring territory smaller.

TAC came up with designs that differed significantly from the architects' designs. Most of these designs again resulted from the architects' working with more goals, both implicit and explicit. TAC's Chatham#2#7 design with the screen the full width of the Living territory, for example, satisfied the specified goals, but violated an implicit goal of creating only useful-sized

territories. TAC also was not told, for example, that the clients preferred that the stair remain in a central location, so it suggested moving the stair to exterior walls, creating plausible designs but not what the client had in mind. TAC was not told that the architects and clients desired that the house be connected to a neighborhood nor given information about the neighborhood. As a result, TAC did not know that the side door makes a better perceived main entry because the street on that side of the house is less busy and the houses closer together. As a result, it did not prune its designs with the front door as perceived main entry.

Not all of TAC's different designs were due to shortcomings, however. Some of its designs are quite plausible and result from its ability as a computational tool to easily and quickly carry out transformations: it produces many variations on a theme, a task an architect would find very tedious. In order to increase visual openness between the Living territory and the new front door, for example, TAC created nine designs by performing each of three operations (remove, puncture, replace with screen) on each of three wall sections. In some cases, TAC's designs may be redundant or even bad (e.g. Chatham#2#7), but they can be easily set aside by the designer as he focuses on the designs that meet his specified and unspecified criteria. In either situation, TAC has utility as a brainstorming tool. In addition, TAC can help a designer and his client elucidate goals by calling attention to desired or undesired features. Creating a design with a Living territory that is too small, for example, may serve as a reminder to a client that he wants a particular size for the Living territory.

The following two figures show the architects' designs and similar designs that TAC created given the seven design goals discussed in this section.

In Figure 8.30, the designs share these features:

- front door, front porch, and front walk have been removed to turn the former side door into the new front door (i.e. perceived main entry)
- the wall between territories labeled "play" (formerly kitchen) and "living" has been opened up: the architects have removed two wall sections; TAC has removed a wall section in the middle design, and replaced a wall section with a screen and removed a second wall section in the bottom design.
- the stair has been turned
- the kitchen and playroom have been exchanged in the original design

In Figure 8.31, the designs share all of the above features except the turned stair.

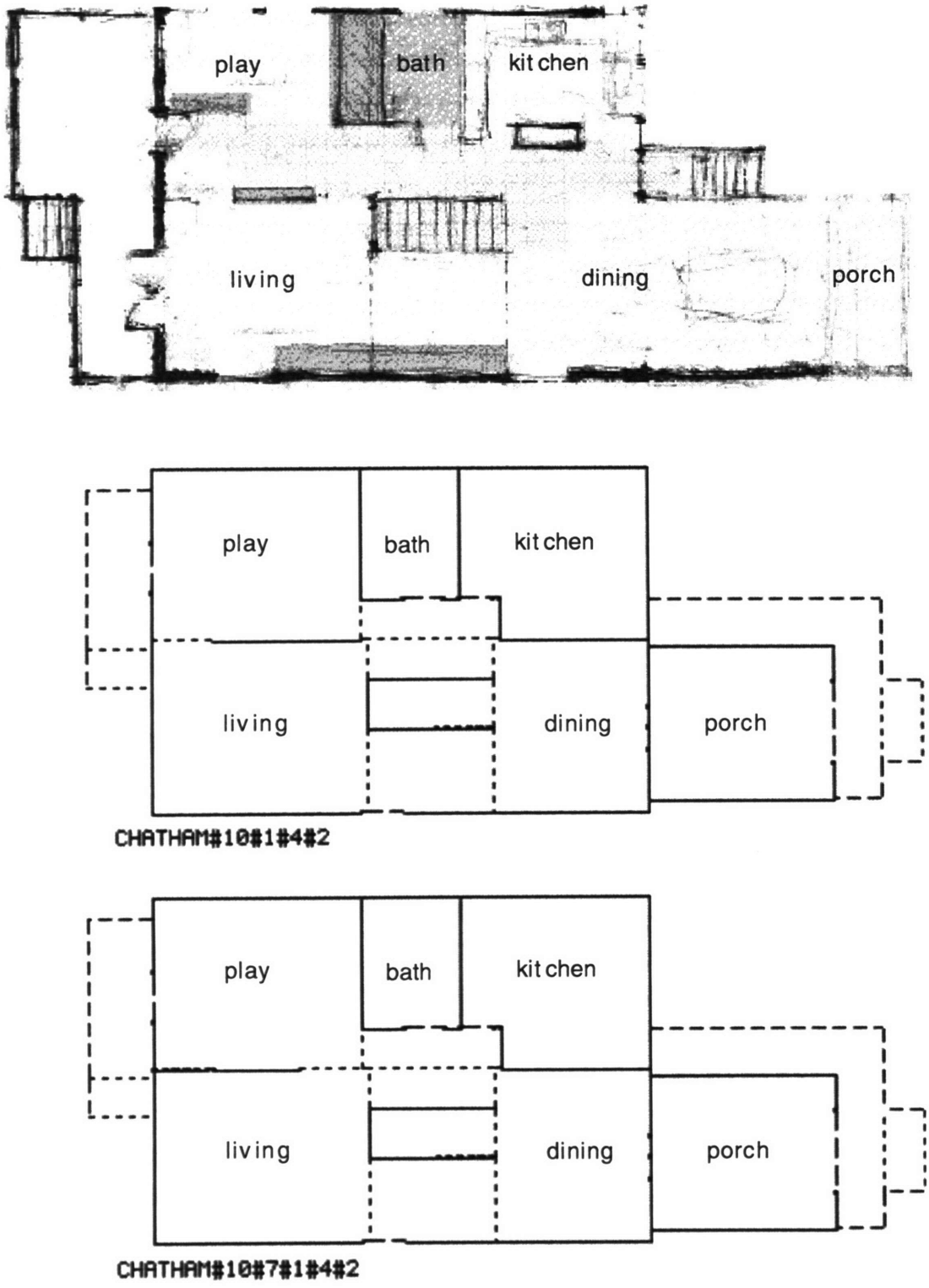



**Figure 8.30:** Architects' design (top) and two of TAC's designs.
The territory labels represent activities.

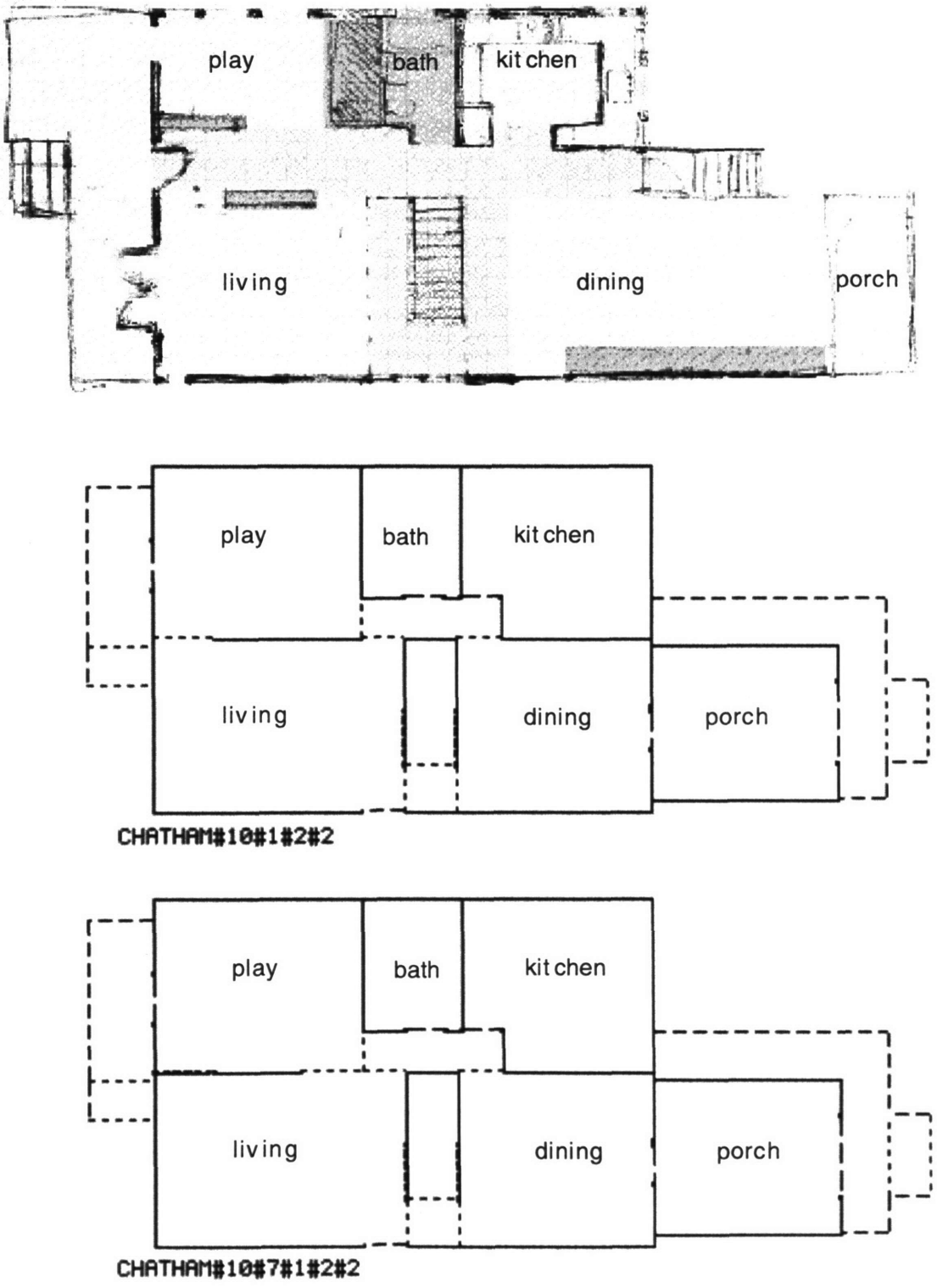



**Figure 8.31:** Architects' alternate design (top) and two of TAC's designs. The territory labels represent activities.

## Epilogue

For the Chatham house exercise, we gave TAC a design problem containing the seven goals presented in this section. In discussing TAC's behavior, we grouped the goals in the following way:

1. The design has one perceived main entry.

```
<goal: (one-perceived-main-entry Chatham) true>
```

2. The Living territory is visually semi-open and physically semi-accessible from the perceived main entry (i.e. partially visible and reached via a somewhat crooked path).

```
<goal:(gt (visual-openness Living from (perceived-main-entry Chatham) 0.3)
      true>
<goal:(lt (visual-openness Living from (perceived-main-entry Chatham) 0.8)
      true>
<goal:(gt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          10) true>
<goal:(lt (change-in-direction-btw Living and (perceived-main-entry Chatham)
          120) true>
```

3. The Dining territory is visually open from the Living territory.

```
<goal: (visually-open Dining from Living) true>.
```

4. The kitchen activity is adjacent to the dining activity.

```
<goal: (use-adjacent kitchen dining) true>.
```

When given all seven goals, TAC created 816 designs, 288 of which satisfied all the goals. The large number of solutions brings to light an important issue: How can TAC display 288 solutions without overwhelming the user? Developing techniques and strategies for presenting large volumes of data to a user is an active area of research (e.g. Shneiderman, 1998). A key question for TAC: How can it group the solutions into manageable subsets using meaningful abstractions? The difficult issue here is how to define "meaningful".

Alternatively, what was useful expositionally may be useful computationally as well as a way of not overwhelming the user: group goals into separate design problems and let TAC work on the problems sequentially, creating designs for a problem and selecting a subset of those solutions as starting points for the next problem. This technique works well when the design problems are independent, as with Chatham, but it may be difficult to know ahead of time whether problems are independent. In addition, it introduces the question of how to select the subsets of designs for subsequent design problems. In particular: How is a subset selected to make sure it contains "interesting" designs, by which we might mean designs that meet many criteria and show wide variation?

As an experiment, we grouped the goals into the four design problems described earlier (and listed above). We gave TAC the design problems and had it work on them sequentially. It chose the four "best" designs for a problem as starting points for the next problem. To determine "best", it employed a simple ranking scheme for the design characteristics in the goals, using a total order when possible, otherwise a partial order. With this scheme, the number of designs was reduced significantly, mainly because only four of the first design problem's fourteen solutions were passed along to the second design problem. TAC created 224 designs, 128 of which were solutions. Starting with a single design for the second design problem, rather than four, reduced the number of designs to 56, 32 of which were solutions.

The four designs that TAC chose as starting designs for each design problem, however, were not always the designs that a user would have chosen. Designs are often judged using multiple criteria, only some of which may have been used when creating the design. TAC's moving the stair to an exterior edge, for example, has an adverse effect on circulation patterns: access paths to the stair cross through the middle of the Living and Dining territories, thereby compromising the usefulness and privacy of the territories. One solution would be to explicitly state all evaluation criteria, but given the opportunistic nature of design and the ever-changing nature of design goals, it would be very difficult to explicitly state all evaluation criteria. What's more, the evaluation criteria are incommensurate, and it is not obvious how to define a single, unambiguous evaluation function based on them.

It seems best, then, to either let TAC create a large number of solutions and deal with how to present them to the user (e.g. by means of a "design solution navigator" of some sort), or have it create designs sequentially for separate problems and let the user select designs of interest for subsequent problems.

## 8.2 Analysis: Prairie Houses

Thus far TAC has been discussed as a design tool: it evaluates a design with respect to design goals, suggests repairs, and creates new designs. It also can be an analysis tool: its evaluation component can be used without invoking the repair suggestion or design creation mechanisms.

Frank Lloyd Wright's Prairie houses were the subject of an experiment designed to investigate TAC's utility in analyzing designs and definitions of architectural type. TAC was given 21 characteristics of Prairie houses and 15 houses—six Prairie, six non-Prairie, and three Frank Lloyd Wright houses considered transitions between pre-Prairie and Prairie periods—and asked to evaluate the "Prairieness" of each of the houses by determining which of the characteristics were present.[10]

### Experiment

Frank Lloyd Wright was chosen for our experiment because he was prolific, has been well-studied, and is regarded as a master at manifesting experiential qualities in his buildings. His Prairie houses were chosen because they share many common features while also being quite varied, and because they have been extensively studied by architectural critics and historians (e.g. Manson, 1958; Brooks, 1972; Twombly, 1979; Hildebrand, 1991), and by researchers interested in computational systems for design (e.g. Koning and Eizenberg, 1981; Chan, 1992). Six representative Prairie houses were chosen, one from each of six Prairie house categories (Pinnell, 1990). The non-Prairie examples were chosen in order to minimize the differences that might be attributed to issues not germane to the experiment. The examples were limited to single-family stand-alone houses, to minimize differences due to building type; to approximately the same time period, to minimize differences due to societal changes, e.g. addition of a garage; to those about the same size, to minimize differences due to mismatch in number or sizes of spaces; to American designs, to minimize cultural influences. Examples also were chosen that were considered transitions between Wright's Prairie and pre-Prairie periods in order to see whether the transition nature of the designs would be reflected in the evaluation.

We specified fifteen characteristics representing details of physical form and six characteristics representing experiential qualities. The characteristics are stated in terms of the living spaces and main living space of a design.[11] By "living spaces" we mean the semi-private spaces in a house—spaces to which guests might be invited, but not casual visitors, e.g. living room, dining room, library.[12] By "main living space" we mean the space corresponding to what would typically

10. This experiment is a repeat of one run with a rule-based precursor to TAC. That system and experiment, as well as a discussion of the shortcomings of representing TAC's knowledge using rules, are described in (Koile, 1997).

11. The terms "space" and "use space" will be used interchangeably. Recall that a use space is a territory paired with its intended use, e.g. the main living space is represented as a particular territory and a "main-living" activity label.

12. We do not include the kitchen as a living space because at the time the Prairie houses were built in the early 20th century, the kitchen typically was considered a work space only.

be called a living room.

Below are the two lists of characteristics we gave TAC.[13] The experiential qualities are manifested by the details of physical form whose indexes into the second list are shown to the right. (See Appendix G for representation details.)

Experiential qualities:

a. The design exhibits Wrightian group togetherness. (and 1 2 3)
b. The design exhibits home/hearth symbolism. (or 4 6)
c. The main living space is private. (and 7 (or 8 9 10 11))
d. The main living space is a place of refuge. (or 8 9 10 11)
e. The main living space is a place of prospect. (or 12 13)
f. An exterior space contiguous with the main living space is private. (and 14 15)

Details of physical form:

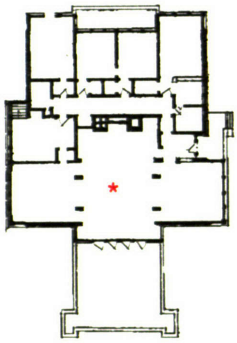
1. The design has a main living space that is the largest living space.
2. The design has a main living space containing a region from which all other living spaces are visible.
3. The design has a main living space connected to all other living spaces. (Two spaces are connected if they are no more than one space apart or if they have axially aligned doorways.)
4. The design has one fireplace location.
5. The design has a fireplace on an interior wall.
6. The design has a fireplace in the main living space.
7. The path from the front door to the private area does not pass within five feet of the center of the main living space.
8. The front door does not open into the main living space.
9. The front door and the main living space are on different levels.
10. The path from the front door to the main living space contains at least two changes in direction of greater than 15 degrees.
11. The path from the usual approach point to the main living space contains at least two changes in direction of greater than 15 degrees.
12. The main living space is elevated above the terrain.
13. An exterior living space at least 40% of the size of the main living space is contiguous with the main living space.
14. The front door does not open into the exterior space contiguous with the main living space.
15. The path from the usual approach point to the front door does not cross the exterior space contiguous with the main living space.

---

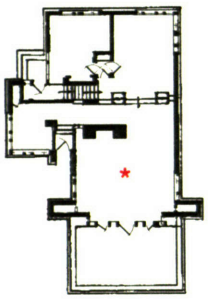
13. The characteristics, shown in English to improve readability, are derived in part from (Hildebrand, 1991). Hildebrand explains prospect and refuge as Appleton (1975) defined them: "By prospect Appleton means a condition in which one can see over a considerable distance, and by refuge he means a place where one can hide; in combination they reinforce one another, creating the ability to see without being seen." Hildebrand argues convincingly that the juxtaposition of prospect and refuge conditions in Wright's houses contribute to their "uniquely widespread devotion" especially in light of the houses' sometimes prominent faults (e.g. leaking roofs). He notes in his conclusions that "Wright had an intuitive but uniquely firm grasp of the shaping of habitation as an interweaving of these characteristics...."

Finally, we gave TAC models for the floorplans show in Figures 8.32 and 8.33 and asked it which of the above 21 characteristics were present in each floorplan.

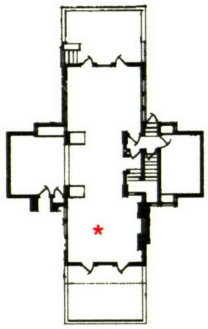
Prairie Houses: Cheney, Gale, Horner, Tomek, Willits, Roberts (Storrer, 1993)



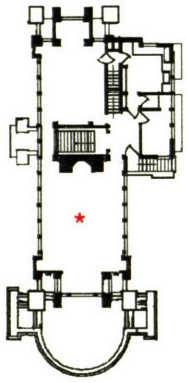
Cheney: 6, 13



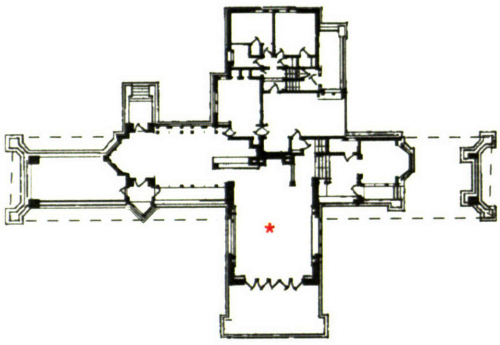
Gale: 6, 15



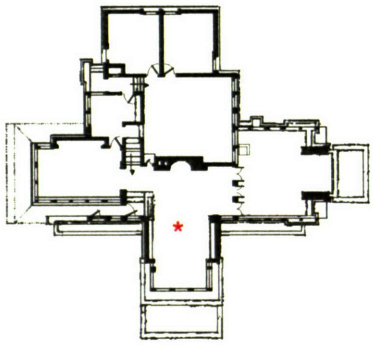
Horner: 6, 14



Tomek: 6, 15



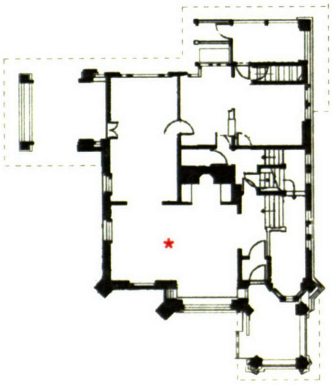
Willits: 5,12



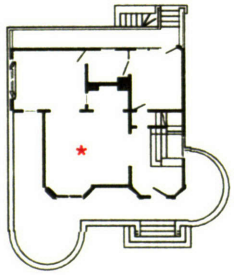
Roberts: 6, 15

Transition Houses: Emmond, Furbeck, Wright (Storrer, 1993)

Emmond: 5, 13



Furbeck: 4, 7



Wright: 4, 7

**Figure 8.32:** Prairie and Transition House data sets; * indicates main living space. First number following house name is count of experiential qualities exhibited (out of 6); second is count of physical form details exhibited (out of 15).

Non-Prairie Houses: Colvin, by George Maher (1916); Jones 5A24 (Jones, 1987);
Lawson, by Bernard Maybeck (McCoy, 1975); Mallory, by Arthur Rich (Scully, 1971);
Stickley 91 (Stickley, 1982); Winslow, by Frank Lloyd Wright, (Storrer, 1993)

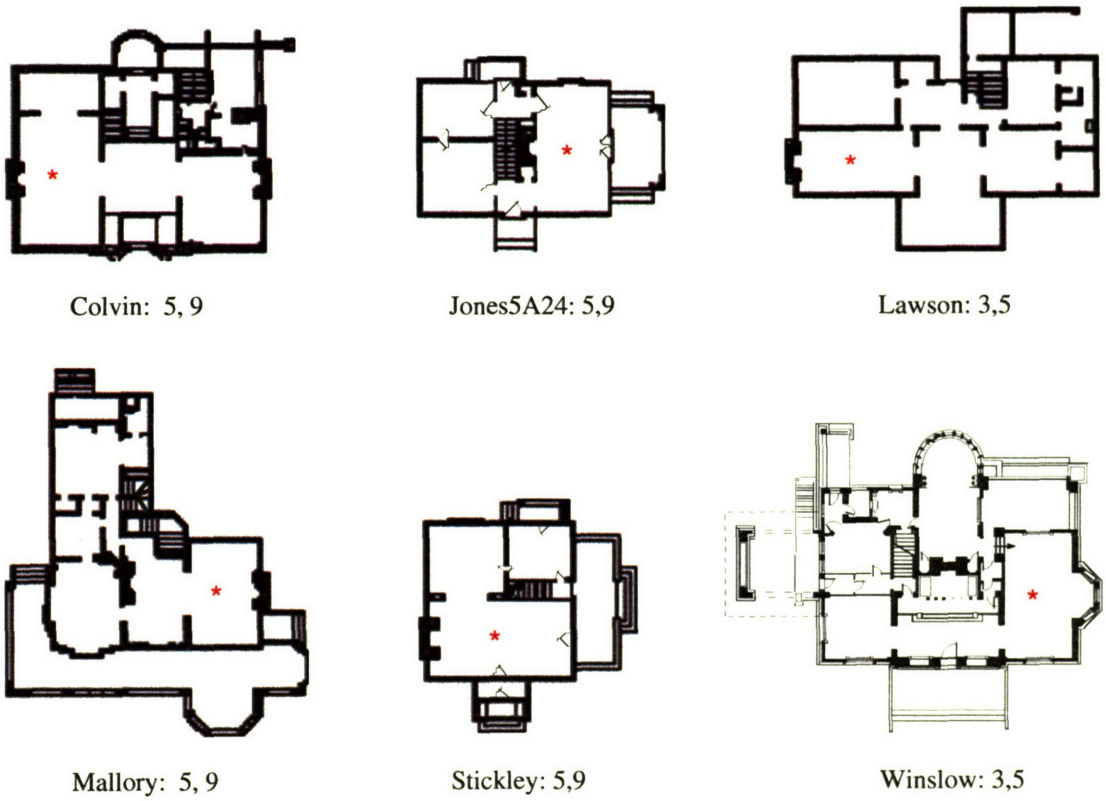



**Figure 8.33:** Non-Prairie House data set; * indicates main living space.
First number following house name is count of experiential qualities exhibited (out of 6); second is count of physical form details exhibited (out of 15).

## Results

The results for the Prairie houses were as expected: all but one exhibited all six experiential characteristics; half exhibited all fifteen physical form characteristics, with the rest exhibiting at least twelve.[14] The transition houses were not distinguishable from either the Prairie examples or the non-Prairie examples; their differences were not captured by the characteristics used in the experiment. The results for the non-Prairie houses were as expected for the physical form details: only one exhibited ten of the fifteen; the rest exhibited nine or fewer. The non-Prairie houses and transition houses, however, exhibited many of the experiential characteristics. This result is unsurpris-

14. See Appendix G for detailed results.

ing: Focusing on physical form characteristics yields a better measure of Prairieness, or perhaps any building type, since experiential characteristics may be manifested using a variety of methods, with particular methods favored by individual designers. In other words, different architects and their clients may want many of the same experiential qualities in their designs, but may prefer different methods for achieving those qualities. (See Hildebrand (1991) for an interesting discussion of this issue.)

The following charts summarize the experimental results.

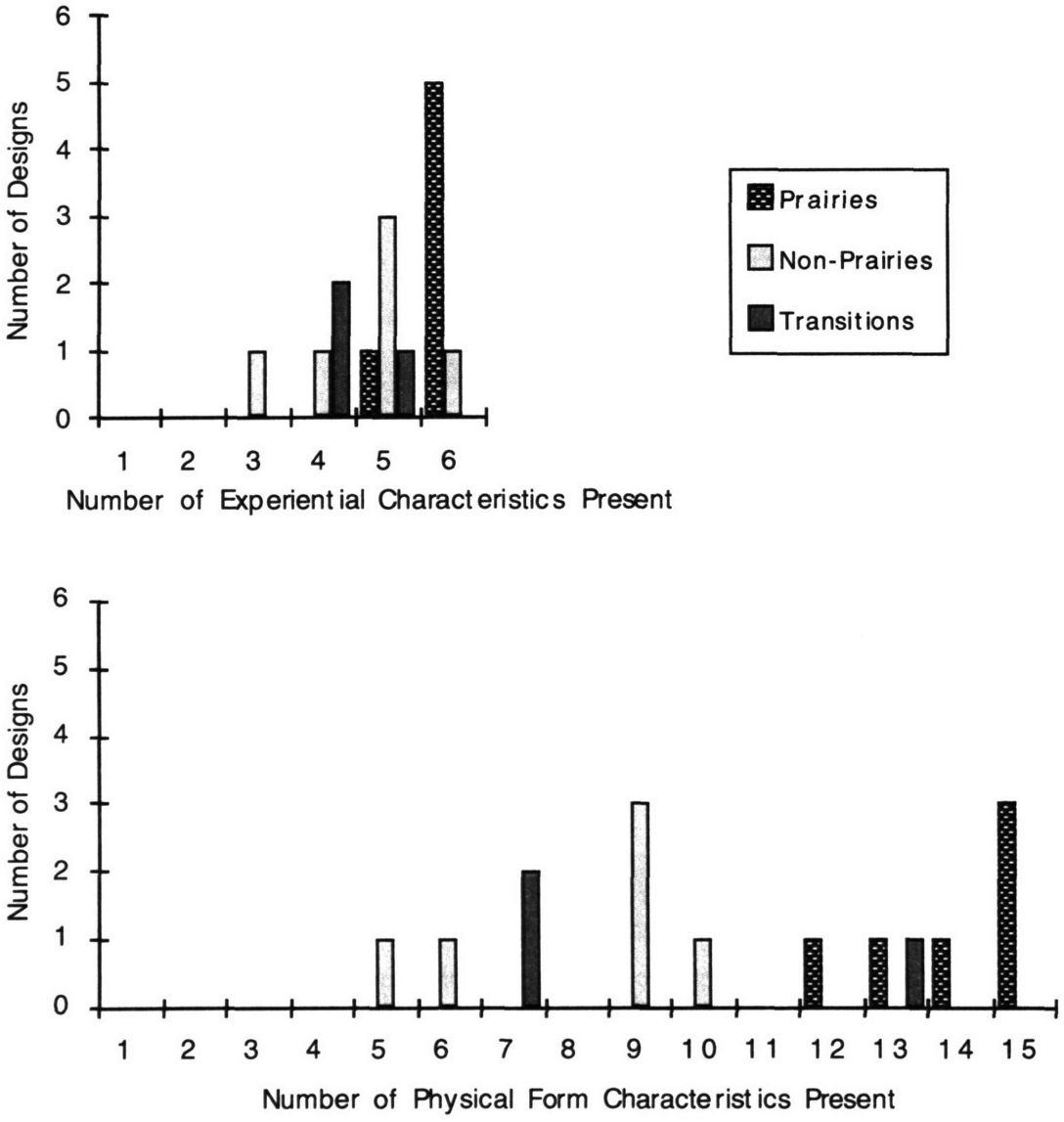



**Figure 8.34:** Experimental results for Prairie, Non-Prairie, and Transition houses.

## Conclusions

This exercise illustrates TAC's use in defining and analyzing characteristics of a particular building type, and in particular, one rich in experiential qualities. We lay no claim to a complete representation of the qualities in Wright's work, nor to a complete representation of his methods for manifesting those qualities. We do claim, however, that those qualities and methods can be represented and reasoned about in TAC. In the hands of an architect or architectural historian, TAC could be used to measure qualities such as visual openness, physical accessibility, privacy, even prospect and refuge, in very precise ways.

An interesting issue illustrated by this experiment is that buildings and building types can be represented by both experiential characteristics and physical form characteristics. The more usual comparison of buildings is via physical form. Houses are often thought to look Wrightian, for example, if they have hipped roofs and wide overhanging eaves. Interestingly, however, buildings may be more alike than initially thought when experiential characteristics are considered. Hildebrand (1991) presents the Scofield house by architect Wendell Lovett which looks nothing like a Wright house on the exterior, but which uses many of the same techniques as Wright to juxtapose prospect and refuge, or "cave and meadow" in Lovett's parlance. With a system such as TAC, the similarities between buildings such as Lovett's Scofield house and Wright's Prairie houses could be discovered easily.

# Chapter 9

# Related Work

This chapter focuses on two categories of work: systems that employ similar reasoning techniques and those that reason with similar experiential knowledge.

## 9.1 Reasoning Methodologies

We discuss several methodologies that share features with TAC's dependency-based redesign strategy: planning, case adaptation, and performance-based refinement.

### 9.1.1 Planning

#### Classical Planning

TAC is not a planning system, but it borrows ideas from classical AI planning (Wilkins, 1988; McDermott, 1995). The goal of a planning system is to find a sequence of actions that will get from a specified initial state to a specified goal state. States are generally represented as sets of propositional statements, operators map one state into another. A sequence of operators is a plan.

The sequence of repairs that gets TAC from an initial design to a solution can be thought of as a plan of sorts. TAC borrows the idea of limiting search for a solution by searching in plan space, repair suggestion space in TAC's case, rather than searching world states for the goal state. TAC first searches for likely repair operators, then searches for designs by applying those operators.

TAC also borrows the divide-and-conquer approach taken by planners. It reduces search by assuming that some goals can be solved independently: it satisfies one goal, then checks for global effects of the operator (Korf, 1987).

TAC differs from planning systems in significant ways, however. Its solution is a set of states, i.e. designs, not a plan, so it doesn't reason about the order of operators. Its divide-and-conquer strategy does not attempt to satisfy goals separately and merge partial solutions. Such a merge is far too difficult because it involves combining complex geometric arrangements of design elements. TAC's operators are not represented as preconditions and effects because operator effects cannot be enumerated. TAC uses the term "effect" in a more general sense to represent an intended goal rather than an immediate change to the world that results when an operator is carried out. If a wall is shortened, for example, the intended effect is increased visual openness,

not just a shorter wall. Planning systems in the domain of architecture have represented designs as sets of propositional statements and represented design modification operators as preconditions and effects (e.g. Coyne and Gero, 1995). Their authors suggest, as we do, that these representations are not powerful enough to capture the complexities of architectural design.

Most importantly, TAC differs from planning systems in that it is not given a goal state; it must search for it. TAC is given an initial design and a set of desired characteristics for that design. In some sense, that set of characteristics is a goal state, but not in a planner's sense. A planner's goal state is represented in the same way as its initial state, so a planner is able to reason directly from initial state to goal state via operators that transform one state into another, e.g. via a chain of one-step inferences. TAC's operators transform one design into another, but its specified goal state is in "desired characteristics" space. This mismatch between operators and goal state means that TAC must use a variation of test and generate to search for its solutions.

While design is not a traditional planning task, construction is: given an initial state (an empty lot or a building to be remodeled) determine a sequence of steps to transform initial state into goal state (the building that is the design solution). It's a complicated planning problem (Kartam, et. al., 1991), but since its goal state is known, it is a planning problem not a design problem.[1]

### Real-world Planning

TAC also borrows from "real-world" planning. Real-world planning systems extend classical planning to work in worlds with unexpected operator effects. These systems generate a plan, execute (or analyze) the plan, then either repair the plan or replan from the current state if necessary. Plan repair systems are generally concerned with a correct plan (e.g. Hammond, 1990; Kambhampati and Hendler, 1992; Beetz and McDermott, 1994). Replanning systems are generally concerned with arriving at a goal state (e.g. Ambros-Ingerson, 1988; Knoblock, 1995; Wilkins, 1988, 1990). TAC borrows ideas from both plan repair and replanning. Like plan repair systems, it uses an explanation of failure to guide repair in a manner inspired by dependency-directed search (Stallman and Sussman, 1977). Like replanning systems, it attempts to reach a goal state by starting from the current state rather than focusing on the sequence of steps from the initial state.

TAC differs from real-world planning systems in many of the same ways that it differs from classical systems, e.g. it does not reason directly from initial state to goal state. It differs from systems that do plan repair in another significant way: TAC only has one kind of operator, namely operators that transform a design into a new design. Systems that repair plans have two kinds of operators: those that transform one state into another (e.g. move a block to a new location) and those that repair a plan (e.g. add a step to a plan, or reorder steps). Because TAC is not concerned

1. Contingencies may require interleaving of planning and redesign of certain aspects of a building.

with the sequence of steps needed to arrive at a solution, it has no need of operators that repair the sequence.

## Related Planning Systems

Interesting comparisons can be made to two planning systems. One was developed to explore AI issues (Simmons, 1988), and one was developed to explore computer-aided architectural design issues (Colajanni, 1991).

### Gordius

Gordius (Simmons, 1988) is a planning system that employs a paradigm its author calls generate-test-debug (GTD). It works on both planning and interpretation problems in the domain of geology.[2] An example of the latter: Given a goal state representing a vertical cross-section of a geologic region, find a set of ordered events that would explain the region's formation. Gordius generates a hypothesis (plan), evaluates the hypothesis via simulation, then repairs it if necessary.

TAC differs from Gordius in the same way that it differs from other planning systems: TAC does not generate a sequence of steps that will take it from a specified initial state to a specified goal state. It is similar to Gordius, however, in several interesting ways.

Because both architecture and geology exhibit complex operator interactions, both systems employ a version of generate-and-test. Assuming that some goals will be independent, they produce a solution, then repair the solution if necessary. Both domains necessitate more sophisticated representations than planning's typical propositional representation of world states. In addition to operator interactions, both domains have to deal with creation and destruction of objects, and with spatial effects. Gordius represents states propositionally, but its goal state is represented as a two-dimensional diagram, and it evaluates states using diagrammatic simulation.

Gordius repairs a plan using a dependency structure that identifies the source of failure.[3] This strategy is similar to TAC's dependency-directed redesign. Gordius' repair strategies for modifying faulty parameter values in its plan steps are domain independent and akin to TAC's fixers. The actual repair step is different, however. If Gordius determines that wrong parameter values are the source of the problem, it fixes the problem by setting the values correctly in the corresponding the plan steps. If TAC determines that the source of a problem is an undesired characteristic value, it determines the correct value or direction of change. If the characteristic is noninvertible, however, TAC cannot set the value directly in the design; it must figure out how to modify the design in order to realize the value.

2. It also was run on traditional AI problems, e.g. blocks.

3. Gordius has two kinds of dependency structures. One structure explains the desired parameter value; this structure has no analog since TAC's desired values are user-specified rather than inferred from a causal model. Another structure explains Gordius' simulated value; this structure is akin to TAC's explanation for a goal expression.

Gordius combines plan steps using a unification scheme similar in spirit to that used by TAC's lookahead mechanism. It also prunes conflicting plan steps, as TAC prunes conflicting repair suggestions. Finally, Gordius controls looping during search by employing a lookbehind mechanism that checks for plan equivalence. This step is akin to TAC's checking whether a newly created design is equivalent to one that has been seen before.

The Ancona System

An architectural design planning system built at University of Ancona (Colajanni, 1991) appears to be similar to TAC in a number of respects. It employs multiple representations for designs: a geometric model that is similar to TAC's edge model, a topological model representing connectivity of spaces, and a set of assertions about qualitative geometric relationships between design elements. Given a starting design and a goal, it evaluates the design and creates new designs that satisfy the goal. Designs are created by generating a plan which maps abstract terms, such as "symmetric" into design modification operators, then carrying out the operators. If the designs do not satisfy the goal, the system repairs the designs. (The paper gives insufficient details about how the repair step works. In particular, it is not clear whether the design or the plan is repaired, nor whether an evaluation and repair cycle ensues.[4])

The system differs from TAC in several significant ways. In the referenced paper, all goals are geometric. Operators are represented via preconditions and effects, and are chosen by matching their effects with a goal expression. Effects represent direct changes to a design (e.g. added space), rather than intended goals as in TAC. Simple expression matching for satisfying a goal works for the topological problems illustrated, but would not work for more complex concepts such as visual openness. Such concepts necessitate representations for design elements, not just spaces, as well as representations for relationships between concepts, operators, design elements, and territories induced by design elements.

### 9.1.2 Case Adaptation

Case-based reasoning is concerned with retrieving cases from memory as a starting point for problem solving (Kolodner, 1993; Leake, 1996). An important step in this methodology is adapting the retrieved case (or cases) to a current context. Case adaptation is similar in spirit to TAC's repair: Given a case (design), modify it to meet specified requirements (design goals). With case-based reasoning systems, the starting design is retrieved from a database of cases (a case base). With TAC, we have assumed that the starting design would be sketched by a designer, but

4. Efforts to contact the authors have been unsuccessful.

conceivably it could come from a case base.

Much of the work on case-based reasoning in architecture has concentrated on case indexing and retrieval (e.g. Domeshek and Kolodner, 1992; Oxman, 1996); there has been some work on case adaptation. Adaptation is especially challenging when cases have strong geometrical components, as with architectural designs. Various techniques have been used for adapting designs. We mention several here and give examples of systems that employ them. (See Voss and Oxman (1996) for a survey.)

Constraint satisfaction techniques have been used to adapt cases (e.g. Maher and Zhang, 1993). Some of these systems first adapt a case's topology using graph algorithms, then adapt geometry using constraint satisfaction techniques (e.g. Smith, et. al., 1996; Hua, et. al., 1996). Design knowledge may be represented implicitly in the systems' parameters and constraints (e.g. Smith, et. al., 1996), or explicitly using techniques such as hierarchies of object types (e.g. Giretti and Spalazzi, 1997). Constraint satisfaction techniques are not appropriate for TAC's repair problem because TAC's goals are not stated in terms of topologies or particular geometric arrangements. They are stated in terms of abstract design characteristics that are then mapped into operators that modify design elements, e.g. by changing their geometric arrangements. Because of the noninvertibility of characteristics and the complex interactions between operators, these geometric arrangements are not known apriori. As a result, it is not possible to specify a set of equations representing constraints between design elements.

Model-based reasoning techniques have been used to adapt cases, though typically for engineering fields in which qualitative models of device behavior can be built (e.g. Faltings and Sun, 1994). Even though not in the domain of architecture, the systems described in Goel (1991) and Prabhakar and Goel (1998) are worthy of mention as examples of using explanation of failure (case mismatch) to guide iterative repair. The systems evaluate a retrieved case of a mechanical device via simulation using a causal model of the device's behavior. They then propose modifications by identifying the source of the device failure and selecting repair strategies such as replacing particular device components. This sort of model-based reasoning is not possible for TAC's task because of the complex and unpredictable interactions between design modification operators in architecture; a causal model analogous to device behavior cannot be built. Instead, TAC evaluates a geometric model of a design and uses the resulting explanation, along with mappings between design characteristics and physical form, to repair the design.

### 9.1.3 Performance-based Refinement

The term "performance-based refinement" has been used in the computer-aided architectural design community to mean using desired values of design characteristics (also called performance variables) to guide design refinement—just what TAC does. Most design tools only evaluate design characteristics and do not do repair; they require that the designer "guess" at likely design modifications for affecting desired values. The systems then evaluate the resulting design to see if the modification worked as intended. (See Flemming and Madhavi (1993) for discussion and examples of this sort of design tool.)

TAC is one of the few tools that supports design refinement using explicit knowledge of preferred design characteristic values and methods for achieving those values. The work of Madhavi (1997, 1998), discussed below, falls into this category. The system being built by Kalay (1999) seems headed in a similar direction, but proposes only giving a designer advice about how to achieve desired values.

GESTALT

Madhavi has built two similar systems: GESTALT is a lighting simulation tool for early stages of design (Madhavi 1997); SEMPER is a tool that simulates thermal properties (Madhavi 1998). We discuss GESTALT, which refines a design with respect to stated lighting performance criteria.

GESTALT is similar to TAC in that its goal is to provide a tool for exploring design space, not optimizing a design solution. It assists a designer in finding solutions and in better understanding complex relationships between design parameters. It supports investigations about how changes in performance variables change a design. Its performance variables are similar to TAC's design characteristics, and both systems reason from desired values of characteristics to particular arrangements of design elements. GESTALT, like TAC, adopts a view of the design process as iterative refinement and employs "intelligent" test-and-generate, using knowledge of the relationships between characteristics and physical form to iteratively modify a design. Some of its knowledge is similar to TAC's: it relates experiential characteristics (e.g. lighting quality/comfort) to characteristics of physical form (e.g. window size). It also maps experiential qualities to methods for changing them, just as TAC does. It knows to increase lighting quality, for example, by increasing window area or increasing visible transmittance of window glazing.

GESTALT differs from TAC, however, in that each experiential characteristic is quantitative-valued, and the dependencies between its value and the values of quantitative design attributes can be mathematically modelled or formalized through regression analysis. In addition, functional relationships (e.g. Gaussian, monotonically increasing) can be defined between quantitative design attributes and preference scales (e.g. a five point scale of lighting quality/comfort). The

preference values are used to weight contributions of design attribute values to a performance variable. The end result is that optimization techniques can be used to select particular design attribute values for desired performance values. The user specifies that a particular performance variable be increased, and the system iteratively optimizes design attribute values to produce new designs.

TAC's power would be enhanced by employing this technique with quantitative characteristics for which knowledge of relevant functional relationships exists. It then could relate characteristics to physical form via particular performance functions in lieu of always assuming monotonic relationships. Many of TAC's characteristics are not quantitative, however, and/or correlation data relating characteristic values to physical form attributes does not exist. TAC's reasoning with influences, therefore, cannot be completely replaced. In addition, overall scoring of a design is less meaningful under such circumstances because weighting strategies for various incommensurate criteria are not deducible from correlation statistics. Instead, the weighting is ad hoc.

Finally, many of TAC's characteristics are for a design as a whole, rather than for a single room, and are therefore very dependent on particular arrangements of many design elements. Such global characteristics are not subject to mathematical modeling or statistical analysis techniques because there are too many possible arrangements of design elements in an entire design.

## 9.2 Experiential Knowledge

A few systems have evaluated designs with respect to experiential qualities, but have not represented or reasoned about physical form, relying instead, for example, on human evaluators and statistical scoring techniques (e.g. Mortola and Giangrande, 1991). One such system (Cao and Protzen, 1994) goes a step further and explicitly represents mappings between experiential qualities and physically measurable properties, but relies on previously collected data rather than measuring properties dynamically from a representation of a design. Several case-based reasoning systems have represented experiential qualities, often derived from post occupancy evaluations, as annotations, but have not related them to physical form or evaluation per se (e.g. Domeshek and Kolodner, 1992). A topological evaluation technique that relates spatial organization to social behavior has been successfully paired with a geometric analysis of visibility (Hanson, 1994). Finally, a recent paper (van der Voordt, et al., 1997) suggests relating physical form to experiential qualities, as is done in TAC.

Several systems are close in spirit to TAC's evaluation component. McLaughlin's system (McLaughlin, 1991) focuses on evaluation of a design with respect to experiential qualities represented as design goals such as open, sunny, and private. The system, which is rule-based, takes as input a design and outputs lists of satisfied and unsatisfied goals. The major difference between

this system and TAC's evaluation component is that McLaughlin's system does not represent or reason about physical form. The system instead represents a design as a topological arrangement of design elements—walls, windows, doors, openings, spaces—and reasons about the design using a fact base of assertions. Addition of a geometric representation for a design would simplify some computational tasks (e.g. computation of circulation paths), enable others (e.g. computation of visual barriers), and facilitate interaction with a designer (e.g. by enabling integration with a sketching system).

Two other systems are similar to TAC in their attempts to represent and reason about experiential qualities and physical form. They differ from the current work in ways that result from adopting different views of the design process.

Galle's system (Galle, 1994) aims to be a design support tool, facilitating development of evolving designs. The design knowledge in Galle's system is that embodied in Alexander's pattern language (Alexander, et al., 1977), which represents general design principles as prototypical arrangements (patterns) of physical design elements. Some of the patterns describe methods for achieving experiential qualities. One pattern, for example, suggests achieving an intimacy gradient by creating a sequence of spaces arranged according to degrees of privacy, with least private near an entrance, followed by slightly more private spaces, leading eventually to the most private. Creating a design in Galle's system amounts to instantiating a pattern. Each resulting design element is associated with a pattern, and no design element can be introduced that is not associated with a pattern.[5] TAC takes a different view of design: It aims eventually to support sketching, parsing of the sketch into design elements, followed by design refinement. Design elements are not associated with a particular design goal. They are implicitly related to a design goal if they contribute to satisfaction of that goal, but this relationship is not of importance in TAC. Further, the design goals may be stated in terms of either experiential qualities, which correspond to Galle's (and Alexander's) patterns, or in terms of physical form characteristics that such as lengths of walls. Physical form characteristics have no user-accessible counterpart in Galle's system.

Gullichsen and Chang (Gullichsen and Chang, 1985) built a design generation system based on Alexander's pattern language. It is a rule-based system that generates designs in a top-down fashion, progressing from general patterns to more specific patterns that implement the general ones. The user initiates the generation by specifying the list of patterns to be satisfied. Their system's view of design as a top-down process differs from TAC's view of design as an iterative process of evaluation and repair that is initiated by a user-supplied starting design. Finally, their system's geometric representation of a design may be similar to TAC's, since mention is made of "lower-level procedures [that] typically employ geometric methods." An example is given of

5. Galle suggests in a footnote that later versions of his system will relax this restriction.

computing positive space between buildings by calculating the "sum of areas enclosed by walls of buildings or segments which constitute the convex polygonal hull of the building's wings, weighted by the ratio of its enclosing perimeter of the hull," but no mention is made of the underlying representation.

# Chapter 10

# Discussion

The Architect's Collaborator is a step toward an intelligent assistant for early stages of architectural design. This chapter discusses ways in which it could be extended and summarizes its contributions.

## 10.1 Future Work

The Architect's Collaborator is a prototype design support system. Additional work in several areas would transform TAC into a more valuable design partner.

We chose a two-dimensional representation for designs, simplifying TAC's computational geometry routines so that we could focus on reasoning about experiential qualities and physical form. Extending the representation to three dimensions would be relatively straightforward: the underlying design representation and computational geometry routines could be changed without disrupting the knowledge base or the overall control structure.

TAC's knowledge base contains a rich set of concepts, both architectural and domain-independent. We'd like to extend it to include knowledge of materials and light. The choice of materials contributes to how we experience a space. Contrast the walls of wood and glass in the living room in Figure 1.4, for example, with the plaster walls in Figure 1.3. The use of light also influences our experience of a space. Compare, for example, the fluorescent light in the classroom in Figure 1.1 with the natural light in the studio in Figure 1.2. We'd also like to add knowledge of sociological influences on physical form (e.g. Wright, 1981; Hillier and Hanson, 1984). Changing attitudes about domestic life, for example, transformed the front and back parlors of Victorian times into the modern-day living room.

TAC's languages and representations form a good foundation for addition of knowledge such as that described above. As with all systems that rely on knowledge bases, however, acquiring the knowledge is nontrivial. Machine learning techniques may help with this task. A designer could give the system a set of designs that contain territories he considers visually open, for example, and the system could determine a threshold for the visual openness measurement. Machine learning techniques also might be useful in adding knowledge of discovered synergies and conflicts to TAC's knowledge base. As noted in Section 7.3.1, for example, TAC discovered that moving a design element to an exterior edge could increase visual openness. This move method could be

added as an increaser on the design characteristic `visual-openness`.

TAC's knowledge base could be extended to include knowledge associated with building codes, costs, and remodeling. The system then could propose new designs based on a wider variety of design goals. In the Chatham house design example discussed in Chapter 8, for example, turning the stair in an existing house is far different from turning the stair "on paper". With remodeling knowledge, the system could rank new designs based on criteria such as construction cost and disruption.

TAC's set of design modification operators could be extended to include additional geometric concepts such as alignment and overlap. A designer might want to align two doorways, for example, or overlap two territories. We'd also like to remove the restriction on TAC's current operators that they cannot change a design's footprint, adding operators, for example, that displace exterior edges of a design or add new ones.

TAC would benefit from integration with a sketching tool. The importance of sketching in architectural design is well documented (e.g.Goldschmidt, 1991; Robbins, 1994; Fraser and Henmi, 1994), and we envision, as described in the opening scenario, a designer moving effortlessly between sketching and TAC's evaluation and repair steps. Sketch programs exist that translate a designer's sketch into a model that is equivalent to TAC's design element model (e.g. Gross, 1996). Adding such a program as a front-end to TAC would be straightforward and would alleviate the current need to enter starting designs by hand using a graphics editor. More difficult, but still possible, would be integrating the two systems in such a way that a designer could modify one of TAC's newly generated designs via sketching, then ask TAC to continue its evaluation and repair.

In addition to a sketching interface for TAC, we envision a "knob" interface—attaching "knobs" to design characteristics so that a user could increase or decrease values and observe the resulting changes in physical form. A similar idea is proposed by Flemming and Madhavi (1993) for quantitative characteristics. We'd like to extend the idea to include non-quantitative characteristics as well.

As previously noted, TAC produces many solutions to a design problem and needs to be able to display those solutions without overwhelming the user. Adding the capability to display subsets of designs would be straightforward. The difficult task is identifying meaningful subsets and supporting navigation through those subsets.

TAC's control structure could be extended to support goal refinement, so that a designer could interrupt TAC's evaluation and repair cycle at any point, redefine goals, then have TAC continue. It also could be extended to allow the user to prune designs from a newly generated set before TAC proceeds with its repair steps. Both of these extensions would be straightforward.

TAC's control structure and knowledge base could be extended to assist a designer in specifying goals by suggesting some goals automatically. If a design has a second floor, for example, TAC could specify that the design needs a stair. The issue is whether a designer must specify all goals, or whether some could be considered part of general architectural knowledge. Defining what constitutes general architectural knowledge is a difficult and open-ended problem, but even augmenting the knowledge base with a small set of simple general design goals would decrease the amount of work a designer must do, while increasing the relevancy of proposed design solutions.

The efficiency of TAC's search capabilities could be improved as discussed in Section 7.4 by defining equivalence classes and by ordering goals. Goal order could be determined by specificity of goals and operators, by using domain knowledge (e.g. deal with circulation issues first), or by grouping goals into subsets, perhaps based on goal similarity metrics of some sort. TAC's search capabilities also could be extended to resolve goal conflicts when no solutions are found, e.g. by relaxing goals or looking for compromises.

## 10.2 Summary

The Architect's Collaborator contributes to artificial intelligence answers to these two questions:

- How are experiential qualities translated into physical form?

TAC's languages and representations formalize abstract concepts such as openness and privacy, and relate those concepts to arrangements of physical form.

- How might a tool assist designers with this process?

TAC enables designers to specify design goals and efficiently explore the space of possible designs satisfying those goals. By means of its dependency-directed redesign strategy, TAC is able to deal with issues of a very large search space, noninvertible evaluation functions, complex interactions between design modification operators, and multiple conflicting goals.

The Architect's Collaborator also contributes to architecture: the clarification of terms used in architectural discourse, a system that functions as both a design brainstorming tool and an analysis tool, a repository for reusable design knowledge, and an example of how to distribute tasks between a designer and a computer assistant.

By virtue of these contributions to both artificial intelligence and architecture, The Architect's Collaborator is a step toward an intelligent tool for conceptual architectural design.

# References

## R.1 Bibliography


(1916) Architectural Record, vol. 39, 175.

Akin, O. (1986) *Psychology of Architectural Design*, Pion, London.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobsen, M., Fiksdahl-King, I., and Angel, S. (1977) *A Pattern Language*, Oxford University Press, New York.

Ambros-Ingerson, J. A. and Steel, S. (1988) "Integrating planning, execution, and monitoring," in *Proceedings AAAI -88*, 83-88.

Appleton, J. (1975) *The Experience of Landscape*, John Wiley & Sons, London.

Beetz, M. and McDermott, D. (1994) "Improving robot plans during their execution," in *Proceedings of the Second International Conference on AI Planning Systems*, AAAI Press, Boston, 3-12.

Broadbent, G. (1973) *Design in Architecture: Architecture and the Human Sciences*, John Wiley & Sons, New York.

Broadbent, G. (1980) "A semiotic programme for architectural psychology," in *Meaning and Behaviour in the Built Environment*, G. Broadbent, R. Bunt, and T. Llorens, eds., John Wiley & Sons, New York, 313-359.

Broadbent, G., Bunt, R., and Llorens, T., eds. (1980) *Meaning and Behaviour in the Built Environment*, John Wiley & Sons, New York.

Brooks, H. A. (1972) *The Prairie School*, Norton, New York.

Cao, Q. and Protzen, J.-P. (1994) "Deliberation and aggregation in computer-aided performance evaluation," in *Automation Based Creative Design*, A. Tzonis and I. White, eds., Elsevier, New York, 251-264.

Chan, C.-S. (1992) "Exploring individual style through Wright's designs," *Journal of Architectural Planning and Research*, vol. 9, 207-238.

Colajinni, B., De Grassi, M., di Manzo, M., and Naticchia, B. (1991) "Can planning be a research paradigm in architectural design?," in *Artificial Intelligence in Design '91*, Proceedings of the Second International Conference on Artificial Intelligence in Design, J. S. Gero, ed., Butterworth-Heinemann, Oxford, 23-48.

Coyne, R. D. and Gero, J. S. (1985) "Design knowledge and sequential plans," *Environment and Planning B*, vol. 12, 401-418.

Cross, N., Dorst, K., and Roosenburg, N., eds. (1991) *Research in Design Thinking*, Delft University Press, Delft.

Davies, S. P. and Simplicio-Filho, F. (1992) "Opportunistic and goal-oriented behaviour in software design," in *Artificial Intelligence in Design '92*, Proceedings of the Second International Conference on Artificial Intelligence in Design, J. S. Gero, ed., Kluwer, Norwell, MA, 839-860.

Domeshek, E. A. and Kolodner, J. L. (1992) "A case-based design aid for architecture," in *Artificial Intelligence in Design '92*, Proceedings of the Second International Conference on Artificial Intelligence in Design, J. S. Gero, ed., Kluwer, Norwell, MA, 497-516.

Eastman, C. M. (1969) "Cognitive processes and ill-defined problems: a case study from design," in *Proceedings IJCAI-69*, 669-691.

Eastman, C. M. (1970) "On the analysis of intuitive design processes," in *Emerging Methods in Environmental Design and Planning*, G. T. Moore, ed., MIT Press, Cambridge, 21-37.

Faltings, B. and Sun, K. (1996) "FAMING: Supporting innovative mechanism shape design," *Computer-Aided Design*, vol. 28, 207-216.

Flemming, U. (1994) "Artificial intelligence and design: a mid-term review," in *Knowledge-Based Computer-Aided Architectural Design*, G. Carrara and Y. E. Kalay, eds., Elsevier, New York, 1-24.

Flemming, U. and Mahdavi, A. (1993) "Simultaneous form generation and performance evaluation: a `two-way' inference approach," in *CAAD Futures '93*, Proceedings of the Fifth International Conference on Computer-Aided Architectural Design Futures, U. Flemming and S. Van Wyk, eds., North-Holland, New York, 161-174.

Foz, A. (1973) "Observations on designer behavior in the parti," in *The Design Activity International Conference, Volume 1*, University of Strathclyde, Glasgow, 19.1-19.4. See also "Some observations on designer behavior in the parti", Master of Architecture and Master of City Planning Thesis, Massachusetts Institute of Technology, June, 1972.

Fraser, I. and Henmi, R. (1994) *Envisioning Architecture: An Analysis of Drawing*, Van Nostrand Reinhold, New York.

Galle, P. (1994) "Computer support of architectural sketch design: a matter of simplicity?," *Environment and Planning B*, vol. 21, 353-372.

Giretti, A. and Spalazzi, L. (1997) "ASA: a conceptual design-support system," *Engineering Applications of Artificial Intelligence*, vol. 10, 99-111.

Goel, A. K. (1991) "A model-based approach to case adaptation," in *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, Lawrence Erlbaum, Chicago, 143-148.

Goldschmidt, G. (1991) "The dialectics of sketching," *Creativity Research Journal*, vol. 4, 123-143.

Gross, M. D. (1996) "The Electronic Cocktail Napkin—a computational environment for working with design diagrams," *Design Studies*, vol. 17, 53-69.

Gullichsen, E. and Chang, E. (1985) "Generative design architecture using an expert system," *The Visual Computer*, vol. 1, 161-168.

Hammond, K. J. (1990) "Explaining and repairing plans that fail," *Artificial Intelligence*, vol. 45, 173-228.

Hanson, J. (1994) "'Deconstructing architects' houses," *Environmental and Planning B*, vol. 21, 675-704.

Heath, T. (1984) *Method in Architecture*, John Wiley & Sons, New York.

Hertzberger, H. (1993) *Lessons for Students in Architecture*, 2nd ed, Uitgeverij 010 Publishers, Rotterdam.

Hildebrand, G. (1991) *The Wright Space: Pattern and Meaning in Frank Lloyd Wright's Houses*, University of Washington Press, Seattle.

Hillier, B. and Hanson, J. (1984) *The Social Logic of Space*, Cambridge University Press, Cambridge.

Hua, K., Faltings, B., and Smith, I. (1996) "CADRE: case-based geometric design," *Artificial Intelligence in Engineering*, vol. 10, 171-183.

Jones, J. C. (1970) *Design Methods*, John Wiley & Sons, New York.

Jones, R. T. (1987) *Authentic Small Homes of the Twenties: Illustrations and Floorplans of 254 Characteristic Homes*, Dover Publications, New York.

Kalay, Y. E. (1999) "Performance-based design," *Automation in Construction*, vol. 8, 395-409.

Kambhampati, S. and Hendler, J. A. (1992) "A validation-struture-based theory of plan modification and reuse," *Artificial Intelligence*, vol. 55, 193-258.

Kartam, N. A., Levitt, R. E., and Wilkins, D. E. (1991) "Extending artificial intelligence techniques for hierarchical planning," *Journal of Computing in Civil Engineering*, vol. 5, 464-477.

Kincaid, D. S. (1996) Conversation with the author, 11/16/96.

Kincaid, D. S. (1997) "An arithmetical model of spatial definition," Master of Architecture Thesis, Department of Architecture, Massachusetts Institute of Technology.

Knoblock, C. A. (1995) "Planning, executing, sensing, and replanning for information gathering," in *Proceedings of IJCAI-95*, 1686-1693.

Koile, K. (1997) "Design conversations with your computer: evaluating experiential qualities of physical form," in *CAAD Futures '97*, Proceedings of the Seventh International Conference on Computer-Aided Architectural Design Futures, R. Junge, ed., 203-218.

Kolodner, J. (1993) *Case-Based Reasoning*, Morgan-Kaufmann, San Mateo.

Koning, H. and Eizenberg, J. (1981) "The language of the prairie: Frank Lloyd Wright's prairie houses," *Environment and Planning B*, vol. 8, 295-323.

Korf, R. E. (1987) "Planning as search: a quantitative approach," *Artificial Intelligence*, vol. 33, 65-88.

Krauss, R. I. (2000) Conversation with the author, 3/15/00.

Krauss, R. I. and Myer, J. R. (1970) "Design: a case history," in *Emerging Methods in Environmental Design and Planning*, G. T. Moore, ed., MIT Press, Cambridge, 11-20.

Lawson, B. (1990) *How Designers Think*, 2nd ed, Butterworth Architecture, Boston.

Lawson, B. (1994) *Design in Mind: The Design Process Demystified*, Butterworth Architecture, Boston.

Leake, D. B., ed. (1996) *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, AAAI Press/MIT Press, Menlo Park/Cambridge.

Lind, C. (1994a) *Frank Lloyd Wright's Life and Homes*, Pomegranate Artbooks, San Francisco.

Lind, C. (1994b) *Frank Lloyd Wright's Usonian Houses,* Pomegranate Artbooks, San Francisco.

Mahdavi, A. and Suter, G. (1997) "On implementing a computational facade design support tool," *Environment and Planning B*, vol. 24, 493-508.

Mahdavi, A. and Suter, G. (1998) "On the implications of design process views for the development of computational design support tools," *Automation in Construction*, vol. 7, 189-204.

Maher, M. L. and Zhang, D. M. (1993) "CADSYN: A case-based design process model," *Artificial Intelligence in Engineering Design, Analysis and Manufacturing*, vol. 7, 97-110.

Manson, G. C. (1958) *Frank Lloyd Wright to 1910: The First Golden Age*, Van Nostrand Reinhold, New York.

McCoy, E. (1975) *Five California Architects*, Praeger, New York.

McDermott, D. and Hendler, J. (1995) "Planning: What it is, What it could be, An introduction to the Special Issue on Planning and Scheduling," *Artificial Intelligence*, vol. 76, 1-16.

McLaughlin, S. (1991) "Reading architectural plans: a computable model," in *CAAD Futures '91*, Proceedings of the Fourth International Conference on Computer-Aided Architectural Design Futures, G. N. Schmitt, ed., Vieweg, Wiesbaden, 347-364.

Moore, C., Allen, G., and Lyndon, D. (1974) *The place of houses*, Holt, Rinehart and Winston, New York.

Mortola, E. and Giangrande, A. (1991) "An evaluation module for 'An Interface for Designing' (AID): a procedure based on trichotomic segmentation," in *CAAD Futures '91*, Proceedings of the Fourth International Conference on Computer-Aided Architectural Design Futures, G. N. Schmitt, ed., Vieweg, Wiesbaden, 139-154.

Oxman, R. (1996) "Case-based design support: supporting architectural composition through precedent libraries," *Journal of Architectural and Planning Research*, vol. 13, 242-255.

Petit, J. L. (1854) *Architectural Studies in France*, London. Referenced in Etlin, R. A. (1994) *Frank Lloyd Wright and LeCorbusier*, Manchester University Press, New York.

Pinnell, P. (1990) "Academic tradition and the individual talent: similarity and difference in the formation of Frank Lloyd Wright," in *Frank Lloyd Wright: A Primer on Architectural Principles*, R. McCarter, ed., Princeton Architectural Press, New York, 19-58.

Prabhakar, S. and Goel, A. K. (1998) "Functional modeling for enabling adaptive design of devices for new environments," *Artificial Intelligence in Engineering*, vol. 12, 417-444.

Rapoport, A. (1969) *House Form and Culture,* Prentice-Hall, Englewood Cliffs, NJ.

Rapoport, A. (1977) *Human Aspects of Urban Form*, Pergamon Press, New York.

Rasmussen, S. E. (1962) *Experiencing Architecture*, MIT Press, Cambridge, MA.

Robbins, E. (1994) *Why Architects Draw*, MIT Press, Cambridge, MA.

Rowe, P. G. (1987) *Design Thinking*, MIT Press, Cambridge, MA.

Schön, D. (1983) *The Reflective Practioner: How Professionals Think in Action*, Basic Books, New York.

Scully, V., Jr. (1971) *The Shingle Style and Stick Style: Architectural Theory and Design from Richardson to the Beginnings of Wright*, Yale University Press, New Haven, CT.

Shneiderman, B. (1998) *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison Wesley, Reading, MA.

Simmons, R. G. (1992) "The roles of associational and causal reasoning in problem solving," *Artificial Intelligence*, vol. 53, 159-208.

Simon, H. (1969) *Sciences of the Artificial*, 2nd ed, MIT Press, Cambridge, MA.

Simon, H. (1973) "The structure of ill-structured problems," *Artificial Intelligence*, vol. 4, 181-201.

Simon, H. (1975) "Style in Design," in *Spatial Synthesis in Computer-Aided Building Design*, C. M. Eastman, ed., Applied Science, London, 287-309.

Smith, I., Stalker, R., and Lottaz, C. (1996) "Creating design objects from cases for interactive spatial composition," in *Artificial Intelligence in Design '96*, Proceedings of the Fourth International Conference on Artificial Intelligence in Design, J. S. Gero, ed., Kluwer, Norwell, MA, 97-116.

Smithers, T. (1996) "On knowledge level theories of design process," in *Artificial Intelligence in Design '96*, Proceedings of the Fourth International Conference on Artificial Intelligence in Design, J. S. Gero, ed., Kluwer, Norwell, MA, 561-579.

Stallman, R. and Sussman, G. (1977) "Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis," *Artificial Intelligence*, vol. 9, 135-196.

Stickley, G. (1982) *More Craftsman Homes: Floor Plans and Illustrations for 78 Mission Style Dwellings*, Dover Publications, New York.

Storrer, W. A. (1993) *The Frank Lloyd Wright Companion*, University of Chicago Press, Chicago. Also available as The Frank Lloyd Wright Companion CD-ROM, Prairie Multimedia, Inc., West Chicago.

Twombly, R. C. (1979) *Frank Lloyd Wright: His Life and His Architecture*, John Wiley & Sons, New York.

van der Voordt, T. J. M., Vrielink, D., and van Wegen, H. B. R. (1997) "Comparative floorplan-analysis in programming and architectural design," *Design Studies*, vol. 18, 67-88.

Voss, A. and Oxman, R. (1996) "A study of case adaptation systems," in *Artificial Intelligence in Design '96*, Proceedings of the Fourth International Conference on Artificial Intelligence in Design, J. S. Gero, ed., Kluwer, Norwell, MA, 173-189.

Wade, J. W. (1977) *Architecture, Problems, and Purposes: Architectural Design as a Basic Problem-Solving Process*, John Wiley & Sons, New York.

Wilkins, D. E. (1988) *Practical Planning*, Morgan Kaufmann, San Mateo.

Wilkins, D. E. (1990) "Can AI planners solve practical problems?," *Computational Intelligence*, vol. 6, 232-246.

Wright, F. L. (1954) *The Natural House*, Horizon Press, New York.

Wright, G. (1981) *Building the Dream: A Social History of Housing in America*, MIT Press, Cambridge, MA.

Zeisel, J. and Welch, P. (1981) *Housing Designed for Families: A Summary of Research*, Joint Center for Urban Studies of MIT and Harvard University, Cambridge, MA.

## R.2 Illustration Credits

Chapter 1

Figure 1.2, 1.4: Lind (1994a, 1994b)

Figure 1.5: Duncan Kincaid and Daniel Gorini

Chapter 3

Figure 3.2: Storrer (1993)

Chapter 4

Figures 4.1, 4.3, 4.4: Storrer (1993)

Chapter 5

Figures 5.2, 5.5: Storrer (1993)

Chapter 7

Figure 7.7: Storrer (1993)

Chapter 8

Figures 8.1, 8.3, 8.4: Daniel Gorini

Figures 8.27: Kimberle Koile

Figures 8.32, 8.33: as indicated in figures

# Appendix A Diagram Conventions

Below are conventions used throughout this document for diagrams that represent floorplans.

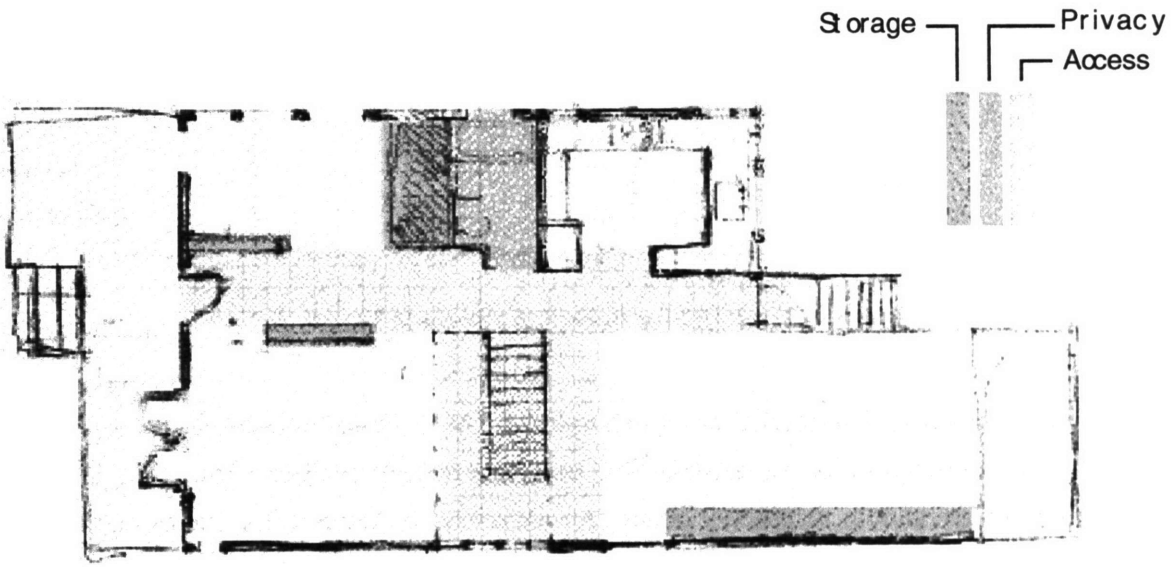



**Figure A.1:** Floorplan drawing.

Figure A.1 shows a floorplan drawing and the key for shaded areas. Storage includes closets, bookcases, etc. Privacy refers to private areas, which include bathrooms and bedrooms. Access refers to well-traveled regions, which include but are not limited to hallways. Unshaded areas represent public regions.

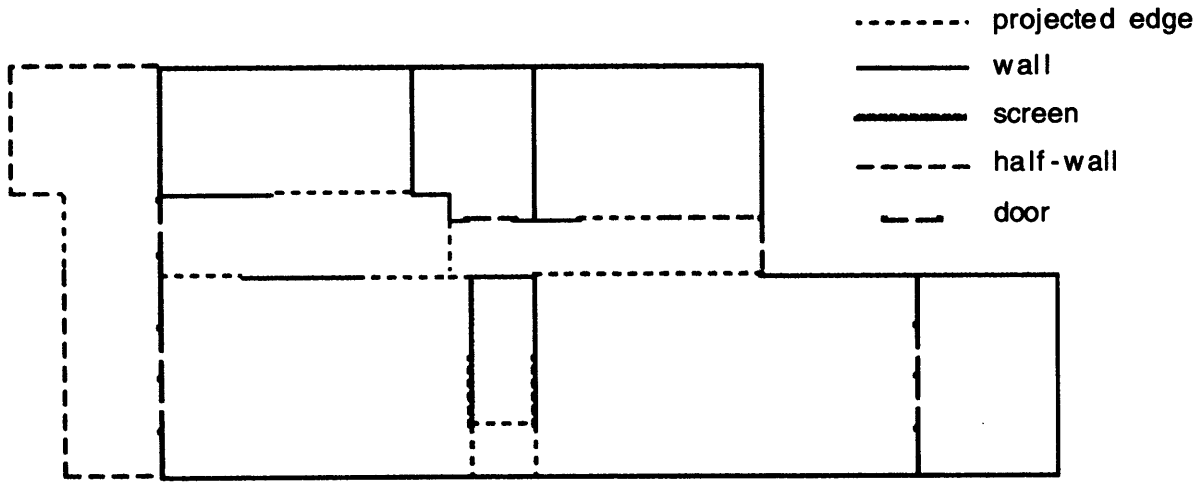



**Figure A.2:** Model for floorplan shown in Figure A.1.

The model shown in Figure A.2 is an example of TAC's representation for floorplans. (See Section 3.1 for a description of the models TAC uses to represent designs.) The line types used to depict the model are shown in the key above. As described in Section 3.1, edges correspond to design elements (e.g. walls, doors) or projections from design elements. Windows and exterior steps currently are not included in the models. For visibility measurements, projected edges are open (i.e., they would be invisible in the building corresponding to the design). Edges representing walls are opaque. Screens (e.g. stair railings) and half-walls are considered semi-open. Edges representing interior doors are considered open; edges representing exterior doors are considered opaque. A stair is depicted as a closed polygon with one open edge representing the bottom of the stair.

# Appendix B Knowledge Base

Below are lists of the design element types, TAC-functions, design characteristics, and design modifiers in TAC's knowledge base.

## B.1 Design Element Types

```
bookcase
door
doorway
fireplace
half-wall
screen
steps
   staircase
wall
window
```

## B.2 TAC-functions

### Arithmetic Relations

```
gt
  :fixer gt-fixer
  :eval-fcn-name greater-than

lt
  :fixer lt-fixer
  :eval-fcn-name less-than

gte
  :fixer gte-fixer
  :eval-fcn-name greater-than-or-equal-to

lte
  :fixer lte-fixer
  :eval-fcn-name less-than-or-equal-to

eq
  :fixer eq-fixer
  :eval-fcn-name equal-to
```

```
not-eq
  :fixer not-eq-fixer
  :eval-fcn-name not-equal-to

more-of
   :fixer more-of-fixer
   :eval-fcn-name more-of

less-of
   :fixer less-of-fixer
   :eval-fcn-name less-of
```

## Logical Relations

```
or
  :fixer or-fixer
  :eval-fcn-name my-or

and
  :fixer and-fixer
  :eval-fcn-name my-and

not
  :fixer not-fixer
  :eval-fcn-name lisp::not
```

## Computational Constructs

```
if
  :fixer if-fixer
  :eval-fcn-name lisp::if

if-nc
  :fixer if-nc-fixer
  :eval-fcn-name if-nc

is-a-value
  :fixer value-fixer
  :eval-fcn-body (not-eq x :no-value)

make-vector
  :fixer vector-fixer
  :eval-fcn-name lisp::list

top-of
  :fixer partial-order-fixer
  :eval-fcn-name top-of
```

```
tops-of
  :eval-fcn-name tops-of

typep
  :fixer boolean-general-fixer
  :t-setters ((replace-elt x y))
  :nil-setters ((replace-elt x :any (transf-types-for y)))
  :eval-fcn-name typep
```

## Set Concepts

```
number-of
  :fixer number-of-fixer
  :eval-fcn-name length

some
  :fixer some-fixer
  :eval-fcn-name lisp::some
```

## Geometric Concepts

```
center
  :eval-fcn-name physical-center-point

segment-btw
  :eval-fcn-name segment-between

open-edges-btw
  :eval-fcn-name open-edges-btw

distance-btw
  :setters ((move x) (move y))
  :eval-fcn-name distance-between

change-in-direction-btw
  :increasers ((increase-change-in-direction-btw x y))
  :decreasers ((decrease-change-in-direction-btw x y))
  :eval-fcn-name change-in-direction-btw

angles-that-minimize-projection-btw
  :eval-fcn-name angles-that-minimize-projection-btw

x-in-y
  :eval-fcn-name x-in-y

relative-size
  :eval-fcn-name compare-relative-size
```

```
largest
  :eval-fcn-name largest

smallest
  :eval-fcn-name smallest
```

## B.3 Design Characteristics

See also Appendix G for design characteristics related to the Prairie house experiment described in Section 8.2.

### Architectural Concepts

```
blocking-elements-btw
  :eval-fcn-name blocking-elements-btw

blocking-elements+edges-btw
  :eval-fcn-name blocking-elements+edges-btw

visible-from
  :eval-fcn-name visible-from
  :fixer boolean-fixer
  :t-setters ((remove (blocking-elements-btw x y) :to-avoid (segment-btw x y))
    (puncture (blocking-elements+edges-btw x y) :at (segment-btw x y))
    (screenify (blocking-elements-btw x y) :at (segment-btw x y)))
  :nil-setters ((fill * :any (open-edges-btw x y) :along (segment-btw x y))))

visible-center
  :eval-fcn-body (visible-from (center x) (center y))

opacity-of-elements-btw
  :decreasers ((remove (blocking-elements-btw x y))
               (puncture (blocking-elements-btw x y))
               (screenify (blocking-elements-btw x y))
               (rotate (blocking-elements-btw x y) :through * :any
                       (angles-that-minimize-projection-btw x y))
               (move (blocking-elements-btw x y) :to-edge :any
                       (exterior-edges-for-elt *)))
  :increasers ((fill * :any (open-edges-btw x y))
               (screenify * :any (open-edges-btw x y)))

visual-openness
  :min-value 0.0
  :max-value 1.0
```

```
  :eval-fcn-name vis-openness
  :influences ((- (opacity-of-elements-btw x y)))

visually-open
  :eval-fcn-body (gt (visual-openness x y) .6)

visually-very-open
  :eval-fcn-body (gt (visual-openness x y) .8)

visually-closed
:eval-fcn-body (lt (visual-openness x y) .3)

physical-accessibility
  :influences ((- (distance-btw x y))
               (- (change-in-direction-btw x y)))

built-exterior-path-p
  :fixer boolean-fixer
  :t-setters ((build-exterior-path to from))
  :nil-setters ((remove-exterior-path to from))
  :eval-fcn-name built-exterior-path-p

built-exterior-paths
  :fixer sequence-fixer
  :non-nil-setters ((build-exterior-path to from))
  :nil-setters ((remove-exterior-paths to from))
  :eval-fcn-name built-exterior-paths

exteriorp
  :eval-fcn-name exteriorp

solidity
  :min-value 0.0
  :max-value 1.0
  :setters ((change-elt-solidity x))
  :eval-fcn-name surface-opacity

degree-of-hinge
  :ordered-range-values (folding sliding hinged)
  :setters ((change-door-hinge-type x))
  :eval-fcn-name door-type

formality-of-entry
  :influences ((+ (solidity x))
               (+ (degree-of-hinge x))) ; hinged > sliding > folding > no-door

ordered-elts
  :eval-fcn-name find-ordered-elements
```

```
ordered-perceived-main-entries
  :eval-fcn-body (ordered-elts x 'perceived-main-entryness 'more-of)

perceived-main-entry
  :eval-fcn-body (top-of (ordered-perceived-main-entries x))

perceived-main-entries
  :eval-fcn-body (tops-of (ordered-perceived-main-entries x))

one-perceived-main-entry
  :eval-fcn-body (if (is-a-value (perceived-main-entry x) true false)

perceived-main-entryness
  :necessary-conditions ((typep x 'exterior-door)
                         (built-exterior-paths x 'usual-approach)
                         (visible-from x 'usual-approach))
  :influences ((+ (physical-accessibility x 'usual-approach))
               (+ (formality-of-entry x)))

privacy
  :influences ((- (visual-openness x y))
               (- (physical-accessibility x y)))

elt-type-count
  :eval-fcn-body (number-of (elts-of-type type dmodel-or-terr))
  :increasers ((add-some-elts-of-type type :to dmodel-or-terr))
  :decreasers ((remove-some-elts-of-type type :from dmodel-or-terr))

elts-of-type
  :eval-fcn-name elts-of-type
  :fixer sequence-fixer
  :non-nil-setters ((add-some-elts-of-type type :to dmodel-or-terr :number 1))
  :nil-setters ((remove-some-elts-of-type type :from dmodel-or-terr :number
                 :all))

elt-on-interior-edge-p
  :eval-fcn-name elt-on-interior-edge-p
  :fixer boolean-fixer
  :t-setters ((move elt :to-edge :any (interior-edges-for-elt elt)))
  :nil-setters ((move elt :to-edge :any (exterior-edges-for-elt elt)))

elt-type-on-interior-edge-p
  :eval-fcn-body (some 'elt-on-interior-edge-p
                       (elts-of-type elt-type dmodel-or-terr)))
```

```
elt-on-exterior-edge-p
  :eval-fcn-name elt-on-exterior-edge-p
  :fixer boolean-fixer
  :t-setters ((move elt :to-edge :any (exterior-edges-for-elt elt)))
  :nil-setters ((move elt :to-edge :any (interior-edges-for-elt elt)))

elt-type-on-exterior-edge-p
  :eval-fcn-body (some 'elt-on-exterior-edge-p
                       (elts-of-type elt-type dmodel-or-terr)))

elt-type-in-territory-p
  :eval-fcn-name elt-type-in-territory-p
  :fixer boolean-fixer
  :t-setters ((add-some-elt-of-type type :to territory :number 1))
  :nil-setters ((remove-some-elts-of-type type :from territory :number :all))

adjacent
  :eval-fcn-body adjacent

use-adjacent
  :eval-fcn-body (adjacent (use-space-territory x) (use-space-territory y))
  :fixer boolean-fixer
  :t-setters ((exchange-use x :any (use-spaces-adjacent-to y)))
  :nil-setters ((exchange-use x :any (use-spaces-not-adjacent-to y)))

use-spaces-adjacent-to
  :eval-fcn-name get-use-spaces-adjacent-to

use-spaces-not-adjacent-to
  :eval-fcn-name get-use-spaces-not-adjacent-to

use-space-territory
  :eval-fcn-name territory-for

path-from-x-to-y
  :eval-fcn-name shortest-path-from-x-to-y

paths-from-x-to-y
  :eval-fcn-name all-paths-from-x-to-y

path-crosses
  :eval-fcn-name path-crosses-territory

on-different-levels
  :eval-fcn-name on-different-levels

opens-into
  :eval-fcn-name door-opens-into-territory
```

## B.4 Design Modifiers

move
puncture
fill
screenify
unscreenify
remove
rotate
add-elt-of-type
remove-elts-of-type
add-some-elts-of-type
remove-some-elts-of-type
add-blocking-elt-btw
replace-elt
change-elt-solidity
change-door-hinge-type
build-exterior-path
remove-exterior-paths
remove-exterior-territories-and-paths
increase-change-in-direction-btw
decrease-change-in-direction-btw
maybe-move-exterior-territories
maybe-move-exterior-access
exchange-use

# Appendix C Language Terms

Below are lists of terms used in TAC's languages.

## Design Characteristic Definition Language

names of design characteristics

names of TAC-functions

names of design element types

symbols: usual-approach, street

variable names

## Goal Specification Language

above terms in Design Characteristic Definition Language

design objects

## Repair Suggestion Language

above terms in Design Characteristic Definition Language

names of design modifiers

value suggestion terms: increase-value, decrease-value, set-value, keep-value

keywords: at, along, to-edge, to-avoid, to-block, through, to, from, number, any, all, until

variable placeholder for iteration construct: *

## Constraint Specification Language

same as Goal Specification Language

# Appendix D Alternate Territory Models

As mentioned in Section 3.1, a design may be represented by several territory models. Below are two territory models for the Tomek house. Figure D.1 shows a territory model in which the Living territory extends the full width of the design. In the territory model shown in Figure D.2, the Living territory is smaller, and two new territories have been added, called Access-1 and Access-2. The new territories are formed by extensions of the walls of the fireplace, alcove, and doors to the terrace. As shown in Figure 4.3, they may also be formed by furniture. Which model is of interest is the designer's decision. We have chosen in this document to use the model shown in Figure D.1.

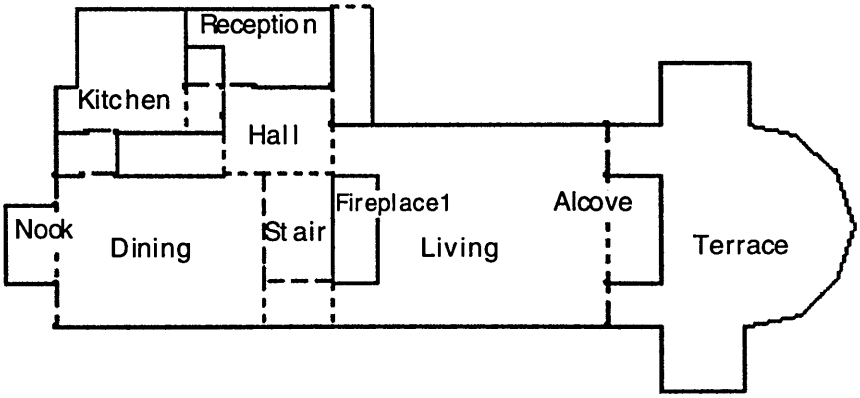



**Figure D.1:** Territory model for Tomek house first floor.

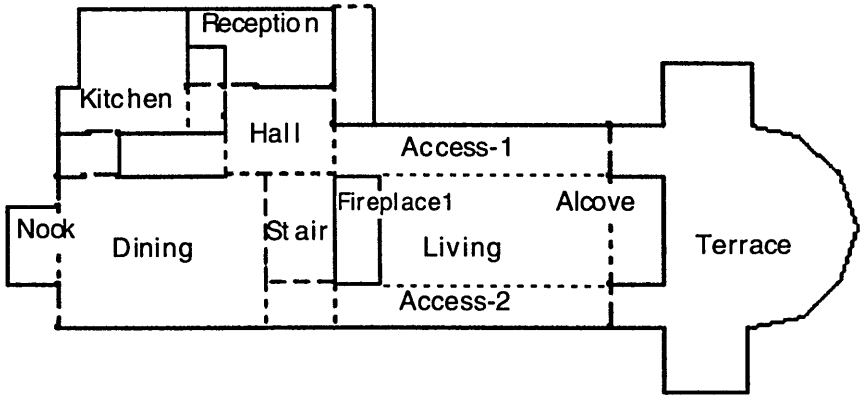



**Figure D.2:** Alternate territory model for Tomek house first floor.

# Appendix E Explanation Examples

## E.1 Building Explanations

Explanations are partially built at compile-time, then completed at run-time. At compile-time, explanation templates, which contain placeholders for run-time values, are built for each design characteristic. Building the explanation tree structure once at compile-time is more efficient than building the structure repeatedly at run-time. At run-time the explanation templates are copied and values are filled in. This scheme is possible because the relationship between design characteristics does not change at run-time. Whenever a design characteristic is defined, or redefined, a new explanation template is built.

The design characteristic `visual-openness` is defined as:

```
(define-design-char visual-openness
   :eval-fcn-lambda-list (a b)
   ...
   :eval-fcn-name calculate-visual-openness)
```

Using the name, lambda list, and evaluation function name in the above definition, the explanation template for this design characteristic looks like:

```
<expl: (visual-openness a b)>
Instance slots:
  value:          :unbound
  expr:           (visual-openness a b)
  expr-types:      (:fcn :arg :arg)
  expr-via:       nil
  next-expl:      calculate-visual-openness
  explains:       nil
```

The unbound and `nil`-valued slots will be filled in at run-time. The expression in the `expr` slot is constructed from the name of the design characteristic and the lambda list stored in the design characteristic. The `expr-types` slot value, which is computed by parsing the expression, identifies the type of each element in the expression. Since the evaluation function for the design characteristic `visual-openness` is an existing function, the name of that function is used as the explanation in the `next-expl` slot.

The next example illustrates how an explanation template is constructed when an evaluation function body is supplied instead of an evaluation function name. As shown in Figure E.1 and described below, the explanation template for `visually-open` is a bit more complicated and must contain the tree structure expected at run-time. A run-time explanation for `visually-open` is shown in Figure 4.6.

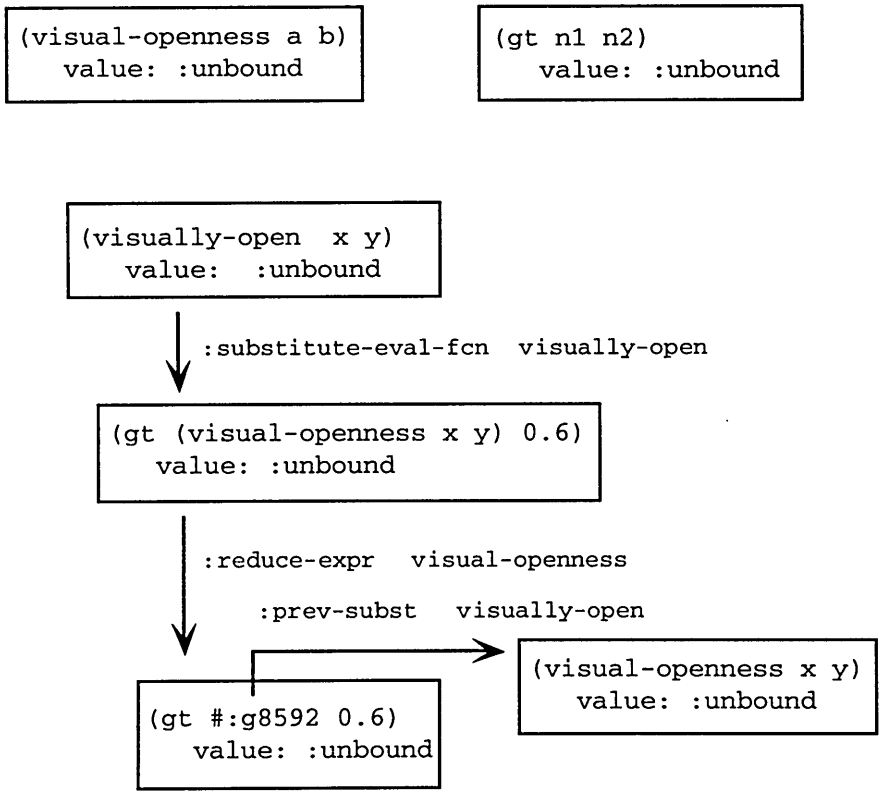



**Figure E.1:** Three explanation templates.
The `visually-open` template is built using the `visual-openness` and `gt` templates.

The design characteristic `visually-open` is defined as:

```
(define-design-char visually-open
   :eval-fcn-lambda-list (x y)
   ...
   :eval-fcn-body (gt (visual-openness x y) 0.6))
```

Using the name, lambda list, and function body information, the explanation template for `visually-open` contains a root node that looks like this:

```
<expl: (visually-open x y)>
Instance slots:
  value:          :unbound
  expr:           (visually-open x y)
  expr-types:     (:fcn :arg :arg)
  expr-via:       nil
  next-expl:      <expl: (gt (visual-openness x y) 0.6)>
  explains:       nil
```

Again, the expression is constructed from the design characteristic's name and lambda list, and the `expr-types` is constructed from the expression. The `next-expl` slot contains an explanation template that represents the next step in the evaluation trace, namely, the substitution of the evaluation function body for `visually-open`.

This `next-expl` explanation node template looks like:

```
<expl: (gt (visual-openness x y) 0.6)>
Instance slots:
  value:          :unbound
  expr:           (gt (visual-openness x y) 0.6)
  expr-types:     (:fcn :expr :const)
  expr-via:       ((:substitute-eval-fcn . <design-char: visually-open>))
  next-expl:      <expl: (gt #:g8592 0.6)>
  explains:       nil
```

The expression in the `expr` slot is the evaluation function body for `visually-open`. The `expr-types` is constructed by parsing the expression. An explanation template will be constructed for each embedded expression.

The term `#:g8592` shown above is a placeholder for the a run-time value. The explanation template in which it appears looks like:

```
<expl: (gt #:g8592 0.6)>
Instance slots:
  value:          :unbound
  expr:           (gt #:g8592 0.6)
  expr-types:     (:fcn <expl: (visual-openness x y)> :const)
  expr-via:       ((:reduce-expr . <design-char: visual-openness>))
  next-expl:      greater-than
  explains:       nil
```

This explanation template was copied from the template for `gt`, shown below, and its expression was constructed by substituting the current context's arguments, `#:g8592` and 0.6, into the expression for `gt`. The `expr-types` slot contains an explanation node template for the embedded expression, `(visual-openness x y)`. This explanation node template was copied from the `visual-openness` template shown as the first example above. Corresponding arguments `x` and `y` in the embedded expression were substituted into the expression `(visual-openness a b)` in the copied template.

```
<expl: (gt n1 n2)>
Instance slots:
  value:           :unbound
  expr:            (gt n1 n2)
  expr-types:       (:fcn :arg :arg)
  expr-via:        nil
  next-expl:       greater-than
  explains:        nil
```

After corresponding argument substitutions, all explanation node templates in the tree contain expressions written in terms of a consistent set of variable names.

## E.2 Example

Below is the explanation for the example presented in Section 4.2. The goal expression for the Tomek house is `(visually-open Living Dining)`. The territory objects Living and Dining are shown here as `<territory: Living>` and `<territory: Dining>`, respectively.

If we examine the root explanation node in the explanation shown in Figure 4.6, we find that it looks like:

```
<expl: (visually-open <territory: Living> <territory: Dining>)>
Instance slots:
  value:           nil
  expr:            (visually-open <territory: Living> <territory: Dining>)
  expr-types:      (:fcn :arg :arg)
  expr-via:        ((:user-supplied))
  explains:        <goal: (visually-open <territory: Living>
                                         <territory: Dining>) t>
  next-expl:       <expl: (gt (visual-openness <territory: Living>
                                               <territory: Dining>) 0.6)>
```

The `value` and `expr` slots are the expression's value and expression, respectively. The `expr-types` slot gives the types for the terms in the expression: a function name and two arguments. The `expr-via` slot documents how the expression came about: it was supplied by the user, in this case as an expression in a goal. The `explains` slot points back to the goal whose expression is explained by this explanation node. Finally, the `next-expl` slot holds the next explanation node in the trace.

Examining the next-expl explanation node above, we find that it looks like:

```
<expl: (gt (visual-openness <territory: Living> <territory: Dining>) 0.6)>
Instance slots:
  value:      nil
  expr:       (gt (visual-openness <territory: Living> <territory: Dining>)
                  0.6)
  expr-types:     (:fcn :expr :const)
  expr-via:      ((:substitute-eval-fcn . <design-char: visually-open>))
  explains:      <expl: (visually-open <territory: Living>
                                       <territory: Dining>)>
  next-expl:     <expl: (gt 0.44 0.6)>
```

The expr-types slot looks a bit different now: it documents that the expression in this explanation node consists of a function name, an embedded expression, and a constant. The embedded expression, the second term in the expression, came into being as a result of substituting the body of the visually-open evaluation function into the original goal expression. This substitution information is conveyed via the expr-via slot. The explains slot points back to the previous explanation node.

Examining the next-expl explanation node above, we find that it looks like:

```
<expl: (gt 0.44 0.6)>
Instance slots:
  value:           nil
  expr:            (gt 0.44 0.6)
  expr-types:       (:fcn
                    <expl: (visual-openness <territory: Living>
                                            <territory: Dining>)>

                     :const)
  expr-via:        ((:reduce-expr . <design-char: visual-openness>))
  explains:        <expl: (gt (visual-openness <territory: Living>
                                               <territory: Dining>) 0.6)>
  next-expl:       greater-than
```

The expr-types slot again looks a bit different: it documents that the expression in this explanation node consists of a function name, a value which has an explanation, and a constant. The second term in the expr-types slot is the explanation node whose value is the 0.44 in the expression. The expr-via slot documents that the value was the result of reducing an expression containing the design characteristic visual-openness. The expression that was reduced, namely (visual-openness <territory: Living> <territory: Dining>), can be found in the explanation node stored in the explains slot . The next-expl slot indicates that the value of this explanation node was the result of evaluating the function greater-than, the evaluation function

for the TAC-function `gt`, with the supplied arguments.

Finally, examining the explanation node in the `expr-types` slot, which explains the derivation of the value 0.44, we find that it looks like:

```
<expl: (visual-openness <territory: Living> <territory: Dining>)>
Instance slots:
  value:            0.44
  expr:             (visual-openness <territory: Living> <territory: Dining>)
  expr-types:        (:fcn :arg :arg)
  expr-via:         ((:prev-subst <design-char: visually-open>))
  explains:         <expl: (gt 0.44 0.6)>
  next-expl:        calculate-visual-openness
```

The `expr-types` slot documents that the expression is a function name and two arguments, and the `expr-via` slot documents that the expression was supplied in a previous evaluation function substitution step. The `next-expl` slot indicates that the value 0.44 was the result of evaluating the function `calculate-visual-openness`, the evaluation function for `visual-openness`, with the supplied arguments.

# Appendix F Modifying Edge and Territory Models

The following figures show the steps in rotating the stair in the example presented in Chapter 6.

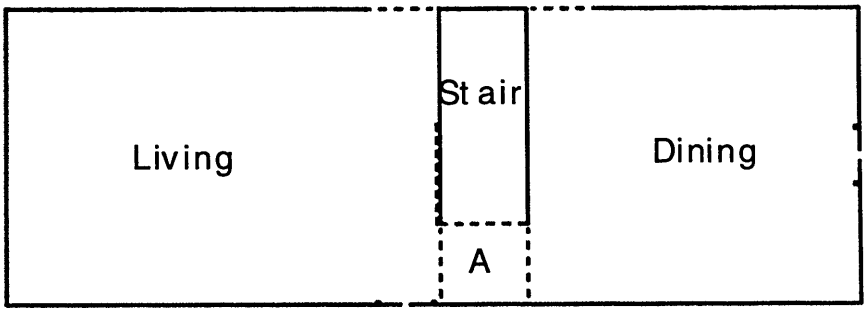



**Figure F.1:** Portion of territory model for Chatham.
Territory model contains territories and edges that bound territories.

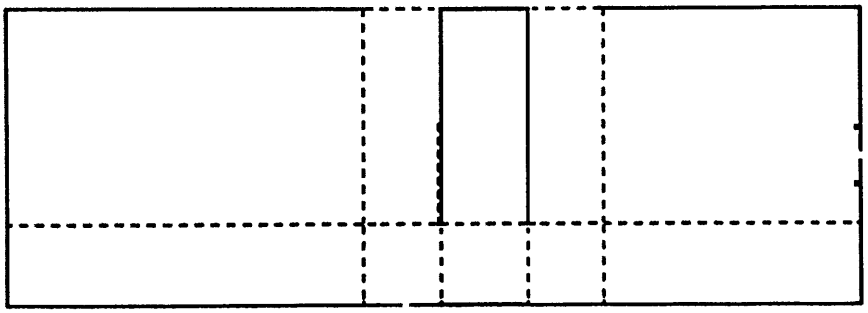


**Figure F.2:** Portion of edge model for Chatham.
Edge model contains all edges.

Modification to be carried out: rotate stair to increase visual openness of Living from Dining.

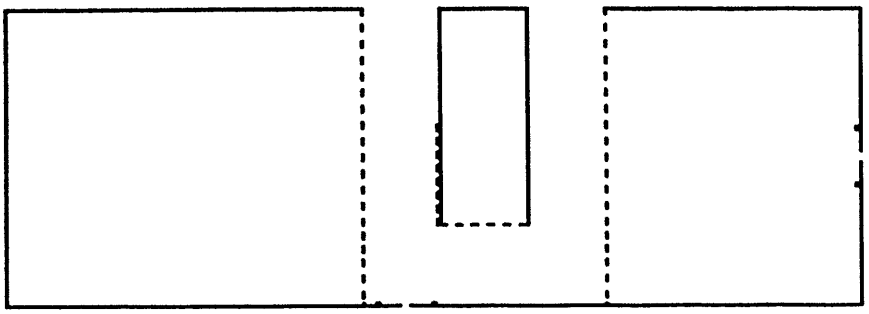


**Figure F.3:** Remove stair's projected edges.
Note that some of removed edges were also derived from walls to left and right of stair.

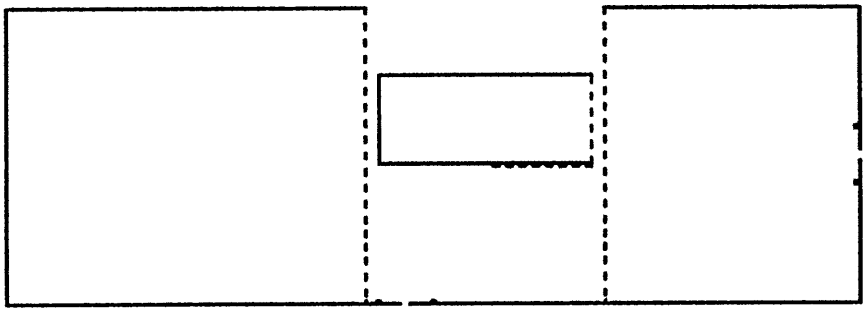


**Figure F.4:** Carry out modification: rotate stair.

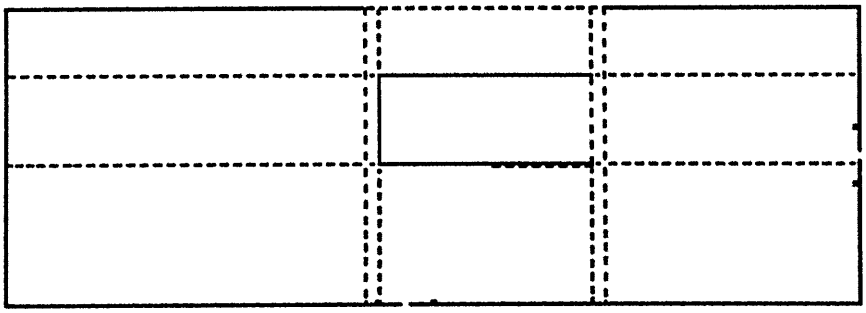


**Figure F.5:** Add projected edges for stair and walls.

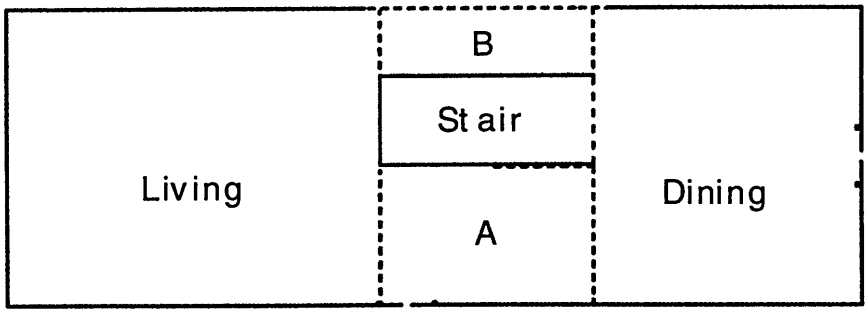



**Figure F.6:** Identify new territories by finding closed polygons in edge model and mapping them to territories in original design:

Stair territory edges have not changed identity, just their locations.
Living and Dining are now bounded by stair edges in new locations.
Territory A is again bounded by stair projected edges, but is larger.
Territory B is a new territory.

# Appendix G Details of Architecture Exercises

## G.1 Design: Chatham

Shown below are TAC's solutions to the first design problem described in Section 8.1, the problem of having one perceived main entry for the Chatham house.

TAC proposes these suggestions:

```
(or ;; increase Front-door pme until more than Side-door's
    (and move Front-door until ...
         maybe move exterior territories for Front-door
         decrease change in direction between Front-door and usual-approach
                  until ...
         maybe move exterior access territories for Front-door)
    ;; set Side-door pme to no-value
     (and replace Side-door with window
          remove exterior territories and paths for Side-door)
     (and replace Side-door with wall
          remove exterior territories and paths for Side-door)
     remove exterior paths between Side-door and usual approach
     fill <edge: 6.00 20.44...> along <edge: 10.11...>
     fill <edge: 6.00 22.89...> along <edge: 10.11...>
    ;; decrease Side-door pme until less than Front-door's
     (and move Side-door until ...
          maybe move exterior territories for Side-door
          increase change in direction between Side-door and usual-approach
                  until ...
          maybe move exterior access territories for Side-door)
     ;; increase Side-door pme until more than Front-door's
     increase solidity of Side-door to 1.0
     ;; set Front-door pme to no-value
     (and replace Front-door with window
          remove exterior territories and paths for Front-door)
     (and replace Front-door with wall
          remove exterior territories and paths for Front-door)
     remove exterior paths between Front-door and usual-approach
     fill <edge: 10.11 30.89...> along <edge: 25.00...>
     fill <edge: 22.33 30.89...> along <edge: 25.00...>
     ;; decrease Front-door pme until less than Side-door's
     decrease solidity of front door to 0.5)
```

It then creates one design per suggestion; the designs are shown in Figures G.1, G.2, and G.3.

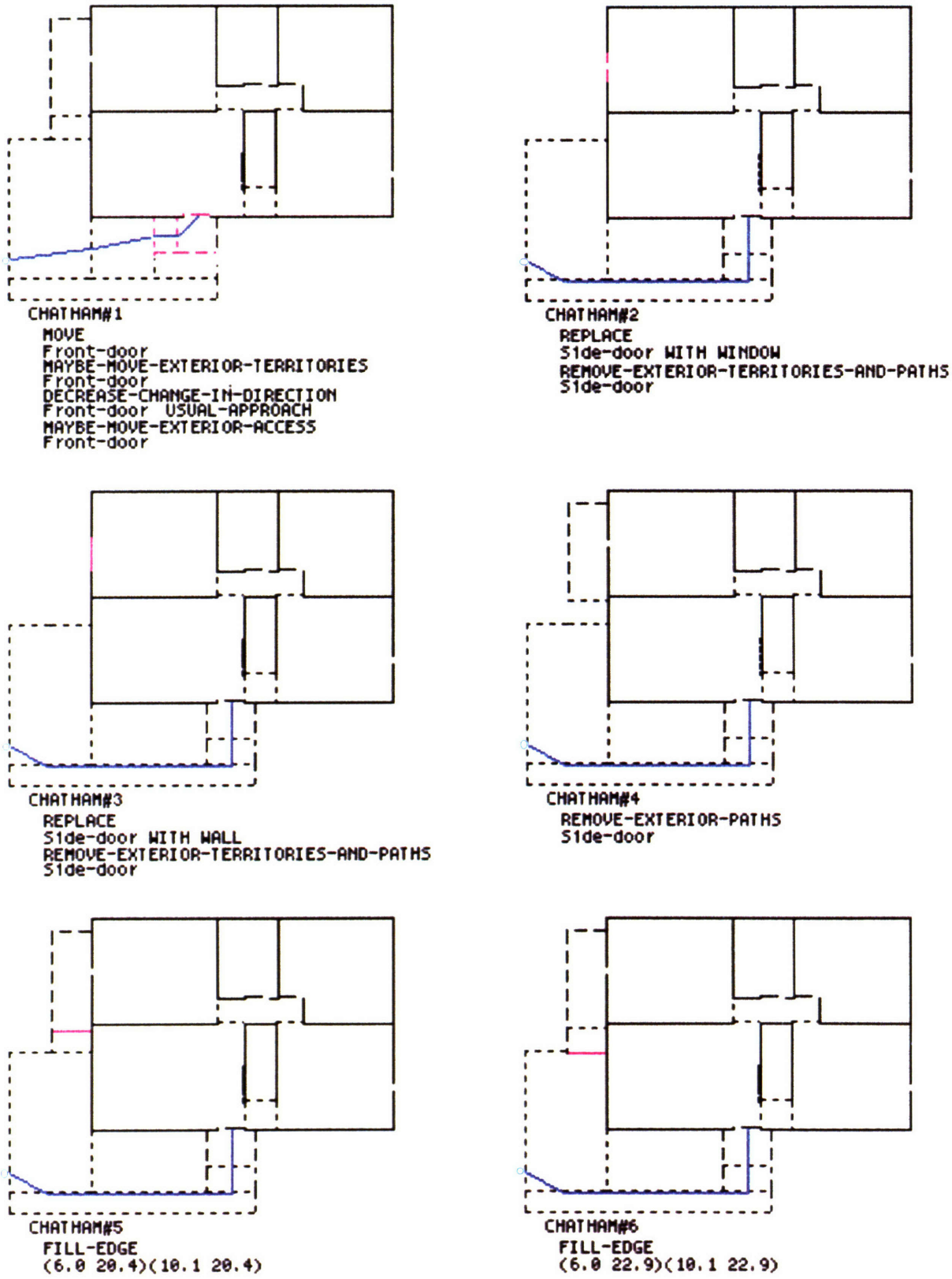



**Figure G.1:** Designs TAC proposes with front door as perceived main entry.

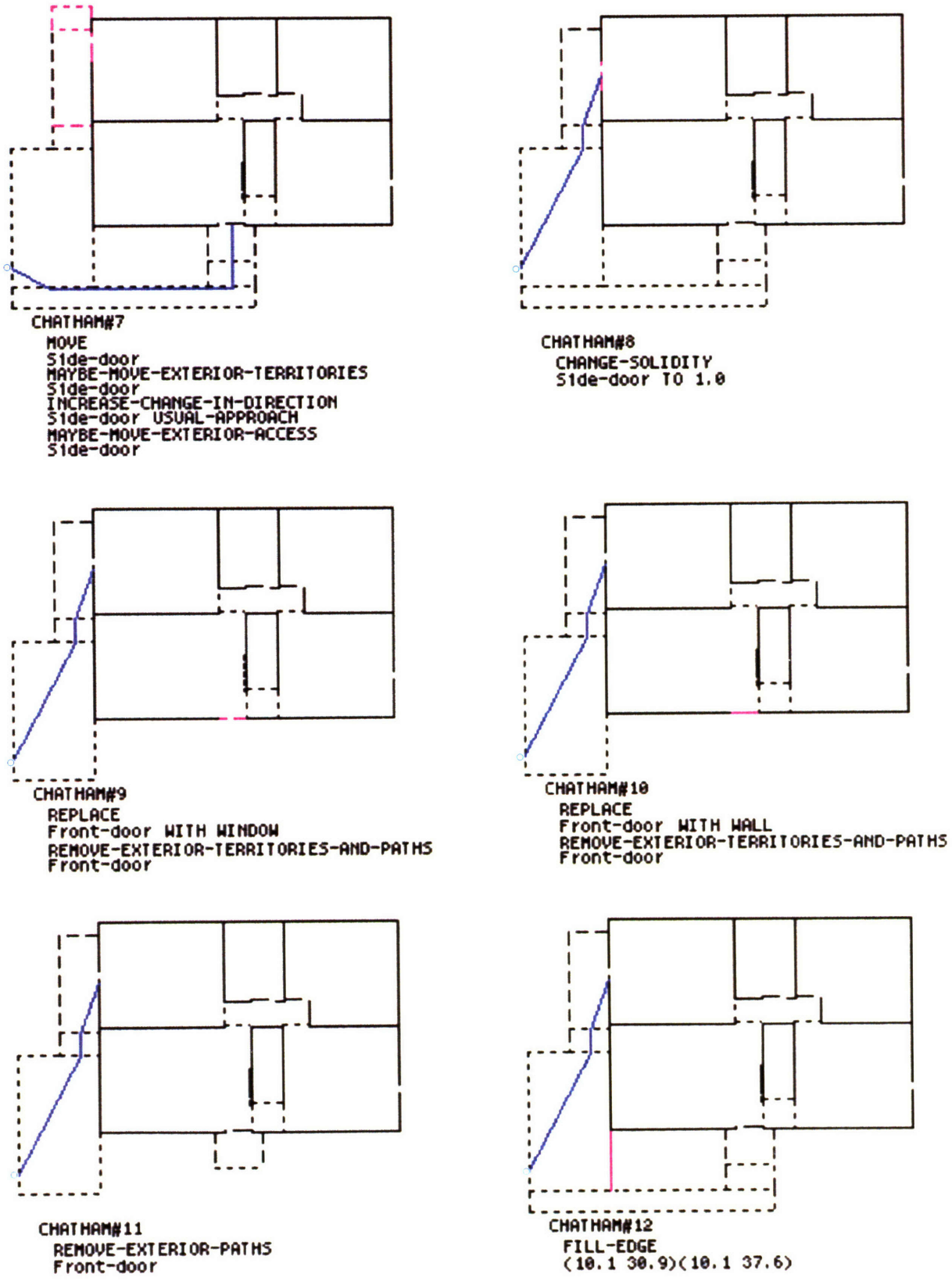



**Figure G.2:** Designs TAC proposes with front door or side door as perceived main entry.

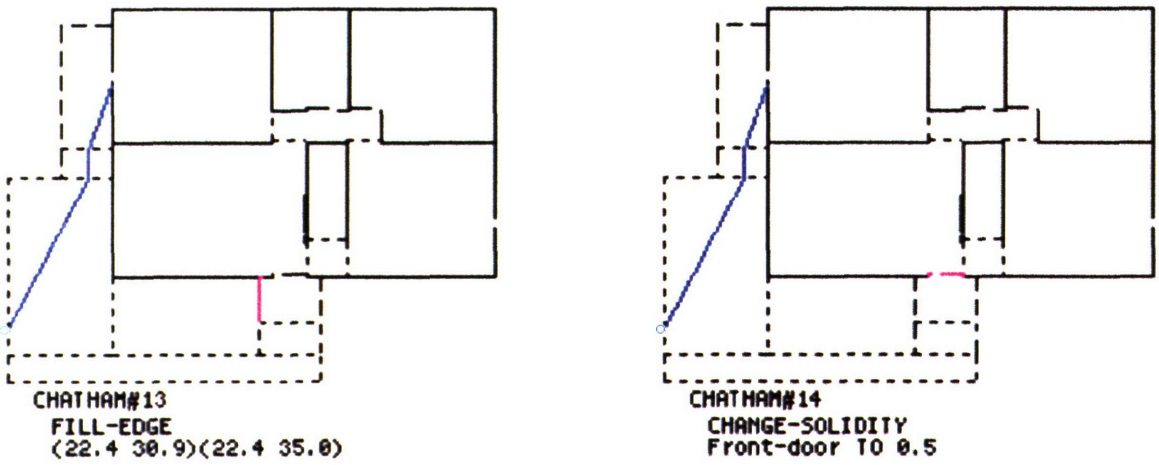



**Figure G.3:** Designs with side door as perceived main entry, cont'd.

## G.2 Analysis: Prairie Houses

We include here the counts of experiential and physical form characteristics for the designs in the Prairie house experiment. We also show how each experiential characteristic was defined using TAC's design characteristic definition language which was introduced in Section 3.2.

### G.2.1 Experimental Results

Shown below, and in Section 8.2, are the experiential and physical form characteristics used in Prairie house experiment. (Numbers to the right of each experiential quality are indexes for physical form characteristics that manifest the quality.)

Experiential qualities:

a. The design exhibits Wrightian group togetherness. (and 1 2 3)
b. The design exhibits home/hearth symbolism. (or 4 6)
c. The main living space is private. (and 7 (or 8 9 10 11))
d. The main living space is a place of refuge. (or 8 9 10 11)
e. The main living space is a place of prospect. (or 12 13)
f. An exterior space contiguous with the main living space is private. (and 14 15)

Details of physical form:

1. The design has a main living space that is the largest living space.
2. The design has a main living space containing a region from which all other living spaces are visible.
3. The design has a main living space connected to all other living spaces. (Two spaces are connected if they are no more than one space apart or if they have axially aligned doorways.)
4. The design has one fireplace location.
5. The design has a fireplace on an interior wall.
6. The design has a fireplace in the main living space.
7. The path from the front door to the private area does not pass within five feet of the center of the main living space.
8. The front door does not open into the main living space.
9. The front door and the main living space are on different levels.
10. The path from the front door to the main living space contains at least two changes in direction of greater than 15 degrees.
11. The path from the usual approach point to the main living space contains at least two changes in direction of greater than 15 degrees.
12. The main living space is elevated above the terrain.
13. An exterior living space at least 40% of the size of the main living space is contiguous with the main living space.
14. The front door does not open into the exterior space contiguous with the main living space.
15. The path from the usual approach point to the front door does not cross the exterior space contiguous with the main living space.

Characteristics present in the houses (indexes into the above lists shown):

Prairie houses

| | |
|---|---|
| Cheney | a-f; 1-8, 11-15 |
| Gale | a-f; 1-15 |
| Horner | a-f; 1-4, 6-15 |
| Roberts | a-f; 1-15 |
| Tomek | a-f; 1-15 |
| Willits | b-f; 1, 4-8, 10-15 |

These Prairie houses were used when testing characteristics (not in the experiment):

| | |
|---|---|
| Hickox | a-f; 1-8, 11-15 |
| Hunt | a-f; 1-9, 11-15 |
| Robie | a-f; 1-15 |
| Thomas | b-f; 1, 4-8, 10-15 |
| Walser | b-f; 1-4, 6-8, 10-15 |

Transition houses

| | |
|---|---|
| Emmond | b-f; 1-8, 11-15 |
| Furbeck | b-e; 1, 4, 5, 7, 8, 11, 12 |
| Wright | b-e; 1, 3, 4, 5, 7, 8, 12 |

Non-Prairie houses

| | |
|---|---|
| Colvin | a, b-e; 1-3, 6-8, 10-12 |
| Jones5A | b-f; 1, 3-8, 12, 15 |
| Lawson | b-d; 1, 4, 6-8 |
| Mallory | a-d; 1-3, 5-8, 11, 12 |
| Stickley | a-f; 1-4, 6-8, 12-15 |
| Winslow | b-e; 1, 4, 5, 7, 8, 12 |

### G.2.2 Design Characteristic Definitions

Shown below are TAC's definitions for the Prairie house experiential characteristics. Each characteristic has an evaluation function written in terms of physical form characteristics or other characteristics derived from physical form characteristics. (See Section 3.1 for description of defining evaluation functions for design characteristics.)

The Prairie house characteristics are represented by what we call design statements, which have the same components as design goals—an expression and a value—but which do not imply design intent. TAC evaluates design statements but does not suggest design modifications if the statements are not true. As with design goals, expressions in design statements are written in terms of TAC's goal specification language (see Section 3.2).

For each experiential characteristic below we show a design statement expression and the evaluation function body for the characteristic. We use "d" as a variable name for a design, "s" for a space (i.e. use space), "x" and "y" for any kind of design objects. We use "=>" to show replacement of a design characteristic name with its evaluation function body.

a. The design exhibits Wrightian group togetherness.

```
(shows-fllw-group-togetherness d) =>
   (is-fllw-social-center (main-living-space d))

   (main-living-space d) =>
       (space-with-activity d :main-living)

   (is-fllw-social-center s) =>
       (and (largest s (other-living-spaces s))
            (physically-connected s (other-living-spaces s))
            (visually-connected s (other-living-spaces s)))
```

b. The design exhibits home/hearth symbolism.

```
(shows-home-hearth-symbolism d) =>
   (or (has-one-fireplace-location d)
       (x-in-y * :any (elts-of-type 'fireplace d) (main-living-space d)))

   (has-one-fireplace-location d) =>
       (equal (location * any (elts-of-type 'fireplace d)))
```

c. The main living space is private.

```
(has-private-main-living-space d) =>
   (and (is-private-interior-space (main-living-space d) d)
        (is-hidden-place (main-living-space d) d)  [see is-hidden-place below]

   (is-private-interior-space s d) =>
       (private-wrt-path-btw s (main-entry d) (private-area-entry d)))

   (private-wrt-path-btw s x y) =>
       (gt (distance-btw (center s) (path-btw x y)) 5))
Note: main-entry is front door; private-area-entry is entrance to bedroom wing.
```

d. The main living space is a place of refuge.

```
(main-living-space-is-refuge-place d) =>
   (is-hidden-place (main-living-space d) d)

   (is-hidden-place s d) =>
       (or (on-different-levels s (main-entry d))
           (not (opens-into (main-entry d) s))
           (not (visible-from s (main-entry d) s))
           (circuitous-path-btw s (main-entry d))
           (circuitous-path-btw s 'usual-approach)))

   (circuitous-path-btw x y) =>
       (gte (number-of-x-degree-turns-btw x y 15) 2)
```

e. The main living space is a place of prospect.

```
(main-living-space-is-prospect-place d) =>
   (is-prospect-place (main-living-space d) d)

   (is-prospect-place s d) =>
       (or (on-different-levels s 'ground-level)
           (some '(and (relative-size * s .40)
                       (adjacent * s))
                 (exterior-spaces d)))
```

f. An exterior space contiguous with the main living space is private.

```
(has-private-exterior-space-adjacent-to-main-living d)

   (has-private-exterior-space-adjacent-to-main-living s d) =>
       (some '(and (is-private-exterior-space * d)
                   (adjacent * (main-living-space d)))
             (exterior-spaces d)))

   (is-private-exterior-space s d) =>
       (and (not (opens-into (main-entry d) s))
            (not (path-crosses s (path-btw (main-entry d) 'usual-approach))))
```

6597-100