Analogical and Inductive Reasoning in Architectural Design Computation

A Dissertation Submitted to
THE SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
For the Degree of Doctor of Technical Science

Presented by

Chen-Cheng Chen
M.S. in Architecture, Carnegie-Mellon University, 1985
Born in September 18, 1957, Taiwan, Republic of China

Diss. ETH No. 9487
Summer, 1991

Accepted on the Recommendation of
Professor Gerhard N. Schmitt, Referee
Professor Herbert E. Kramel, Co-Referee
Abstract

Computer-aided architectural design technology is now a crucial tool of modern architecture, from the viewpoint of higher productivity and better products. As technologies advance, the amount of information and knowledge that designers can apply to a project is constantly increasing. This requires development of more advanced knowledge acquisition technology to achieve higher functionality, flexibility, and efficient performance of the knowledge-based design systems in architecture.

Human designers do not solve design problems from scratch, they utilize previous problem solving episodes for similar design problems as a basis for developmental decision making. This observation leads to the starting point of this research: First, we can utilize past experience to solve a new problem by detecting the similarities between the past problem and the new problem. Second, we can identify constraints and general rules implied by those similarities and the similar parts of similar situations. That is, by applying analogical and inductive reasoning we can advance the problem solving process.

The main objective of this research is to establish the theory that (1) design process can be viewed as a learning process, (2) design innovation involves analogical and inductive reasoning, and (3) learning from a designer's previous design cases is necessary for the development of the next generation in a knowledge-based design system. This thesis draws upon results from several disciplines, including knowledge representation and machine learning in artificial intelligence, and knowledge acquisition in knowledge engineering, to investigate a potential design environment for future developments in computer-aided architectural design.

This thesis contains three parts which correspond to the different steps of this research. Part I, discusses three different ways - problem-solving, learning and creativity - of generating new thoughts based on old ones. In Part II, the problem statement of the thesis is made and a conceptual model of analogical and inductive reasoning in design is proposed. In Part III, three different methods of building design systems for solving an architectural design problem are compared - rule-based, example-based, and case-based. Finally, conclusions are made based on the current implementation of the work, and possible future extensions of this research are described. It reveals new approaches for knowledge acquisition, machine learning, and knowledge-based design systems in architecture.
Acknowledgements

I would like to express my sincere thanks to my family for their support during my graduate study. Especially, my father and my mother who taught me to be optimistic in life, encouraged me to follow my interests, and always think whatever I am doing is wonderful.

I am so fortunate to have a great committee, with Prof. Gerhard Schmitt as adviser, and Prof. Herbert Kramel and Prof. Boi Faltings as committee members. I appreciate for their academic guidance, intellectual communication, continued confidence, as well as their friendships.

I am grateful to Prof. Gerhard Schmitt for he led me through the entire study. He encouraged me to study in the Ph.D. program at CMU, and brought me to ETH to complete my study. He always had patience to listen to what I was thinking, what problems I had, and gave me the freedom to do whatever I thought. I learned not to be afraid to express my own ideas, and how to do scientific research. From him, I understood what is humble. He is not only my academic adviser, but also a friend of mine for the past seven years.

I am thankful to Prof. Herbert Kramel, for his support in the development of BAU. Without the input of his knowledge and experience, this research would not have been possible nor complete as it is now. He encouraged me to study at ETH, and helped me to re-think the role of CAD in architecture both form the C and the D sides. He taught me how to look at things not in a "either-or" view but in a "both-and" view, and told me that to be "lowkey" is not bad sometime.

Prof. Boi Faltings is the one who always gave me confidence whenever I reached a dead-end of the research. From him, I realized the fundamentals of the declarative-procedural controversy. I learned from him not only the principles of artificial intelligence, but also the spirit of western scientific thinking. His numerous comments also gave me profound understanding of case-based reasoning and machine learning.

I am also grateful to the following people.

Dr. Horace Flatt from the IBM Palo Alto Scientific center, for his support in the Joint Study on 'Expert Systems in Architecture' that took place at CMU, and the Joint Study on 'Graphical Knowledge Acquisition' that took place at ETH.

My previous committee members at CMU, Prof. Charles Eastman, Prof. Michael Rychener, and Prof. Robert Woodbury, for their discussions of the early work of this research. Prof Chin Pei, for directing my early efforts as an undergraduate, and recommending me to study at CMU.
Table of Contents

Chapter 1  Introduction .............................................................. 1
1.1  Problem Description ......................................................... 1
1.2  Motivation ................................................................. 2
1.3  Objectives ............................................................... 4
1.4  Scope of Work .............................................................. 4
1.5  Organization ............................................................... 5

PART I  PERSPECTIVES OF PREVIOUS WORK

Chapter 2  Computational Design Problem-Solving .......................... 8
2.1  Knowledge Engineering .................................................. 8
2.1.1  Knowledge-Based Design Systems (KBDS) ...................... 9
2.1.2  Knowledge Representation in KBS .................................. 10
2.1.3  Inference of Knowledge in KBS ...................................... 11
2.1.4  Knowledge Acquisition of KBS ....................................... 12
2.2  Recent Developments in CAAD .......................................... 13
2.2.1  Top-Down Refinement ............................................... 13
2.2.2  Bottom-Up Composition ............................................. 14
2.2.3  Summary of Recent Development in CAAD ...................... 15
2.3  Theoretical Background of Current Design Computation ............ 16
2.3.1  Herbert Simon: The Science of Design .......................... 17
2.3.2  Karl Popper: Philosophy of Science .............................. 19
2.3.3  Christopher Alexander: Synthesis of Form .................... 20
2.3.4  Conclusions ......................................................... 21
2.4  Example: Routine Design ............................................... 21
2.5  Summary of Design Computation Models ................................ 22

Chapter 3  Machine Learning .................................................... 23
3.1  Analogy ........................................................................ 24
3.1.1  Propositional Analogy .................................................. 24
3.1.2  Predictive Analogy ..................................................... 25
3.1.3  Metaphor ............................................................... 25
3.1.4  Structure-Mapping Theory ........................................... 26
3.2  Induction .................................................................... 27
3.2.1  Concept Formation .................................................... 27
3.2.2  Grammar Induction .................................................... 27
3.2.3  Concept Clustering .................................................... 28
3.2.4  Version Space .......................................................... 29
3.3  Media for Knowledge Acquisition ....................................... 29
3.3.1  Explanation as a Medium ............................................. 29
3.3.2  Similarities as a Medium ............................................ 30
3.3.3  Cases as a Medium ................................................. 30
3.3.4  Neural Nets ............................................................ 30

Chapter 4  Design and Creativity ............................................... 32
4.1  Some Characteristics of Architectural Design ......................... 32
4.2  The Other Two Classes of Design ...................................... 33
PART II
PROBLEM STATEMENT
AND A CONCEPTUAL FRAMEWORK

Chapter 5  Problem Statement and Requirements for Solutions ........................................ 37
  5.1  Limitations of Current Design Computation .................................................. 37
  5.1.1  Limitations of the Computer ................................................................. 37
  5.1.2  Limitations of Problem-Solving ............................................................. 38
  5.1.3  Limitations of Knowledge Representation ............................................... 40
  5.1.4  The Knowledge Acquisition Bottleneck .................................................... 40
  5.2  Requirements for Solutions ............................................................................. 43
     5.2.1  Knowledge Acquisition by Inductive Learning ...................................... 41
     5.2.2  Case-based Reasoning and Lateral Thinking ........................................... 41
     5.2.3  Graphical Knowledge Acquisition in Design ........................................... 42
     5.2.4  Summary ................................................................................................. 43

Chapter 6  Design, Learning, and Knowledge .............................................................. 45
  6.1  Learning in Architectural Design ................................................................. 45
  6.1.1  Analogical Learning in Architectural Design .............................................. 45
  6.1.2  Inductive Learning in Architectural Design .............................................. 46
  6.2  Declarative-Procedural Controversy of Knowledge ....................................... 47
  6.3  Types of Knowledge ....................................................................................... 48
  6.4  Knowledge Compilation and Knowledge Acquisition .................................... 50
  6.5  Knowledge in Learning Systems .................................................................... 51
  6.6  Causal Inversion and Knowledge Acquisition in Design .............................. 52

PART III
IMPLEMENTATIONS

Chapter 7  BAU: A KBDS for the investigation of A Basic Architectural Unit ............ 56
  7.1  Introduction ..................................................................................................... 56
  7.2  Methodology ................................................................................................. 58
     7.2.1  Prefabrication and Standardization ......................................................... 59
  7.2.2  The Basic Architectural Units ................................................................. 59
  7.2.3  BAU: System Overview ............................................................................ 60
  7.3  Development of BAU ..................................................................................... 61
     7.3.1  Subtask I (BAU.I) ................................................................................... 61
           7.3.1.1  Rules for Internal Connections ......................................................... 62
           7.3.1.2  Rules for External Orientations ....................................................... 63
           7.3.1.3  Ranking Rule of BAU.I ................................................................. 64
           7.3.1.4  Results of BAU.I ........................................................................... 64
     7.3.2  Subtask II (BAU.II) ................................................................................ 66
           7.3.2.1  Rules of BAU.II .............................................................................. 67
     7.3.2.2  Results of BAU.II .............................................................................. 68
     7.3.3  Subtask III (BAU.III) ............................................................................... 69
           7.3.3.1  Rules of BAU.III .............................................................................. 70
           7.3.3.2  Results of BAU.III ........................................................................... 70
           7.3.3.3  Different Levels of Abstraction ....................................................... 72
     7.3.4  Current Development of BAU (BAU.IV) ................................................. 72
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>Experiences of BAU</td>
<td>73</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Knowledge Representation</td>
<td>73</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Control Strategy</td>
<td>74</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Problem Space and Solution Sets</td>
<td>75</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Software and Hardware</td>
<td>75</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Project History</td>
<td>76</td>
</tr>
<tr>
<td>7.5</td>
<td>BAU and CAM</td>
<td>76</td>
</tr>
<tr>
<td>7.6</td>
<td>Summary of BAU</td>
<td>77</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Inductive Reasoning in BAU</td>
<td>79</td>
</tr>
<tr>
<td>8.1</td>
<td>Test Case: Acquiring the Rules of BAU I</td>
<td>79</td>
</tr>
<tr>
<td>8.2</td>
<td>Conjunctive and Disjunctive Generalizations</td>
<td>80</td>
</tr>
<tr>
<td>8.3</td>
<td>Acquiring the Rules of BAU II</td>
<td>81</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Knowledge Representation of EB-BAU II</td>
<td>82</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Knowledge Acquisition in EB-BAU II</td>
<td>84</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Rules Derived by EB-BAU II</td>
<td>84</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Knowledge Application in EB-BAU II</td>
<td>85</td>
</tr>
<tr>
<td>8.4</td>
<td>Summary of Inductive Reasoning</td>
<td>90</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>Analogical Reasoning in BAU</td>
<td>91</td>
</tr>
<tr>
<td>9.1</td>
<td>Adaptations of Previous Design Solutions</td>
<td>91</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Null adaptation</td>
<td>92</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Case-Based Reasoning in BAU</td>
<td>93</td>
</tr>
<tr>
<td>9.1.2.1</td>
<td>Comparing RB-BAU and CB-BAU</td>
<td>94</td>
</tr>
<tr>
<td>9.1.2.2</td>
<td>CB-BAU: System Overview</td>
<td>95</td>
</tr>
<tr>
<td>9.1.2.3</td>
<td>Results of CB-BAU</td>
<td>97</td>
</tr>
<tr>
<td>9.1.2.4</td>
<td>Limitations of the Current CB-BAU</td>
<td>110</td>
</tr>
<tr>
<td>9.2</td>
<td>Adaptations of Previous Design Problem Solving Process</td>
<td>110</td>
</tr>
<tr>
<td>9.3</td>
<td>Summary of Analogical Reasoning</td>
<td>111</td>
</tr>
<tr>
<td>Chapter 10</td>
<td>Conclusions and Future Work</td>
<td>113</td>
</tr>
<tr>
<td>10.1</td>
<td>Conclusions</td>
<td>113</td>
</tr>
<tr>
<td>10.2</td>
<td>Parting Reflections</td>
<td>115</td>
</tr>
<tr>
<td>10.3</td>
<td>Future Work</td>
<td>116</td>
</tr>
<tr>
<td>Appendix I</td>
<td>Source Code of EB-BAU I</td>
<td>118</td>
</tr>
<tr>
<td>Appendix II</td>
<td>Number of Solutions Generated by Rule-RF4</td>
<td>120</td>
</tr>
<tr>
<td>Appendix III</td>
<td>Source Code of CB-BAU</td>
<td>121</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>137</td>
</tr>
</tbody>
</table>
List of Tables

Table 7-1: Rules for describing the BAU environment. 62
Table 7-2: Rules for internal connections. 63
Table 7-3: Rules for external orientations. 65
Table 7-4: Ranking rule of BAU1. 64
Table 7-5: Design rules for roofs. 67
Table 7-6: Design rules for adjacent walls. 70
Table 7-7: Knowledge representation of the basic architectural unit. 73
Table 7-8: Problem space generated by BAU1. 75
Table 8-1: Comparisons of different sample space and solution sets. 85
Table 9-1: A case-frame representation of a BAU case. 96
Table 9-2: Sample scores and weighting factors from a comparison of source and target case frames. 97
Table 9-3: Summary of CB-BAU's sample runs. 98
Chapter 1

Introduction

Two observations on modes of reasoning are the starting point of this research. Analogical reasoning which refers to how we utilize the experience of past design problems to solve new design problems by detecting the similarity between the past problem and new problem. Inductive reasoning which refers to how we acquire or learn design constraints and design rules by detecting the similarities between different designs and by identifying the similarities between different design contexts. In the following sections, the problem statement, the objectives, and the motivation for this research are described. Then, the scope of the work and related research areas are defined. Finally, the organization of this thesis is described.

1.1 Problem Description

One of the most challenging problems of building a knowledge-based design system is the transfer of a designer's expertise into the system. Present knowledge acquisition techniques are cumbersome and often require knowledge engineers to act as human translators between the knowledge source and the program. Besides, most knowledge-based design systems generate design solutions from scratch, thus they do not take advantage of existing partial design solutions. To overcome the deficiencies of this approach, future design systems should acquire design knowledge from partial design solutions directly, and apply the knowledge to similar but unsolved problems.

To develop such a system, two problems must be addressed: First, given a design problem, the issue arises of how to find and use solutions of previous problems which are known to be similar to the current problem. Secondly, once a solution for a specific design problem has been found, the question of how to utilize this solution for solving a class of similar
design problems appears. Learning by analogy and learning by induction are the proposed responses to those two problems.

1.2 Motivation

The main objective of this research is to establish the theory that (1) design process can be viewed as a learning process, (2) design innovation involves analogical and inductive reasoning, and (3) learning from a designer's previous cases is necessary for the development of the next generation in a knowledge-based design system. In particular, the following propositions support the objective more specifically.

Investigating analogical and inductive learning in architectural design.
The architectural design process involves learning about the context of a design problem and ways to solve that problem. Human designers deliberately learn about a design by doing several rough design schemata. What they learn is the process which will enable them to develop a better design schema. It is quite common within the design process that a designer looks for previously solved design units or design rules that could apply to the problem at hand, identifies the unit, extracts it, and modifies it. While copying and editing are a trivial kind of analogy, less trivial analogies would enhance the capabilities of design knowledge acquisition. Therefore, we can associate an architectural design process with a learning process, and anticipate that future design knowledge-based systems should contain a certain level of learning capability. Such problem-specific learning is a powerful technique for focusing and thus reducing the exhaustive search efforts throughout a design process.

Building computer-aided design systems with learning capabilities.
Learning by observing an experienced designer is one effective technique for acquiring design knowledge. By using analogical and inductive mechanisms, we can learn how designers actually develop designs. While most traditional computational models merely rely on built-in deductive reasoning mechanisms, they do not reflect the essential inductive mechanism of design thinking. An ideal environment with appropriate graphics and monitoring capabilities would monitor the operation of designers' behavior and record the design process, which is important both for the design discipline and man-machine interaction. Another advantage of the proposed system is that, rather than
maintaining an oversized data base, it segregates and extracts only the necessary information.

**Applying dynamically changing knowledge bases to knowledge-based design systems.** Standard knowledge representation techniques in artificial intelligence, such as frames, scripts, objects, and so on, represent static memory structures. Dynamically changing knowledge bases\(^1\), which acquire new knowledge during the processes of understanding and problem solving, more closely resemble the human design knowledge representation. For instance, contrasting a source frame with a destination frame, may generate a transfer frame synchronously. Furthermore, although traditional knowledge engineering relies on a rule-based approach, a case-based approach may be more appropriate for guiding search in a dynamically changing design process.

**Using adaptation to solve a design problem which contains incomplete or uncertain knowledge.** If a problem-solver knows enough about the problem he faces, and the required methods for solving the problem, then the domain of the problem is closed - the problem-solver can predict the outcomes of his actions. Generally, problem-solving in a closed domain, a strategy can be created for solving the problem, then carried out with assurance of success. However, real problems consist of situations which are unique and sometimes even contradictory. There may be only one experience that is relevant to a given problem, and sometimes so many, that each leads to such different results, that no path is obvious. Problem solving in such open domains, where the problem-solver has to deal with incomplete or uncertain knowledge is more complex. In order to cope with open domains, the initial task of the problem-solver is to identify the differences and similarities between the previous situation and the new one in question, and to then adapt the previous situation to the new problem.

**Extending routine design work to innovative design.** Standardized and one of a kind designs are the two extremes of design productions types. For example, you may spend fifty Swiss Francs to purchase a SWATCH,

---

\(^1\) For example, MOPS-based (Memory Organization Packets) memory techniques (Schank 82), which is based on script representation but emphasizes dynamic memory.
or you may spend fifty thousand Swiss Francs to purchase a luxurious handmade watch; although both watches serve the same function of telling you what time it is. The computer is already a powerful tool for helping in the design of standardized productions; however at the other end, that of custom design, masters of design develop creative design productions relatively independent of computer aid. There should be possibilities for using the computer to fill the gap between those two types of design problems, i.e., by maintaining and updating the design knowledge base when new design requirements are introduced into a standardized design. In such a way, standardized design could be extended into innovative design, which is related to analogical and inductive reasoning.

1.3 Objectives

The purpose of this research is to achieve a better understanding of the mapping between human design reasoning and machine information processing. The ultimate goal is not only to make machine and computational processes more efficient for use in architectural design processes but also to gain a better understanding of human intelligence. The studies of human and machine learning complement each other. In particular, the following issues are addressed:

1. Investigation of architectural design reasoning through analogy and induction.
2. Classification of human design knowledge acquisition and development of the appropriate representations.
3. Acquisition and representation of graphic and non-graphic design objects, and of knowledge about these objects in a uniform and easily manipulatable form.

This study explores architectural design problems in an interdisciplinary manner. It reveals new approaches for knowledge acquisition, machine learning, and knowledge-based design systems in architecture.

1.4 Scope of Work

This research incorporates knowledge engineering and machine learning techniques in artificial intelligence to investigate a design environment for future computer-aided architectural design systems.
Design processes and design thinking are discussed with respect to traditional artificial intelligence paradigms as well as recent research results in machine learning. Knowledge representation and knowledge acquisition for the architectural design discipline are investigated based on knowledge engineering technology. As a result, this research intends to develop a more user-friendly man-machine interaction for computer-aided architectural design to support designers in exploring design alternatives with the computers, which so far has been attempted using traditional modes of interaction.

1.5 Organization

This thesis contains ten chapters grouped into three major parts which correspond to the different steps of this research. Part I, *Perspectives of Previous Work*, describes three different ways to generate new thoughts based on old ones. Part I contains three chapters. Chapter 2 summarizes the problem-solving methods that have been used in CAAD and knowledge-based design systems. Chapter 3 describes the utilization of machine learning techniques in knowledge acquisition processes for building expert systems. Chapter 4 examines the fundamental characteristics of architectural design, and various classes of design other than routine design, which is discussed in Chapter 2.

In Part II, *Problem Statement and a Conceptual framework*, the problem statement of this thesis is made and a conceptual framework of analogical and inductive reasoning in design is proposed. Chapter 5, examines the fundamental limitations and problems of current design computation and the requirements for solutions. Chapter 6, explores the kinds of knowledge and learning involved in design.

In Part III, *Implementations*, different approaches of building knowledge-based design systems for solving an architectural problem are compared. Chapter 7 presents a design system called BAU, which is implemented with a traditional rule-based approach. The purpose of the development of BAU is to help the architect investigate a basic architectural unit. In Chapter 8, an example-based reasoning system EB-BAU is developed, which demonstrates how part of the design rules of BAU can be derived based on inductive learning mechanisms. Chapter 9 emphasizes the reuse of design solutions and the design problem solving
strategies of BAU for new and similar design problems. Additionally, a case-based design system, CB-BAU is introduced, which demonstrates another method of building design systems. The underlying mechanisms of CB-BAU are based on analogical and inductive reasoning. Finally, conclusions are made based on the current implementations; future extensions of this research are also discussed.
PART I

PERSPECTIVES OF PREVIOUS WORK

The common topic in this part is the generation of new thoughts from old, by the intelligent development of representations already in the mind. In this part we will see how the topic underlies and connects computational changes variously attributed to problem solving, learning, and creativity. Problem solving is seen as grouping or arranging a finite number of basic information processing mechanisms into strategies that allow complex problems to be solved [Newell 72]. Learning is the improvement of cognitive capacities under outside influence (e.g., teachers) [Simon 82]. Creativity retrieves knowledge that is not routinely applied to a situation, and uses it in a new way [Schank 89].

It is evident that these phenomena are closely connected [Boden 77]. General learning may derive from specific experiences, and may involve creative thinking; “spontaneous” construction of new representations may be elicited by a particular need to solve a problem, and may be aided by environmental cues; and solving a specific problem may require creativity and lead to general learning. So the distinctions should be regarded as a matter of emphasis rather than an expression of rigid partitions within mental reality.
Chapter 2

Computational Design Problem-Solving

There are two major streams for developing tools in architectural design computation. The first one is to give the system powerful problem solving capabilities. This can be achieved by developing a dedicated knowledge-based design systems (KBDS) for designing within narrow problem domains. The second approach is to put emphasis on understanding the designer's intention in order to enhance his creativity. The concept of computer-aided architectural design (CAAD) can be classified in the second category. The scope of this chapter is limited to generative design computation models, excluding analytical design computation models which provide information on the performance of the design along one or more dimensions. Current research in design computation models assumes that design is a uniform and homogeneous process, which is specific to a well-defined class of design problems called routine design [Brown 89]. We will discuss routine design and the current theoretical background of design computation at the end of this chapter.

2.1 Knowledge Engineering

Traditional artificial intelligence (AI) focused on the construction of general purpose intelligent systems. However, the more classes of problems a system could handle, the more poorly it seemed to do on any individual problem. This realization led to the development of special-purpose computer systems called expert systems. In contrast to conventional AI systems, expert systems generally represent domain specific knowledge in the form of rules or facts that are explicit to a knowledge base that is separated from the inference engine. Therefore, expert systems are highly domain specific, they know much about a narrow range of knowledge rather than something about everything.

Expert systems can be grouped into two major categories according to the kinds of problems they address [Sowa 83]:

8
• **Classification.** The diagnosis of diseases, interpretation of geological surveys, and analysis of error reports require a large amount of data to be classified in one or more categories. Since many interacting events can obscure the data, classification systems typically use statistics or fuzzy logic to give a range of possibilities rather than a yes/no answer.

• **Design.** These systems search for a combination of structures to satisfy a particular goal. They may design VLSI chips, select components for a computer configuration, or suggest gene-splicing experiments in molecular genetics. Unlike classification systems that apply fuzzy logic to a list of symptoms, design systems use exact reasoning on highly structured data.

Although these two categories are extremely diverse, they have three common features: first, there exist recognized human experts in these fields; second, the knowledge that the experts have is quantifiable; third, the knowledge can be expressed in declarative rules instead of procedures.

One of the best known examples of an expert system is MYCIN. It is a classification system that uses surface rules for diagnosing bacterial infection [Buchanan 84]. To a knowledge engineer, the MYCIN knowledge base is a set of facts together with a set of production rules that drives the inference engine. Each production rule has one or more premises that are to be matched to the facts and conclusions that are added to the facts if all the premises match.

### 2.1.1 Knowledge-Based Design Systems (KBDS)

The purpose of a design system is to assemble a design description that meets certain specifications [Rychener 88]. While it does the assembly, it must observe constraints on the possible way that the components in a design fit together. A typical design system is XCON (R1) for designing computer configurations. It starts with a customer’s order for a VAX computer system and checks it for consistency [McDermott 80]. It ensures that required control units, cable and power supplies are present in the order and prevents mutually exclusive or redundant options from being ordered. After XCON has determined what features are needed, it prints
a complete list of components and draws a diagram of the final configuration.

An expert system for the configuration design of 3-dimensional high rise building structures has been developed by Maher [Maher 85] in a program called HI-RISE. The program generates different structural configurations, made up of standard structural subsystems, based on user defined constraints of size and load. The rules guiding the generation are based on basic or heuristic principle knowledge.

Mittal et al. [Mittal 86] have developed a computer program called PRIDE for the design of paper transport mechanisms in copiers. The program uses a knowledge-base to generate, evaluate, and re-design configurations of rollers to guide the paper along a smooth path based on design constraints.

For assistance in the configuration design of integrated circuits, a program called VEXED [Steinberg 87] has been developed. The program uses an interactive rule-based system to help the human designer develop the layout, or configuration, of an integrated circuit. An important feature that of this system has is the ability to backtrack to any previous point in the design. This is useful if the current line of reasoning proves to be undesirable.

2.1.2 Knowledge Representation in KBS

A production system was first introduced by Post in 1943, and it r-emerged in the context of natural language processing in the form of rewrite rules by Chomsky in 1957. Production systems were proposed for modeling human problem-solving behavior by Newell and Simon in 1972 [Newell 72]. Today, the intuitive structure for representing knowledge has made production systems the most popular base for expert systems, including the pioneering programs MYCIN and XCON. Production systems typically include three major components:

1. A global data base that represents facts and assertions about the problem.
2. A set of rules that constitute the program. Each rule has a condition part and an action part. The condition part describes the
data configuration for which the rule is appropriate. The action part supplies instructions for changing the data configuration.

3. An inference engine to execute the rules. The inference engine determines which rules are relevant to a given data memory and chooses one to apply.

Production systems use data-sensitive unordered rules rather than the sequential instructions typical of traditional programming techniques as the basic unit of computation. A production system is appropriate when the knowledge to be encoded occurs naturally in rule form, when a program's control is complex, or when a program is expected to be modified significantly.

2.1.3 Inference of Knowledge in KBS

There are two important ways in which rules can be used in production systems; one is called forward-chaining and the other backward-chaining [Nilsson 80]. Forward-chaining systems progress from the given information to a goal. Starting from what is initially known, the current state of knowledge is used to make a chain of inferences until either a goal is reached, or a solution is shown to be unattainable. The inference engine matches the left-hand sides of rules against data memory, and executes the right-hand-sides of rules to update the knowledge base by making changes to data memory. By contrast, backward-chaining systems work in the opposite direction. Starting from the overall goal, they break down goals into simpler subgoals until the result is a collection of goals. In a backward-chaining architecture, the inference engine matches the right-hand sides of rules against the currently unattained goals. The solution to the directly achievable goals are then merged in appropriate ways to construct an overall solution.

In forward-chaining systems, the selection or control strategy is often called conflict resolution. In backward-chaining systems conflict resolution does not occur. A backward-chaining system sets off all valid rules and combines them using certainty factors.

Forward-chaining systems are most appropriate when there are many equally acceptable goal states and a single initial state. XCON is a typical forward-chaining system, and it is implemented with its inference engine of the OPS5 language [Brownston 85]. Backward-chaining
systems, by contrast, are most appropriate for tasks such as diagnosis, in which there is a single goal state and much potentially relevant initial information. MYCIN is a typical backward-chaining system, and it is implemented with its inference engine EMYCIN - the first expert system shell [Buchanan 84].

2.1.4 Knowledge Acquisition of KBS

Knowledge acquisition involves the integration, organization, representation, and abstraction of data and information so that it can be used for problem solving, decision making, and other cognitive functions we associate with learning and thinking. Knowledge acquisition can occur at two levels. The lower level is concerned with structuring facts in a knowledge base. The higher level is concerned with relating information to previously stored information [Buchanan 83]. This higher level more closely resembles the function we call learning.

In knowledge engineering, knowledge acquisition is the process of mapping domain knowledge onto some sort of knowledge base [Boose 89]. This mapping task can be roughly divided into two subtasks. The first is the acquisition of the domain knowledge with the goal of understanding the domain properly. This means, being able to identify, explicate, and structure the relevant facts, relationships, and heuristics. The second subtask is the mapping of conceptional and finally implementable constructs.

Traditional knowledge acquisition relies on knowledge engineers extracting domain knowledge from domain experts, typically a tedious and time-consuming process. Techniques for automating the knowledge acquisition process are important for future development of knowledge-based systems. There are three ways in which knowledge acquisition might be automated: First, special editing tools that allow domain experts to process domain knowledge that interacts directly with the knowledge bases. Second, natural language processing techniques that allow experts to instruct computer systems through conversations. Third, knowledge-based systems with learning abilities which might acquire knowledge directly from experience in their domain [Hayes 83]. This project focuses on knowledge acquisition and learning, and we will emphasize on the discussions of machine learning in next chapter.
2.2 Recent Developments in CAAD

This section describes the main approaches in the computer-aided architectural design (CAAD) and related engineering fields. Most CAAD systems currently in use are intended for modelling three-dimensional design objects in order to analyze them visually or computationally [Mitchell 77, Schmitt 88]. Top-down refinement and bottom-up composition are the most common CAAD models\(^2\). The first is the generation of non-standard forms from scratch, followed by their redesign in order to fulfil a specified function. The second branch being explored is the assembly of standard parts or components to perform a specific task or function.

2.2.1 Top-Down Refinement

The idea of top-down refinement is to start with the highly abstract initial problem specifications and to refine them by adding details and by decomposing them to the point at which primitive operators are available to complete all the subtasks defined [Dixon 89]. Top-down refinement depends particularly on the identification of suitable abstraction, problem decomposition, and the management of conflicts which arise during the design process. Mostly, this process is executed using iterative steps.

It is not always possible to decompose a design problem into completely independent sets of subproblems. When these subproblems interact, conflicts will inevitably occur. For example, the input for one subproblem may depend on the output of another subproblem. For top-down refinement, the first step is to seek partitions with a minimum of interaction. When important interactions do occur, conflicts often are avoided by constraint propagation.

Due to faulty prior decisions or additional details not considered at higher abstraction levels, conflicts cannot always be avoided. In this case, modification and backtracking are used to resolve conflicts. Modification makes changes at the same level of detail and typically applies a hill-climbing approach. Backtracking involves returning to an earlier refinement state and then attempting to continue with different and more successful refinement. Unfortunately, there is no guarantee that all

\(^2\) In natural language processing and computer vision, different ways of information processing are made according to top-down and bottom-up.
conflicts can be resolved by the methods mentioned above. Sometimes, it is necessary to go back to the initial specification and re-define it, or even admit failure.

2.2.2 Bottom-Up Composition

The idea of bottom-up composition is to explore what can be constructed using available components. While top-down refinement starts from problem specification and works toward basic solution components, bottom-up composition does the opposite. For a simple problem, it may involve exhaustive enumeration of possible alternatives. For a more complicated problem, it requires guidance to avoid the exponential explosion of possible alternatives.

Bottom-up composition in CAAD is exemplified in the theory of Shape Grammars\(^3\) [Stiny 80]. In the mid 1970s, Stiny and Gips devised grammars that composed patterns of lines, called shape grammars, for the recursive generation of shapes for visual purposes. As Gips [Gips 74] states in his book:

"Shape grammars are similar to phrase structure grammars, which were developed by Chomsky in Linguistics. Where a phrase structure grammar is defined over an alphabet of symbols and generates a language of sequences of symbols, a shape grammar is defined over an alphabet of shapes and generates a languages of shapes."

The use of shape grammars in architectural design is predicated on the analogy between language and design. In the same way that we have rules governing how words may be strung together to form a sentence, there are rules governing how a vocabulary of design elements can be combined to form design. Some recent research work on shape grammars is implemented in production systems [Gips 80]. The types of architecturally important grammars that can be defined include floor plan layout grammars, furniture arrangement grammars, elevation layout grammars, and site layout grammars [Flemming 88]. Once a grammar has been defined, a computer can be employed to generate form in the corresponding languages. The task may either be to enumerate

\(^3\) In this survey, we only take the restricted definition of shape grammars, which does not include parameterized shape grammars.
forms within the languages, or to produce particular forms in the language that have desirable properties or restrictions. Traditional shape grammar research relies on humans discovering the grammar or patterns of existing building layouts.

Fitzhorn [Fitzhorn 89] shows that a variant of graph grammar can produce three-dimensional solids. He creates three grammars, one of which generates the constructive solid geometry representation, the second of which generates the boundary representation, and the third of which generates plan models.

2.2.3 Summary of Recent Development in CAAD

For top-down refinement tasks, the abstract refinement process may be precompiled into a standard structure of parameterized components, and the design process may consist of reasoning about constraints in order to determine appropriate parameter values. Inverse reasoning is applied for the top-down approach, the designer starts by recognizing the overall function or form of the device and, thereby, from a hypothesis which can be tested by examination of component parts and their functional relationships. Bottom-up composition converts a specification into an implementation via a sequence of transformations from one complete description to another. Forward reasoning is applied for the bottom-up approach, the designer starts by recognizing one or more components, rather than the entire device, and proceeds by reasoning about the formal or functional relationships of the components recognized with those of other components.

Current CAAD models very much rely on solving problems using constraints, since one of the goals of a design process is to produce a detailed structural description of an artifact within explicit and implicit constraints on functionality, structure and manufacturability. Constraints are information that limit design variables to a specific set of permissible values. This set of values may be either qualitative or quantitative, continuous or discrete. In either case, designers use constraints to help them generate concepts and refine these concepts into a completely defined, detailed solution [Sussmann 80].
Top-down refinement does not handle goal coupling well, systems based on abstract refinement typically use constraint propagation to achieve consistency between different parts of the design [Gosling 83]. Constraint propagation is the creation of new constraints from formulated constraints. Bottom-up composition is achieved by adding a new design element which satisfies existing constraints of an existing partial design solution. Constraint satisfaction is the operation of finding values for variables so that the constructions on these variables are satisfied [Stauffer 89].

Sometimes, the complete set of constraints can be expressed formally in terms of design parameters. The evaluating criteria can be expressed by an objective function, after which optimization can be applied to the design problem. The optimization problem in design is to find an admissible set of values of the variables, compatible with the constraints, that maximize or minimize the utility function for the given values of the environment parameters [Radford 88].

Constraint systems have a long history in computer science. Ivan Sutherland’s SKETCHPAD [Sutherland 63] was a novel drawing system that allowed the definition of arbitrary objects and constraints. It pioneered the use of interactive computer graphics and constraint systems. Alan Borning’s THINGLAB system [Borning 79] carried on where SKETCHPAD left off. It was a generalized simulation laboratory based on constraints. Mark Gross describes the Constraint Explorer, a constraint-based system for architectural design [Gross 85]. He emphasizes the constraint-based approach as a theory of design.

2.3 Theoretical Background of Current Design Computation

Recent models of design computation have strong connections with several design theories developed in the 1960s [Cross 84]. Since the beginning of the design methods or system design movement, science has provided design researchers with their prime model for theory and research method. A recurring theme in design theory has been a desire to relate design method to scientific method - to create the 'science of design' or a 'design science' [Jacques 81]. Three important theories related to the science of design are discussed in this section: Herbert Simon’s The Sciences of Design [Simon 82], Karl Popper's philosophy of
science [Popper 72], and Christopher Alexander's Notes on the Synthesis of Form [Alexander 79]. Finally, relationships between routine design and current design computation will be discussed.

2.3.1 Herbert Simon: The Science of Design

The basic text on which is founded the faith of 'the science of design' appears to be Herbert Simon's The Sciences of the Artificial [Simon 82]. In the beginning of his text Simon attempts to make a distinction between science and design: "The natural sciences are concerned with how things are, whereas design, on the other hand, is concerned with how things ought to be." Simon went on to outline a series of elements that would embody 'the science of design'. The examples offered by Simon were methods of optimization from management science, and methods of problem structuring based on the hierarchical decomposition techniques developed by Alexander.

Simon states that the design process is organized hierarchically and that a design problem is nearly decomposable. A hierarchical decomposition process is normal wherever design processes are necessarily formalized and externalized, Simon claims. Such a process offers a sequence of well-structured transformations which enable a problem-solver of limited capabilities to solve a variety of problems. During any given period of time, the architect will find himself working on a problem which, perhaps beginning in an ill-structured state, then converts itself through evocation from memory into a well-structured problem. He therefore concludes that there is no reason to suppose that other different problem-solving abilities or techniques are needed to tackle ill-structured problems. The main distinction between ill-structured and well-structured problems may be nothing more than the respective size of the knowledge base [Simon 73].

Clancey introduced two hierarchies, one for application problems and one for problem-solving methods [Clancey 86]. Broadly, the problem hierarchy divides into analysis and synthesis tasks. Generally, analysis tasks involve identifying sets of objects based on their features. Synthesis tasks require that a solution is built up from component pieces or subproblem solutions. High-level application problems include identification, prediction, control, design, specification, and modification assembly.
Identification is further broken down into diagnosis and monitoring; design is broken down into configuration and planning. Problem-solving methods described by Clancey include heuristic classification and heuristic construction. Relationships exist between design problems and these methods. For instance, the heuristic classification problem-solving method has been used for knowledge-based systems that solve analysis problems and is employed in a variety of knowledge-based system development tools (M.I., EMYCIN). General methods that have been applied to synthesis problems are sparse. Clancey classifies these methods under heuristic construction. However, it may be difficult to generalize the method.

Newell and Simon's information processing theory has become a popular model for explaining the reasoning ability of the human mind, especially in cognitive science [Newell 72]. The "General Problem Solver" program of Newell and Simon is also a landmark in research on AI because it shows how to write a program to solve something the programmer doesn't know how to solve. The trick is to tell the program what kind of things to try; you need not to know which one actually will work. The major issues of information processing theory are [Newell 72]:

1. A human being, when engaged in problem-solving, is an information processing system consisting of an active serial symbol-manipulating processor, input (sensory) and output (motor) systems, internal short-term and long-term memories, and an external memory.

2. Each type of problem is represented by symbolic search space, and problem solving consists primarily of searching through alternative nodes for a solution.

3. The nature of a class of tasks largely determines the structure of the search space needed for solving those tasks. An effective (human or computer) problem-solver constructs a search space appropriate to the type of problem at hand before it begins the detailed job of searching for a solution to a specific problem.

4. The structure of a search space determines the nature of the possible program that can be used for problem solving. Moreover, the problem-solver's knowledge - the specific facts, techniques, and experience he has previously acquired - play a central role in determining problem solving behavior.
knowledge is of particular importance to those interested in design methods.

2.3.3 Christopher Alexander: Synthesis of Form

In the book *Notes on the Synthesis of Form* [Alexander 79], Christopher Alexander argued for a rational, explicit design method to replace intuitive individualism. The general argument was followed with a proposal for a technique for analyzing the complex structure of design problems, using set and graph theories, the mathematics of classification and structural relationships. He claimed that "The starting point of (design) analysis is the requirement. The end product of analysis is a program, which is a tree of sets of requirements. The starting point of (design) synthesis is the diagram. The end product of synthesis is the realization of the problem, which is a tree of diagrams.... To achieve this we must learn to match each set of requirements in the program with a corresponding diagram" [Alexander 79, page 54]. The method was applied by way of illustration to the design of an Indian village.

Alexander introduced a distinction between two kinds of design processes, one which he called *unselfconscious* and the other *self-conscious*. The unselfconscious process is that which goes on in "primitive societies", or in architectural vernacular contexts; while the self-conscious process is that which is typical of educated, professional designers or architects. The two ends of his distinction are contrasted as representing quite distinct methods of producing designs. It seems that Alexander prefers the products of unselfconscious design processes rather than the products of self-consciousness. In order to see the value, he inspects the nature of the unselfconscious case and how its results are actually achieved.

Charles Eastman described Alexander’s method of generating problem decompositions as bottom-up planning which is in contrast to the top-down planning that has occurred traditionally in design [Eastman 72]. Top-down decomposition is based on the typological orientation that has existed in design for many years. Since the pattern language approach [Alexander 77] is apparently not being pursued consistently enough for its advantages to become apparent, designers will probably have to rely on a typological approach sometimes. According to Eastman’s view, we may
presume that Alexander’s self-conscious design process is much more related to the top-down approach.

2.3.4 Conclusions

There is considerable commonality between the theory mentioned in this section and the models of design computation. In particular, they all emphasize first on extensive problem exploration and analysis to identify all factors to be taken into account, and second on systematically establishing interconnections between these factors so that all sub-problems are identified. They all adopt the common approach of first breaking down the overall problem into its sub-problems and then attempting to synthesize a complete solution by combining partial solutions. Another very interesting observation is that Alexander, Akin, Eastman, Newell, and Simon have worked intensively with the computer.

2.4 Example: Routine Design

Both current CAAD and KBDS developments are related to a class of human design tasks which is called routine design. Routine Design is probably the most common type of design. Routine Design can be categorized into the following two types of design: parametric design and configuration design, based on its general characteristics. Parametric design starts with a given schema, the designer then adjusts parameters to the specification of the design problem. The parameters, typically geometric properties or materials, are well understood and are manipulated either in the designer’s memory or with advanced modeling systems. The functional requirements of the design are known and the semantics or the teleology (the purpose of each element) of the design are not changed but accepted from previous examples. This type of design relies heavily on instantiation of designs from a catalogue of parameterized examples. In order to achieve a desired effect, configuration design involves the constant adjustment of various parts (e.g., rooms) with respect to one another and with respect to a whole (e.g., a building). One of the crucial issues in configurating assemblies is the representation of geometry and spatial relations among parts. Parametric design is related to top-down refinement, and configuration design is related to bottom-up composition. To be more precise, routine
design is the result of making design decisions in the context of a design situation where all the decision variable are known a priori [Gero 91].

2.5 Summary of Design Computation Models

In this survey, the dichotomy of CAAD versus KBDS is a rather philosophical one and very much dependent on the application. If all design knowledge of the application domain is well-known and clearly defined, we might be tempted to pursue implementation of a KBDS. On the other hand, if a sufficient amount of domain specific knowledge is not captured in the system, the design process indispensably involves interaction between the designer and the system, which means we should prefer the CAAD approach. A CAAD system contains less domain specific knowledge than an KBDS which is good for a narrow domain.

The distinction of inference methods in KBS is similar to that made in the case of CAAD: between bottom-up, equivalent to forward chaining because they start from data, and top-down, equivalent to backward chaining because they start from possible goals.

To summarize, routine design is related to deduction, which is logically truth preserving. Routine design problems can be solved by a top-down (goal-driven) process, or a bottom-up (data-driven) process (or the combination of the two). Mathematical models, such as algebra and logic are implemented to guide computer models to find the desired solution. More precisely, top-down refinement tools apply constraint propagation or strong methods, such as linear and dynamic programming to produce results. The problem lies in finding appropriate variables to model the problems and to express solution criteria. Bottom-up composition tools apply weak methods, such as generate-and-test, best-first search (current states are compared), hill-climbing (existing states are compared), means-ends analysis (goal states are compared), or constraint satisfaction to guide the system. It is inefficient and will result in the exploration of numerous dead-ends before a satisfactory result is found. Routine design relies on refinement of a pre-defined schema until the schema satisfies some constraints. Current routine design tools are limited to describing building forms under certain geometric or logical constraints.
Chapter 3
Machine Learning

The focus of this section is to discover and define machine-based autonomous intelligent agents. Learning is part of intelligent performance, since without it an agent is unable to adapt to unforeseen circumstances in the world, and cannot take advantage of experience to increase its capabilities for prediction and action. Machine Learning is concerned with improving performance by automating knowledge acquisition and refinement. A learning system accepts environmental observations and incorporates them into a knowledge base, thereby facilitating some performance task. Most researchers suggest methods for building upon existing knowledge bases, which should be highly compatible with psychological processes. Examples are the uses of schemata, accretion, tuning, and restructuring. Machine learning falls into three logical categories [Michalski 83, Michalski 86]:

1. Learning by Deduction draws deductive inferences from knowledge and reformulates it in the form of useful conclusions which preserve the information content of the original data. (Note: Learning by deduction was identified as a separate category only recently, e.g., explanation-based learning.)

2. Learning by Analogy consists of (i) transferring knowledge from past problem solving episodes to a new problem that shares significant features with corresponding past experience, and (ii) applying the transferred knowledge to construct solutions to new problems.

3. Learning by Induction, commonly defined as "the process of discovering principles by the observation and combination of particular instances", consists of drawing conclusions from what has been observed in a number of cases [Michalski 83].

According to standard opinion, the two main modes of inference corresponding to learning from experience are analogy and induction [Jantke 87, Russell 89]. These methods provide, respectively, new
knowledge of cues and models, new knowledge of facts, and new skill. We shall see that these are computationally similar, in that each involves the development of internal representations of whatever it is that is learnt.

3.1 Analogy

Analogy was much less well understood than induction, and perhaps it still is. The Handbook of AI goes so far as to say "We do not include ... any articles discussing learning by analogy, since this area has not received much attention" [Cohen 82, page 334]. Analogy is observed to play a key role in cognition, especially in creative problem solving and in learning [Keane 88]. Exercises in propositional analogy are a standard part of various intelligence tests. The following definition of analogical learning is given by Carbonell [Carbonell 83]:

> Analogical problem solving consists of transferring knowledge from past problem solving episodes to new problems that share significant aspects with corresponding past experience and using the transferred knowledge to construct solutions to the new problems.

Wolstencroft has defined seven important stages of analogy [Wolstencroft 89]. These seven stages begin with identifying that analogical reasoning may prove useful, which is followed by retrieving a source situation to reason by, elaborating it with respect to the purpose of the analogy, placing parts in correspondent mapping, inferring new knowledge, justifying the analogy, and finally consolidating the analogy. Most research to date has focussed on the stages of mapping and inference.

However, there are wide variations in what researchers perceive to be the exact role played by analogy itself. This section is an attempt to identify these different meanings of analogy and the roles played in cognition. The state of machine learning research with respect to three modes of analogy will be discussed.

3.1.1 Propositional Analogy

The first use of analogy, called propositional analogy, is concerned with the relations having the form "A is to B as C is to D". The process referred to is usually that of generating the fourth term of the propositional
analogy relation, given the other three. At first sight, propositional analogy seems to be a syntactic process: it appears that only the terms A, B, C, D and their structures are called into play, but no meaning is involved. Early attempts in building computer models of the propositional process started from these assumptions and were successful [Evans 63]. However, on a deeper reflection, one finds that even propositional analogy relations rely on an element of interpretation.

Indurkhya distinguishes the syntactic and interpretive propositional analogy depending on whether the grouping needs to be changed or not [Indurkhya 89]. A description can be associated to an object and labeled natural description. Any propositional analogy relation that can be comprehended from the natural descriptions of its objects is syntactic. On the other hand, if a re-description of at least one object is necessary, then it is interpretive. Stiny notices this syntactic and interpretive distinction in the applications of CAAD in his paper "A New Line on Drafting Systems" [Stiny 86].

3.1.2 Predictive Analogy

Predictive analogy has been the most widely accepted type of analogy and is in the center of interest for AI research. It refers to the process of justifiably inferring further similarities between two objects or situations based on some existing similarities. Of the two situations, one, called the source, is usually more familiar to the subject than the other, called the target. The inferred similarities form the basis of making a prediction about the target from the known features of the source.

An inference from predictive analogy is not necessarily "true" but is considered "justified", which is to say an inference from predictive analogy is not considered an inference in the logical sense of the word. The degree of reasonableness, or justification, is associated with the amount of known similarity between the source and the target.

3.1.3 Metaphor

The main characteristic of metaphor is that the similarities between the two objects or situations do not exist prior to viewing one as another, but are instead created by the process [Kittay 84]. Whenever we notice that two
objects or situations are similar, we can do so only with respect to their existing ontologies and descriptions. However, in metaphor a new ontology and a new level of description is created for the target object. Perhaps the main role of metaphor in cognition lies in creative problem solving. Metaphor can be defined as interpretations of a system of symbols in an environment other than its host environment. Thus, theories of models have a direct bearing on metaphor.

Metaphor has not been explicitly addressed by AI research so far. However, metaphor is very important in any type of design. Laseau lists possible sources of metaphor in architectural design [Laseau 80]:

1. Structural - referring to shape or relationship.
2. Mechanical - the way something operates.
3. Control - maintaining a condition.
4. Plant - goal orientation and differentiation.
5. Animal - behavior.
6. Man - Imagination and choice.
7. Society - interaction, competition, organization.
8. Symbolic - convention, references, suggestion.

Metaphor differs from predictive analogy in some important respects. First, predictive analogy is a process less often applicable than metaphor, since it requires that there be some existing similarities between the two situations. Secondly, the similarities in predictive analogy that are observed between the source and the target stem from their existing ontology. In metaphor, on the other hand, the ontology of the target is changed as a result of the process, and new perspectives are created. The created similarities between the source and the target are between the source and this new perspective of the target. Finally, predictive analogy predicts there might be other similarities, while metaphor makes no such claim. For instance, we can make an analogy relation "gills are to fish as lungs are to man". The metaphor "mermaid" might be derived from this analogy, but nothing in an opposite manner.

3.1.4 Structure-Mapping Theory

Structure-mapping Theory (SMT) is perhaps the best known and most widely used model for analogical reasoning. SMT assumes a predicate-like representation, distinguishing between objects, object attributes, and
relations. Structure-mapping uses connectedness through higher order relations, and consistent mapping through pairing the arguments of predicates when the predicates themselves are matched. However, the main focus is on mapping; there is no identification, retrieval, elaboration, consolidation. The inference is syntactic, and in the computational implementation works by traversing causal arcs from those relations not mapped to those relations mapped. SMT requires little domain, organizational, or semantic knowledge, instead asserting the importance of abstract, structural similarity [Gentner 83].

3.2 Induction

Induction is the process of explaining how past experience, the accumulation of observations, may be used to fill in gaps in our knowledge, and of constructing a model of the unknown, for example the future. Without this ability, rationality, which requires the prediction of the outcome of actions, would be impossible [Holland 86]. Classification and generalization of knowledge applied to a broader class of situations are key inference mechanisms in learning by induction as well as in design.

3.2.1 Concept Formation

Concept formation involves induction of general concept descriptions from specific instances of these concepts. It specifies all common properties of known objects in that class, and so defines the class in the context of an unlimited number of other object classes [Michalski 80]. One of the most famous learning programs that falls within the concept formation paradigm is Winston's program [Winston 75]. This program learned the concept of an arch from a number of structural descriptions of examples and non-examples of the arch concept. When new positive instances of the concept are not covered by the current definition, the current definition is too specific and has to be generalized. When non-examples are covered by the current definition, it is too general and has to be specialized.

3.2.2 Grammar Induction

Language is a major component of cognition and a central concern of machine learning research. However, progress in this area has been
much slower than in most other learning tasks, undoubtedly due to the inherent complexity of natural language. Despite its complexity, the task of natural language processing can be divided into a number of well-defined subtasks, and one of these centers on syntax. Because our knowledge of syntax is more complete than that of other components of language, the vast majority of language-related research in machine learning has focused on the task of grammar induction.

Two different paradigms have emerged for describing the grammar induction task. The first approach was formulated by Solomonoff in the early days of AI [Solomonoff 59]. It assumed only a set of legal sentences as input, from which the learner induced a grammar that would parse those sentences. This paradigm has sometimes been called grammatical inference. The second approach takes an alternative view of the grammar induction task: given pairs of sentences and their associated meaning, the learner induced grammatical rules for mapping legal sentences onto meaning structures. This paradigm is called grammatical mapping. Researchers who favored the grammatical mapping paradigm rejected the earlier approach because the language involved not only syntax but semantics [Langley 87].

3.2.3 Concept Clustering

Concept clustering is a machine learning task defined by Michalski [Michalski 86]. A concept clustering system accepts a set of object descriptions and produces a classification scheme over the observations. Clustering forms a classification tree over objects. These systems do not require a teacher to pre-classify objects, but use an evaluation function to discover classes with good concept descriptions. There are two problems that must be addressed by a concept clustering system:

1. The clustering problem involves determining useful subsets of an object set. This consists of identifying a set of object classes, each defined as an extensional set of objects.
2. The characterization problem involves determining useful concepts for each object class.

Methods of concept clustering differ from earlier methods of numerical taxonomy in that clustering quality is not solely a function of individual
objects, but is dependent on concepts that describe object classes and the
map between concepts and the classes they cover.

3.2.4 Version Space

The version space model is a significant step towards a general theory of
inductive learning. Many existing methods can be cast into the general
framework of version spaces. Version space has been defined by Mitchell
to provide a unified framework for describing systems that use a data-
driven, single-representation approach to concept learning [Mitchell 78].
Mitchell has noted that, in all representation languages, the sentences
can be placed in a partial order according to the generality of each
sentence. The set \( H \) is initialized to contain all hypotheses consistent with
the first positive training instance. New training instances are examined
one at a time and pattern-matched against \( H \) to determine whether the
hypotheses in \( H \) should be generalized or specialized.

3.3 Media for Knowledge Acquisition

The problem of knowledge acquisition can be approached from many
angles. According to the media used in a learning system, we divide the
research of machine learning into three areas: explanation-based
learning, similarity-based learning, and case-based learning [Gruber 89].

3.3.1 Explanation as a Medium

Explanation-based learning (EBL) is a deductive approach to obtain the
correct generalization. In such a program an explanation is typically a
proof, derived by a theorem-prover from an axiomatization of the domain,
and a single training instance is useful for the problem solving task
[Keller 88]. Explanations can be a two-way medium for knowledge
acquisition. The system can use explanations to communicate to the user
what it is doing. The user can employ explanations to teach the system
what to do. The structure of explanations mirrors the structure of
knowledge, as represented in the knowledge base. To fill in an existing
structure, borrow an existing explanation and substitute terms. To
extend the structure, construct a new explanation or modify an
analogous one.
3.3.2 Similarities as a Medium

Similarity-based learning (SBL) is the induction on empirical grounds of general concepts from a training set of instances. The generalization captures some regularity in the training set. In terms of set theory, a general class description should contain all of the training instances labeled with that class and exclude those labeled with a disjoint class [Michalski 80]. If instances and class description are represented in logic statement, a generalization should imply the instance. The process of forming generalizations from training data is called induction or generalization, and the purpose of SBL systems is to induce useful generalizations.

3.3.3 Cases as a Medium

Case-based learning (CBL) represents an alternative to inductive (SBL) and deductive learning from examples [Schank 82]. Case-based Learning emphasizes the teacher and de-emphasizes generalization algorithms. The purpose of learning generalizations in SBL and EBL is to find compact, abstract descriptions of classes, so that new instance can be efficiently classified. The case-based learner acquires knowledge to index and interpret cases. A generalization is a class represented as a richly indexed network of prototype cases, called exemplar. A new case is interpreted by its relationship to exemplar of classes.

3.3.4 Neural Nets

Neural nets simulate a different process from purely symbolic learning approaches. The major difference is that, in the former, the learning process is very much closer to an approximation process [Hopfield 86] and, in the latter, it is frequently a process of assembling and modifying symbolic descriptions. Neural nets have shown some success in knowledge-poor tasks, such as character recognition and other perceptual tasks. Expert systems, by their very nature, are knowledge-intensive, and thus are less amenable to learning with syntactic reinforcement methods. Once a neural net is tuned, there is no way to understand why it succeeds on some cases and fails on other except to say that the weights on some nodes are higher than on others [Buchanan 89]. Moreover, their construction is not free; considerable effort must be invested in laying out the structure of the network before examples can be
presented. The fact is that we do not have a highly developed conceptual frameworks yet for reasoning about neural nets.
Chapter 4

Design and Creativity

The domains of design range from the well-defined reasoning methods used in engineering design to the more intuitive methods one finds in architectural design. This simplistic taxonomy is perhaps as misleading as it is apparently helpful. If reasoning and imaging were truly independent categories of thought one should not be able to speak sensibly of "creative problem solving" or "logical artistic development", which are both quite meaningful concepts. Many kinds of problems, even in such apparently logical disciplines as engineering, can be solved creatively and imaginatively. Thus, the purpose of this section is to examine the fundamental characteristics of architectural design, and some other classes of design different from routine design discussed in chapter 2. For future design systems, the understanding of non-routine design is of paramount importance.

4.1 Some Characteristics of Architectural Design

While the architectural design process involves decision-making aimed at the reduction of alternatives in search of a final solution, it also involves elaboration aimed at expanding the range of possibilities. Most architects are not content with solving problems with existing knowledge; they want to expand the knowledge base at the same time. This leads inevitably into the realm of cognitive psychology, the study of problem solving and creativity. Psychologists have tended to study thinking by attempting to divide and classify types of activity which could be investigated separately. Perhaps the best used division is that between "reasoning" and "imaging"; both of which are obviously needed in design. This seems to offer the psychological distinction between design and art as discussed earlier. Design is directed towards solving a real world problem while art is largely self-motivated and centered on the expression of inner thought. The control and combination of rational and
imaginative thought is one of the designer’s most important skills [Wade 77, Lawson 80].

Major differences between architectural design and engineering design can be explained with the following three observations. First, it seems that in most cases, the problem space of architectural design problems is larger than that of engineering design problems. Normally, all the problems in an architectural design project can not be solved completely, so an architect only concentrates on the problems which are important or interesting based on his or her experiences. Second, the basic functional specifications of engineering design are required to be precision and safety, while for architecture they may be flexibility and luxury. Finally, for economic and performance reasons, engineer designers rarely redevelop the form of design artifacts, while some architects may be interested in developing new building forms to create a new design expressions. Faltings proposes positive-negative constraints, and design-domain knowledge to describe similar observations [Faltings 89]. These differences also lead to a different philosophy for the development of computational tools: engineering design tools emphasize constraint manipulations, while architectural design tools also need to be visualization aids at different levels of abstraction.

4.2 The Other Two Classes of Design

Buildings which look similar may have been designed in quite different ways. The effort a designer puts into a design varies depending on the type of the design problem. In routine design, the design process can be viewed as the selection among a previously known set of well-understood design alternatives. There are at least two other classes of design besides routine design [Coyne 87, Gero 88]: innovative design and creative design.

Innovative design may arise during routine design when a new requirement is introduced that takes the design away from routine, requires new components and techniques. Since some design knowledge remains constant, architects do not have to re-develop a new design schema for each encountered design problem. In building renovation, for example, parts of the original building are transformed while some new parts are built. Often architects modify other architects’ schemata because their design requirements are almost identical. Innovative
design includes prototype modification and prototype combination. In prototype modification, the designer has a general idea of the desired object; the design process is a goal-directed activity but cannot be completed by routine design because the functional description or the object properties are not achievable utilizing a given prototype. Therefore, modification of an existing prototype is necessary. In prototype combination, instead of modifying one single prototype, several prototypes are combined to form a new design prototype. The attributes of a new prototype are the combination of the attributes from other prototypes in order to resolve conflicts existing among different prototypes.

Creative Design is rare and defined by the development of new solutions whose goals may only be partially defined at the outset. Both functional requirements and the object's properties are not completely known. It is possible that a unique solution may be found to a problem in which case the result would be a new design expression. For instance, Gordon talked about "making the strange familiar", where an unfamiliar situation is seen in terms of a familiar one, in order to bring it within the cognitive grasp of the subject; and "making the familiar strange", where a familiar situation is viewed as a unfamiliar one, in order to solve a problem of the first situation that cannot be solved from the familiar perspective [Gordon 61].

Most people like to think that the architect's main problem is that of creativity. However, the overall creative process is not well understood [Broadbent 77]. Creativity may be described as: the "creative person" will initiate or carry out some specifiable psychological process - "the creative process" - in order to reach some creative solutions, "the creative product". Creativity occurs under the condition that a new and valuable intelligibility comes into being. A creation in the radical sense, then, must exhibit structure that is both unprecedented and unpredictable. The criteria of creativity might be [Amsler 86]:

1. created outcomes have intelligible structures that are irreducible;
2. the structure of created outcomes is unpredictable;
3. the structure of created outcomes is inherently and usually instrumentally valuable;
4. and the acts that lead to created outcomes include an element of spontaneity so that although they are directed and are controlled, they are discontinuous.

Creative persons and products may be more easily described than the creative process. In a certain sense, a creative artist or scientist is personally as much the product of a tradition as he is the producer of a product which transcends that tradition. Examples are creative architects - Le Corbusier, Frank Lloyd Wright. A creative scientific, or artistic, product constitutes or incorporates a innovative solution to a problem, inherent in a background of prior products, but not soluble on the basis of these prior products themselves. Examples are innovative designs like - Chapel at Ronchamp, Falling Water. It is not the purpose of this research to describe the creative process in design nor to develop a creative computer. However, the brief excursion to creativity was necessary to map the entire spectrum that future design systems need to cope with.
PART II

PROBLEM STATEMENT
AND
A CONCEPTUAL FRAMEWORK

In order to investigate the assumptions of this research, previous work of related fields has been examined in the context of computational design problem-solving, machine learning, and design creativity. Our survey in Part I supports the fundamental theory of this research. This part explains the necessity for research in this area based on the analysis through exploration of the characteristics of architectural design and the fundamental limitations of current design computations. In Chapter 5, the limitations of computer, problem-solving, knowledge representation, and knowledge acquisition are indicated, and requirements for the solutions are made according to the common direction of related research. In Chapter 6, the general relationships between design, knowledge, and learning are discussed, and a conceptual framework of analogical and inductive reasoning in design is described.
Chapter 5

Problems Statement and Requirements for Solutions

To develop computational tools that can better support designers and architects, relationships between current computational theories and architectural design theory will be examined. Inherent weaknesses and strengths will be identified. The analysis will be undertaken through exploration of the characteristics of architectural design and the fundamental limitations of current design computation. Requirements for solutions according to the concepts presented in the analysis will be described briefly.

5.1 Limitations of Current Design Computation

Historically, attempts have been made to explain the human brain's behavior in terms of the most advanced artifact of the time: in terms of clockwork mechanisms, telephone switchboard analogies, and now in terms of the digital computer with the associated mathematical theory of computation [Fischler 87]. However, this hope should be reconsidered carefully, both with respect to the problem of understanding and to man's attempts to build an intelligent device in his own image. In this section, the capabilities of the computer as an inference engine for design will be examined.

5.1.1 Limitations of the Computer

"Could a machine think?" There were some reasons for saying yes, based on the two important results in computational theory. The first was Church's thesis, which states that every effectively computable function is recursively computable [Wulf 81]. The second important result was Alan M. Turing's demonstration that any recursively computable function can be computed in finite time by a maximally simple sort of symbol-manipulating machine that has come to be called a universal
Turing machine (Turing 50). These two results entail that, whatever the function of a conscious person might be, a suitable symbol-manipulating machine could compute it.

There were also some arguments for saying no for the question "Could a machine think?" One of the arguments is Searle’s discussion that any system which merely manipulates physical symbols in accordance with structure-sensitive rules will be at best a hollow mock-up of real conscious intelligence, because it is impossible to generate semantics from empty syntax (Searle 81). For example, we naturally admire Einstein and Beethoven, and wonder if a computer could ever create such theories or symphonies. Most people think that “creativity” requires some mysterious “gift” that simply cannot be explained. If so, then no computer can create - hence, anything a machine can do can be explained.

In a normal person, the two hemispheres of the human brain are believed to deal with problems in two different ways (Springer 85). The left hemisphere of the brain is specialized to deal with tasks amenable to a sequential (logical) paradigm. These include language understanding and production, logical reasoning, planning, and time sense. The right hemisphere of the brain is more competent to deal with spatial tasks and tasks requiring a global (gestalt) synthesis. These include comparing and identifying visual imagery, visual and analogical reasoning, and body sense and coordination. This distinction is too simplistic, and the physiological basis of creativity versus intelligence or analytic thinking versus intuition is still unknown. However, this distinction may help us to make a comparison between the human brain and the computer. The conventional digital computer is a sequential symbol manipulator, and is primarily suitable for tasks that can be broken down into a series of simple steps. While design tasks have strong connections with the functions of the right hemisphere of brain, the original design of the current computer may be insufficient to support all human design activities.

5.1.2 Limitations of Problem-Solving

Existing design systems, such as HI-RISE (Maher 84), VT (Marcus 86), and IBDE (Fenves 89) utilize either a set of predetermined decompositions
of the structure of a design, or utilize design methods that generate, as part of the design process, a decomposition belonging to such a set. Therefore, all of the designs generated by these systems will share, at a relatively low level, a structural similarity. For design domains in which the most desirable structures can be enumerated or explicitly decomposed in advance, this approach proves useful. But there are many design problems where the structural decomposition of the most useful solution is not pre-determined. In terms of the amount and specificity of domain knowledge they require, problem solving methods may be classified as [Carbonell 85]:

1. No structured domain knowledge or useful past experience exists, weak methods are applied to search heuristically for possible solutions.
2. Application of general plans to reduce the problem by partitioning the problem or providing "islands" in the search space.
3. Specific domain knowledge exists in the form of plans or procedures and it instantiates these specific plans recursively.
4. Solving a new problem by analogy to previously solved similar problems.
5. A combination of these approaches.

For (1.) and (2.), mostly weak methods (configuration design) and direct plan instantiation (parametric design) have received substantial attention in current CAAD research. These two approaches generate design solutions from scratch and take no advantage of existing partial design solutions. An example for (3.) is the expert system XCON for planning computer system configurations; it typically searches 800 levels deep with a branching factor of 3. Although $3^{500}$ is astronomical, good heuristics enable XCON to make an optimal choice at each step. As a result, the computation is nearly linear. Still, the danger that all search programs have to face is the possibility of a combinatorial explosion: the amount of calculation can quickly grow exponentially.

Current models of design computation rely very much on the fundamental building block of formal reasoning systems which is called deduction. Formal reasoning involves the syntactic manipulation of a data structure to deduce a new one following pre-specified rules of inference. Deduction is a process of reasoning in which a conclusion follows necessarily from the stated premises; inference by reasoning from
the general to the specific. However, mathematical logic and deductive inference play only a minor and secondary role in human expert decision making. Moreover, in many design cases, designers use non-measurable judgement from the category of ‘love’ or ‘hate’ to decide on design alternatives, a more intuitive approach than deductive inference alone.

5.1.3 Limitations of Knowledge Representation

Ernst reports on success and failure of 16 knowledge-based systems (XCON, MACSYMA, PUFF, ACE, MYCIN, DELTA, and etc.) of which only 5 are in commercial and operational use [Ernst 86]: “As systems grow in size to, say 8-10,000 rules, there begins to be a loss control on the part of the developers. The system is too large to be fully comprehended, and our means for representing knowledge (in terms of rules) are inadequate in terms of providing tools or structures to maintain a manageable perspective and level of understanding of exactly what is happening.” The essence of the problems, is due to inadequacies in the problem solving methods, rather than inadequacies in the available knowledge.

Most first-generation expert systems were rule-based with a separate inference engine. The rule-based approach has proven to be both practical and profitable, and resulted in a number of commercial expert systems. Nevertheless, for handling more complex forms of expert problem solving, there is a need for knowledge representation approaches with a richer set of constructs. The constructs should be helpful in capturing other more structured forms of knowledge and should be such that they help organize both knowledge and problem solving behavior for more focussed problem-solving.

5.1.4 The Knowledge Acquisition Bottleneck

Present techniques of knowledge acquisition, derived from non-architectural applications such as medicine and geology, are cumbersome and require knowledge engineers as human translators between knowledge sources and computer programs [Schmitt 88B]. Most AI programs written to date have achieved their knowledge through careful ‘hand crafted’ programming, which have been purely deductive in nature and have not involved learning or self improvement through experience. For example, the XCON configurator handles all routine
orders for VAX systems, but orders with unusual requirements still require the attention of a human expert. A learning system which could sort, sift, and organize the information would free us from the tedium of doing the knowledge encoding by hand. Techniques for automating knowledge acquisition process are essential for the future development of knowledge-based systems.

5.2 Requirements for Solutions

The requirements for solutions are based on our previous discussion of the characteristics of architectural design, the limitations of current problem-solving, knowledge representation, and knowledge acquisition technologies. They will be the foundation of our theory and for the proposed design environment.

5.2.1 Knowledge Acquisition by Inductive Learning

The primary goal of many design processes is to refine the design, change the design state so that it is more detailed in the final state than it was initially. Inversely the process may be to abstract the design, to generalize some aspect of it. This is usually associated with learning and is an integral part of the design process [Chen 90]. Learning denotes the way that people increase their knowledge and improve their skills.

To extend the problem solving abilities and to avoid the knowledge acquisition bottleneck of current design computation tools, systems allowing the problem-solver to learn from experience are necessary [Mostow 85, Shapiro 87, Poetschke 89]. They must be capable of automatically filling in missing information (by asking appropriate questions), by drawing upon an internal knowledge base and of integrating acquired knowledge into the original knowledge base. An ideal learning program should remember a lot about its past so that, for each new problem, it would search for methods like the ones that work best on similar problems in the past.

5.2.2 Case-based Reasoning and Lateral Thinking

Simple inference methods for controlling the reasoning like generate and test, forward-chaining, and backward-chaining are very effective in a wide variety of problem domains, when they are coupled with powerful
knowledge bases. They are good in applying a standard problem-solving method to many different cases, but they perform poorly in discovering new methods. Often experts are not aware of the rules they use to solve a problem; their knowledge exists at a subconscious level [Quinlan 82, Mitchell 90]. One possibility of getting the information to be put in a knowledge base is to learn from examples supplied by experts.

Traditional architectural design relies in practice and in education much on architectural history and design case studies of existing buildings. The fact that architects can synthesize a new design from merely "looking" at two or more existing drawings, thus executing a design problem solving task, strongly suggests the involvement of analogical and inductive reasoning in the design process. The production of architectural working drawings, commonly known as design draughting, involves copying-and-pasting and subsequent modification of the copied entities. The draftsman must decide on transformations between the copied entities and the target working drawings. Also, the common use of tracing paper as overlay in architectural offices suggests that the design process involves learning by analogy.

5.2.3 Graphical Knowledge Acquisition in Design

The design process in architecture can be thought of as a series of transformations going from uncertainty toward commitment of design details. The successive stages of the process are usually registered by some kinds of graphic model. In the final stages of the design process, designers use highly formalized graphic conventions some of which make use of descriptive geometry. One of the most useful qualities of graphic communication is that it can be transmitted and received on several levels, simultaneously [Eastman 85]. Spatial reasoning takes advantage of the power of visual perception by making images external and explicit [Adams 86, Arnheim 69].

A language consists of a set of rules by which symbols can be related to represent larger meanings. The difference between verbal and graphic languages is both in the symbols used and in the ways which the symbols are related. Much more significant, verbal language is sequential, it has beginning, a middle and an end. Graphic language is simultaneous, all symbols and their relationships are considered at the same time. The
simultaneity and complex interrelationship of reality accounts for the special strength of graphic language in addressing complex problems. Solving a problem using graphical representation is often similar to performing a physical experiment on a "real-world" situation, as opposed to obtaining the solution by an algorithmic technique applied to a symbolic description [Barr 81]. Architectural practice [Eastman 75] and psychological experiments [Akin 88] supply the evidence that there is much graphical thinking involved in architectural design processes:

- In design development, graphical representation is preferred over others because of its compatibility with physical object descriptions.
- Graphical representation provides the designer a means to examine design at various levels of abstraction.
- Designers do not deal with problems directly but with representations of problems.
- Designers communicate with the client and the contractor using drawings.
- Results of psychology experiments show that experienced architects pre-store composite units of information about shape, functional relations, and architectural elements graphically [Canter 74].

This suggests that, given a valid computational model for design and its classes, graphical thinking and learning can be integrated into the CAAD process itself and thus end the unfortunate situation that a growing number of computational methods is separated from the rich set of graphical metaphors developed over time by designers [Laseau 80, Schmitt 89B]. As geometric and functional concepts of multi-dimensional objects in architectural design are difficult to describe [Bijl 85], it would be easier to teach the system by showing examples of existing concepts and have it learn the appropriate generalizations and descriptions.

5.2.4 Summary
The current knowledge representation of most knowledge-based systems is based on production rules; they have a fixed strategy for applying the rules (either forward or backward-chaining). These strategies only represent one special type of human reasoning called deduction. We define as important features of the next generation knowledge-based design systems in architecture: case oriented strategy, problem solving by
analogical reasoning, automated knowledge acquisition by induction, and
graphic learning capabilities.

It was not the purpose of this chapter to establish a two-part theory in this
chapter - right-left brain, architect-engineer, reasoning-imaging, art-
science - but to highlight possible deficiencies of the current approach of
knowledge engineering in design.
Chapter 6

Design, Learning, and Knowledge

Questions about how humans learn design, and how they should be taught, cause speculations about distinct types of knowledge in architectural design education. To consider how a machine can learn to design, the distinction between representing concepts and procedures plays an important role in knowledge acquisition. The discussion of declarative-procedural controversy enables us to examine the knowledge involved in the design synthesis and design analysis, and the acquisition of synthetic design knowledge and analytic domain knowledge.

6.1 Learning in Architectural Design

The activity of design is directed towards resolving a problem at hand and producing acceptable solutions by seeking additional information in a cyclical process. While going through a design process, the designer continually learns more about the problem, its constraints and potential solutions. Thus, we may consider the design process also as a learning process. This view of design has considerable advantages over traditional computer-based design models because it better reflects the dynamic aspects of design and learning and offers an open versus the traditional closed system approach. Two interesting forms of architectural design knowledge acquisition strategies are learning by analogy and induction. Analogy is the transfer of information from one situation to another, similar situation; induction is the acquisition of general principles from collections of individual experiences.

6.1.1 Analogical Learning in Architectural Design

In a process that can lead to innovation, designers modify an existing schema or combine several existing design schemata, an activity that can be mapped to analogical problem solving [Mostow 87, Navinchaandra 87].
Design prototype modification: If a particular design solution has been found to solve a problem similar to the one at hand, perhaps it can be used, with minor modifications, for the present design problem. The solution to previous problems must be transformed to satisfy the requirements of the new problem statement. Knowledge transfer is accomplished by transforming the copy of an original solution incrementally based on some primitive transformation process until the modified solution satisfies the requirement of the new problem.

Design prototypes combination: When a new design problem is encountered that does not lend itself to direct plan instantiation or direct recognition of a solution pattern, the designer begins to analyze the problem by combining components from different schemata to derive possible design solutions. Useful experience is encoded in the reasoning process needed to derive solutions for similar problems, rather than just in the resulting solution. Previous design case studies are important at the beginning of each design project, because a design concept might develop from studying previous design experience, which is also one of the method in teaching architectural design. Design prototype combination also applies when a design project is required to combine functions from different types of buildings.

6.1.2 Inductive Learning in Architectural Design

The process of adding and organizing new information into an existing knowledge base is important for design as well as for other disciplines. The human ability of induction enables designers to interact with the design environment without becoming overwhelmed by its complexity.

Concept formation in design: Specific architectural expression exists in design products and is represented by common features related to a particular period, region, or building type (individual designer or school of design) [Simon 75, Chan 90].

Grammatical inference in design: An architect’s expression is the result of specific actions. Acquisition of the common action of a design language is similar to the grammatical inference in natural language. In shape grammar applications, a computer program can be employed to generate forms in the corresponding languages once a design grammar has been
defined (Flemming 86). Reversely, a design grammar can be generated by the computer from a family of design solutions. Such a method may be applied to investigate the design grammar of individual designers or schools of design, e.g. Mies van der Rohe or the New York Five.

Concept clustering in design: A conceptual clustering system for design accepts a set of design descriptions and then produces a classification scheme over the observations. In developing a new design concept, the designer re-packages existing knowledge into new schemata. For example, design knowledge can be classified into building type, style of design, and national or regional architecture.

6.2 Declarative-Procedural Controversy of Knowledge

To incorporate problem solving techniques in the design process, the major issues become how to represent each state of design and the control of state-transformations in an efficient manner. Related issue in AI is the declarative-procedural controversy. The declarative approach assumes that knowledge is a collection of facts that can be stated in logical propositions, concept, or other symbols. The procedural approach assumes that a person's knowledge of the world is embodied in procedures that actively interpret the environment and operate on it. As an example, Simon cited the following two specifications for a circle [Simon 82]:

1. A circle is the locus of all points equidistant from a given point.
2. To construct a circle, rotate a compass with one arm fixed until the other arm has return to its starting point.

The first sentence is a declarative definition that does not say how to draw a circle. The second is a procedure for drawing a circle that does not say how to recognize a circle. Similar distinctions have been made by other researchers, for example Piaget distinguishes between conceptual understanding and successful action [Piaget 78]; Tulving distinguishes between semantic memory and episodic memory [Tulving 83]; Anderson distinguishes between declarative and procedural knowledge [Anderson 83]; Gruber distinguishes between substantive and strategic knowledge [Gruber 89]. Parallel distinctions are made in philosophical and psychological theory of knowledge. For example, Scheffler distinguishes
between the propositional use of "knowing that" and the procedural use of "knowing how to" [Schell 65]. Gestalt psychology puts emphasis on concept learning and development; and behavior psychology focuses on skill performance.

However, linking declarative and procedural knowledge benefits declarative knowledge as well as procedural knowledge [Hiebert 86]. Procedural knowledge that is informed by declarative knowledge results in symbols that have meaning and procedures that can be remembered better and used more effectively. Procedural knowledge provides a formal language and action sequences that raise the level and applicability of declarative knowledge.

Winograd explored the advantages of these two approach, and concluded the distinction is not a technical one, but rather it is an expression of an underlying difference in attitude toward the problems of complexity [Winograd 75]. Programming language like LISP considers "program are data". We can think of the LISP interpreter as the only program in the system, and everything else as data on which it works. Everything, then is declarative. From the other end, we can view everything as a program, or everything is a procedure. For instance, PASCAL programming emphasizes on the constructions of a sequence of procedures to manipulate pre-defined data structures. Nevertheless, most of the programs written in procedural programming languages can be converted into declarative programming languages, or vice versa. While most of the traditional programs are interested in "knowing how", most AI programs are interested in "knowing that", thus, most of the AI programs are written in declarative languages.

6.3 Types of Knowledge

One of the remarkable attributes of human intelligence is the ability to convert a problem into a familiar form or representation that can be operated on using previously known techniques [Barr 81]. Knowledge representation in computers can be divided into declarative knowledge, procedural knowledge, compiled knowledge and meta-knowledge:
1. *Declarative knowledge* (Kd) describes how things are, and contains objects, their attributes and values (Kd-1), and the relationships or associations between them (Kd-2). It can be retrieved and stored but cannot be immediately executed; to be effective, it must be interpreted by procedural knowledge.

2. *Procedural knowledge* (Kp) is knowledge about how to do things, containing a set of instructions for performing a task; it allows to manipulate declarative knowledge.

3. *Compiled knowledge* (Kc) is knowledge which generalizes a set of related Kd or Kp to perform a specific task. Although the details of the specific experiences may be lost in compiled knowledge, it provides a short cut which allows the problem-solver to conduct a task easily.

4. *Meta-Knowledge* (Km) is knowledge about the knowledge represented. It is a self-knowledge that a system has about operators or reasons, and when and how to best use and control its domain knowledge.

For instance, when a CAAD system is used, the data structure and its associated operations or functions establish a design world. One of the simplest components of CAAD programs is the set of operators that determine the order in which elements will be considered, e.g., move, insert, delete, and etc., which is corresponding to procedural knowledge. Another set of elements in most CAAD programs consists of prefabricated solutions to subproblems that arise repeatedly in different contexts, e.g., line, circle, polygon, and etc., which is corresponding to declarative knowledge. Combining these two representations, designers could work on a problem from the inside out or from the outside in. The “inner environment” of design is concerned with the design process represented by a set of given alternatives of action. The “outer environment” of design is concerned with the design product which is represented by a set of parameters [Simon 82]. The “inside-out” process relies on determining the progressive actions of a design process⁴, the “outside-in” process relies on specifying a design description⁵. In this sense, design can be considered as an activity either to find satisfactory actions or to determine proper parameters for given design problems.

---

⁴ This leads to the bottom-up composition process which we discussed in Chapter 2.
⁵ This is related to the top-down refinement process.
Recent discussions of *design prototypes* [Gero 88] are based on the representation of compiled knowledge employing frames or rules. From a machine's view, the knowledge compilation can be conceptually formulated as:

\[
Km (Kd, Kp) \Rightarrow Kc \quad (6.3)
\]

The purpose of (6.3) is not intended to be exhaustive or exclusive, but provides a notation to help describe how different types of knowledge can be represented in computers.

Kd, Kp, Kc, and Km are related to "states", "operators", "intermediate or generated states", and "control strategies or heuristics" in *information processing theory* [Newell 72], and are related to "data structure", "procedures", "outputs of procedure", and "compiler" in traditional programming languages.

In a pure frame-based representation system, the representation of the declarative knowledge is emphasized, which strongly relies on "IS-A" or "PART-OF" to describe the relationships between objects. In a pure rule-based representation system, the representation of procedural knowledge is emphasized, in which rules have the form "IF some conditions are met THEN take some action". In deductive systems, which are the variants of production systems, the declarative knowledge is also encoded into a rule format. The rules of deductive systems have the form "IF some predictions are true, THEN conclude some other predictions are also true" [Schank 89].

### 6.4 Knowledge Compilation and Knowledge Acquisition

Knowledge acquisition might be considered as the inverse process of knowledge compilation [Brown 89]. Design cases can be considered as pieces of compiled knowledge, or the outputs of some programs. Compiled knowledge must be transformed into more general, lower or higher level pieces of knowledge that can be used efficiently. Once a set of design cases which are related to a design concept are presented, we can examine the similarities between the generating patterns or rules\(^6\) to derive another level of knowledge. In such a way, we can assume compiled knowledge is

\(^6\) More precisely, rules of deductive systems.
already known as priori in a knowledge acquisition process and we can
declare two of the other types of knowledge in (6.3) to be constant, then the
last unknown type of knowledge may be acquired. More precisely, the
meta-knowledge should be also known a priori, which relies on the
designer's domain knowledge. Theoretically, we can either acquire the
declarative knowledge or procedural knowledge from the design cases.
However, due to the computational intractability of procedural knowledge
acquisition, only the acquisition of declarative knowledge will be
implemented in this research. Further details of this approach are
described in the following sections.

6.5 Knowledge in Learning Systems

We may classify learning systems based on the trade-offs between the
amount of effort required from the learner and from the teacher. For
example, the teacher may put much more effort on an explanation-based
learning system than on a learning by discovery system. A useful
classification of search methods in learning distinguishes methods in
which the presentation of the training instances drives the search (data-
driven methods) from those methods in which an a priori model guides
the search (model-driven methods).

Another long-term dichotomy - within the linguistic and psychological
literature on language acquisition - concerns the amount of knowledge
initially available to the learner. Some researchers argue that humans
come to the grammar learning task with considerable knowledge of the
domain, including the basic form of sentences and the basic word classes.
According to this view, a weak learning method suffices to acquire syntax
because the space of the possibilities is so constrained. In contrast, some
other researchers claim that humans have little innate knowledge of
language and that they acquire grammar using the same powerful
inductive techniques they employ for other domains. In this scheme, the
learning task is constrained not by prior knowledge, but rather by
experience itself [Langley 87].

This research declines the first view of language acquisition and takes
the model-driven approach. There are two major reasons: First, design is
a knowledge intensive task, which is hard to be interpreted based on
grammatical inference only. Secondly, because of the difficulties of
spatial reasoning, there is more than one way to interpret an architectural design description (drawing) into one single syntactical form. A design description contains information in three dimensions, which cannot be easily interpreted as a sequential sentence.

6.6 Causal Inversion and Knowledge Acquisition in Design

In most engineering disciplines, design has been described as the transformation from some design functions (design requirements or specifications) to design structures (structures or design descriptions). In design computational processes, the properties of the function are transformed into required behaviors, and the properties of the structure are transformed into actual behaviors. Since function and structure are not homogeneous, the required behaviors and the actual behaviors are the intermediate phases represented in computational processes to carry out the matching process between function and structure [Gero 89]. In architectural design, functions may include building types, purpose, etc., and structures describe the physical existence of design, behaviors are the expected responses of the design in current design context, e.g., cost, circulation, and etc. [Rosenman 91]

If we accept the simplified definition of design in the last paragraph, then, there are at least two types of knowledge involved in design, there are synthetic design knowledge, and analytic domain knowledge. The first type of design knowledge requires the presence of a function and the designer tries to find the proper behavior and structure. The second type of knowledge analyzes whether a proposed structure satisfied a function. Although design is concerned with the transformation from function to structure, however, knowledge acquisition in design is interested in going from structure to function [Bobrow 84]. The same observations of "form follows function" or "function follows form" can be found in architectural design.

In the knowledge-based design systems, synthetic design knowledge which includes Kd-1 and Kp is inherited in the design generator, and analytic domain knowledge includes Kd-2 which is inherited in the design evaluator. In axiomatic mathematics, the general procedures are applied to the domain-specific data to make deduction. The facts are

---

7. This argument is commonly accepted in most IFIP conference proceedings.
axioms and the thought process involves proof procedures for drawing conclusions from them [Winograd 75]. As well as, from the view point of shape grammar, Kd-1 is a set of vocabulary, Kp is the grammar, and Kd-2 is the evaluator. Kd-2 examines the semantics of a generated 'sentence', which is produced by Kd-1 and Kp, is legal or not. The examples in Chapter 7 will provide us a better understanding of these explanations.

However, the acquisition of synthetic design knowledge has difficulties. There are two major problems that will be encountered in this approach. First, the mapping from function to structure is a one-to-many mapping, there are several causes for one effect. The second problem is, even if all mappings are known in advance, it is not clear which mapping decides the final structure. Even in the case when the acquisition of individual design operators in the synthetic design knowledge is possible, it gathers bulky information in a disorganized raw state form.

Three other reasons convince us to decline the acquisition of synthetic design knowledge: First, there are too many degrees of freedom to represent synthetic design knowledge in architecture (Section 4.1). Secondly, design knowledge contains Kd-1 and Kp, which cannot be predefined and separated easily. Thirdly, our discussions in previous sections support this point of view. Specially, we refer to Clancey's explanation of heuristic construction (Section 2.3.2), Indurkhya's distinction between syntactic and interpretive propositional analogy (Section 3.1.1), grammar induction (Section 3.2.2), and dichotomy of grammar learning (Section 6.5).

At this point, the automated knowledge acquisition in design has to be reconsidered based on the fundamental mechanism of the computer and the relationships among domain knowledge, design structure and actual behaviors. The domain knowledge becomes a powerful tool for design if it can be made a causal, linking aspect of behavior or function to features of the design artifact itself [Faltings 89]. The previous cases of design, then, can be considered as a designer's views of previous design problems, which can be represented as conceptual graphs [Sowa 84] or semantic networks. Therefore, we may use the domain knowledge to identify su-

---

8 In a context-free sense.
trees of design common across many cases. Knowledge acquired in this mode then can be synthesized to form a case prototype which can be executed to solve the similar problem [Garner 88]. Comprehensive examples will be provided in Chapter 8 and Chapter 9.
PART III
IMPLEMENTATIONS

The previous chapters describe modeling issues of analogical and inductive reasoning in architecture. This part describes how the various pieces fit together by presenting comprehensive examples which illustrate the major components of the model, and the boundaries of the modeling domain. Several computer programs are developed in this part to demonstrate different approaches for building knowledge-based design systems in architecture.

In Chapter 7, a rule-based system called BAU is developed, helping the domain expert to investigate a basic architectural unit. BAU is developed under the traditional knowledge engineering paradigm, and uses traditional knowledge acquisition techniques - discussions between domain expert and knowledge engineer to translate the knowledge into the system. During the development of BAU, several interesting problems arise with regard to knowledge acquisition. Those problems give us insights to develop the consecutive programs of BAU in Chapter 8 and Chapter 9.

Chapter 8, presents an example-based reasoning system EB-BAU, which demonstrates that part of the design rules of BAU can be derived based on the inductive learning mechanisms. Chapter 9, emphasizes the reuse of the design solutions and the design problem solving strategies of BAU to the new and similar design problems. A case-based system CB-BAU is introduced, which demonstrates another method of building design systems. The underlying mechanisms of CB-BAU are based on analogical and inductive reasoning.
Chapter 7

BAU: A KBDS for the Investigation of A Basic Architectural Unit

The control of incremental complexities within an evolutionary design process has been a concern in both architectural education and practice. One method of examining this problem is to first define a "basic architectural unit" and a design environment which is composed of multiple units. Different levels of detail will be added to the unit as the design process continues. Secondly, a related computer program called BAU is introduced, which demonstrates that a computer is a meaningful tool for helping the architect to investigate various aspects of a design problem. Finally, both the domain expert's and the knowledge engineer's experiences during the development of BAU are described.

7.1 Introduction

The notion of anonymous architecture [Moholy-Nagy 57] together with the concept of self-organizing processes [Alexander 79] as basic phenomena in architecture have changed the perception of architecture in the 20th century in a rather unobtrusive yet very fundamental way. A further expansion of these basic notions took place when the observation of basic types, and the consequent development of typology, were introduced into the field of anonymous architecture. Together, self organization and typology form powerful concepts for the understanding of human habitat and its settlement patterns as well as the generic processes of both phenomena.

The idea of a small, low key research test project, relating to anonymous architecture came into existence during a chance meeting of a group of people involved in CAAD. A suggestion that concepts developed in AI techniques may be useful for exploring aspects of anonymous architecture guided us in the initial formulation of this research project.
It was assumed that similar to cells in a living organism, basic units could be defined in an anonymous architecture which again like cells in an organism form clusters or aggregations of larger functional entities such as a house or habitat [Alexander 77, Steadman 79].

Starting with this notion, we defined for our own work a “basic architectural unit” in the form of a cube, since the cube is both in quantitative as well as qualitative terms highly defined and relates well to the CAD environment. In addition to this, the total number of units⁹ in an aggregation, their functions - or attributes, as well as the environment (as a geometrically structured field) were defined. Following the definition of basic conditions, a process was set in motion which can be seen either as a design process based on AI methodologies or as a simulation of a process leading to a typologically defined “population” of aggregates (comparable to the habitations we find in ecological niches, such as islands or mountain valleys).

This project, if the goals set out at its inception (described above) are pursued to their logical conclusion, will provide additional insight into the nature of self-organizing processes in architecture. It will provide a better understanding of the processes which generate the human habitat. Furthermore it will inform us about contemporary housing problems in developing nations since they can not be seen in the framework of a traditional projects environment. In an “architecture without architects” [Rudofsky 88] the principles of “classical” or “formal” architectural design do not apply. Finally, it can be expected that the project will provide new insights into the nature of possible CAAD application in an area which does not compete with the architectural profession but rather fills a gap where architecture is needed but the architect can not be afforded.

It is quite clear that there are no easy solutions. Yet we believe that this modest research project points in a direction in which, up to now, very little intellectual energy has been applied. The computer with its power of simulation could make substantial contributions in this area. Additionally, this project could show us a new direction in design.

⁹ From now on, we will use “unit” to indicate the “basic architectural unit”.
education where, until now, only the area of "Architettura Maggiore" has been addressed. "Architettura Minore" or Anonymous Architecture is the kind of architecture that concerns the majority of the urban population in developing nations today.

7.2 Methodology

Although design has been classified as a complex human activity, it is only recently that concepts and methods for capturing, organizing, and using design expertise have emerged from research in artificial intelligence and cognitive psychology. To generate new insights into the design process, this research incorporates knowledge engineering techniques [Rychener 88] to investigate the architectural design process. The primary purpose of this study is to explore the incremental complexity within a design process, the secondary purpose to specify the knowledge that the architect used for solving a design problem and the utilization of that knowledge in a design system [Hayes 83]. BAU is a knowledge-based design system which has been developed for these purposes.

Design has been modeled within the framework of a general theory of problem-solving [Newell 72]. Several prescriptive models related to design computation have been proposed, such as, near decomposable hierarchical systems [Simon 82], design model of functional reasoning [Freeman 71], generative systems [Mitchell 77], general design theory [Yoshikawa 87], and prototype refinement [Gero 88]. According to these approaches, design problem-solving can be characterized as a process of searching through alternative states of the representation in order to discover states that satisfy specified criteria. BAU is related to the idea of design problem-solving [Kalay 85]. The development of BAU is aimed at defining a proper knowledge representation with which to manipulate the process of transforming a design problem into solutions.

This research is also related to shape grammars [Stiny 80] and space synthesis systems [Eastman 75], however, this research will only be described according to the knowledge engineering paradigm. In this section, we introduce the basic architectural design object, and our

---

10. The term is used here as employed by Stanislaus Von Moss in 1964.
methodology for controlling and investigating the design process within
the BAU design environment.

7.2.1 Prefabrication and Standardization

Prefabricated building systems consist of a set of components to construct
complete structures for some building types. A major advantage of such
systems is that they could be pre-engineered. The performance
requirements for various combinations of components could be analyzed,
separated from any particular use, and stored in tabular form. Thus the
engineering for such buildings becomes simple [Eastman 80]. Several
prefabricated building systems have been implemented, e.g., OXFORD
and SEAC systems [Myer 75].

The main objectives for this kind of systematic design is to consider
design for assembly in an organized manner, such that each aspect of
design activity is considered in a fixed sequence and that the
implementation of design are consistent with the decisions made
[Eastman 80]. The time and cost of design and manufacturing can be
reduced through standardization associated with group technology.
Group technology (GT) is a manufacturing philosophy in which similar
parts are identified and group together to take advantage of their
similarities in manufacturing and design. Hence, the processing of each
member of a given family would be similar, and this results in
manufacturing efficiencies. These efficiencies are achieved in the form of
reduced setup times, lower in-process inventories, better scheduling,
Improved tool control, and the use of standardized process plans [Groover
84].

7.2.2 The Basic Architectural Units

We defined a “basic architectural unit” as a cube that is initially 3 x 3 x 3
meters. A unit is a conceptual building module. The four vertical faces
of the defined cube can be thought of as four walls, the bottom face as the
base, and the top face as the roof of a building module. The allowable
combinations of the units are governed by a set of geometric cubic
relationships and design rules defined by the domain expert. As the levels
of specification increase, the variations generated by the units becomes an
extremely large set of design alternatives, too numerous for a human
designer using traditional methods to produce. BAU has been developed in order to assist the architect in the selection of satisfactory alternatives based on the design rules he defines, and to help him control the incremental complexity of the design process. The design process incorporated in BAU is a reflection of the domain expert's preferences. BAU is developed as a rule-based design system [McDermott 80]. The development of BAU incorporates the following three alternate processes:

1. Possible design alternatives are generated based on the knowledge available about the current unit.
2. The domain expert's design rules are applied to the generated alternatives, and only those solutions, which comply with the rules are retained.
3. The unit is redefined as a more sophisticated unit for the next generation.

In the first phase, BAU generates configurations of basic units according to geometric relationships which allow units to be located, removed, or replaced within the design environment. At higher levels of development, the allowable combinations of units are governed by sets of design rules defined by the domain expert. Detailed operations are described in the following sections.

7.2.3 BAU: System Overview

To examine the complexities within a design process, we should discuss at least two basic phenomena of design: design process and design object [Simon 75]. An architectural design process involves the transformation of one physical situation to another, which is a large sequence of smaller design processes. The sequence could conceivably never be complete. The efforts of these transformations focus on the "design object", which is the result of the design process.

Broadly speaking, BAU has been developed as a top-down refinement paradigm. Since the problem space generated by BAU is large and complex, it has to be decomposed into pieces or subproblems which are small enough to solve. Solutions to subproblems have to be integrated into an overall solution. However, because generate-and-test cycles are also necessary for each intermediate subtask, bottom-up composition is incorporated in the system.
The design process of BAU can be considered as a state-space search process in which the designer attempts to locate satisfactory choices within an extremely large set of design alternatives. A hierarchy is established at the uppermost level and the general functional requirements for the design units are stated. The abstract refinement process is precompiled into a standard structure of parameterized components, and the design process consists of reasoning about constraints in order to determine appropriate parameter values.

7.3 Development of BAU

BAU involves a whole sequence of generate-and-test cycles. The first process generates certain design objects that are candidate solutions or components of solutions. The second process tests whether or not a given candidate satisfies a design constraint. In such a generate-and-test process, the design is assembled component by component. Each generated component is added to the previous assembly and the new structure then is tested.

BAU contains four subtasks, BAU.I, BAU.II, BAU.III, and BAU.IV. BAU.I designs two dimensional configurations for BAU. BAU.II extends a unit into a three dimensional unit which contains wall elements, slanted roofs and interior level changes. BAU.III introduces modularized elements to the units and associated construction rules to the environment. BAU.IV constructs four bau-houses in a small housing cluster. The following sections describe what each of BAU’s subtasks involves.

7.3.1 Subtask I (BAU.I)

We begin by positioning the units on a plane. The chosen environment (building site) for this study, in order to be compatible with the concept of the "basic architectural units", is a 3 by 3 square grid, with each square measuring three meters by three meters (figure 7.1A).

The unit is only considered as a two dimensional square in BAU.I. Four types of functional units, each three meters square, are chosen: a utility unit, consisting of a kitchen, bathroom, and hallway, represented by the
U-Unit shown in figure 7-1B, a living unit, a sleeping unit, and a balcony unit, represented by the L-, S-, and B-Units in figure 7-1C to E respectively. There are eight possible U-Unit configurations, which can be obtained by rotating the standard U-Unit and its symmetric equivalent in ninety degrees increments. Correspondingly, there are four possible B-Unit configurations. The design task is to locate one U-Unit, S-Unit, and B-Unit, and two L-Units in the 3 by 3 grid.

![Diagram of U-Unit configurations]

Figure 7-1: The Basic Architectural Units and their environment.

The relationships between the units and their environment are governed by the three fundamental rules described in table 7-1. Rule BE-1 and BE-2 confine the search space of BAU into a manageable size. Rule BE-3 imposes a basic formation on the house as a response to changing weather. Based on these three BE rules, the architect develops more sophisticated two dimensional design rules for BAU.I. The design rules of BAU.I are classified into three categories. There are rules for: (1) internal connections, (2) external orientations, and (3) ranking.

<table>
<thead>
<tr>
<th>Table 7-1: Rules for describing the BAU environment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule BE-1: Each unit must be located exactly on a single cell of the 3 by 3 grid.</td>
</tr>
<tr>
<td>Rule BE-2: Two connected units must share an edge.</td>
</tr>
<tr>
<td>Rule BE-3: All units must be connected.</td>
</tr>
</tbody>
</table>

### 7.3.1.1 Rules for Internal Connections

The first category of design rules is "rules for internal connections", which specify the functional relations between units. Table 7-2 lists seven
rules for internal connections. The sequence of these rules is determined by first the simplest units (L-Units), in order to reduce the search space of BAU.I. Rule IC-1, IC-2, and IC-3 specify spatial relations between units. The architect’s contention that a hallway between the living units and sleeping unit is necessary for a house is the basis for Rule IC-4 and IC-5. Rule IC-6 and Rule IC-7 are imposed by the definition of the B-Unit.

<table>
<thead>
<tr>
<th>Table 7.2:</th>
<th>Rules for internal connections.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule IC-1:</td>
<td>The two L-Units have to be connected.</td>
</tr>
<tr>
<td>Rule IC-2:</td>
<td>The B-Unit has to be next to an L-Unit.</td>
</tr>
<tr>
<td>Rule IC-3:</td>
<td>The kitchen has to be next to an L-Unit.</td>
</tr>
<tr>
<td>Rule IC-4:</td>
<td>The hallway has to be next to the S-Unit.</td>
</tr>
<tr>
<td>Rule IC-5:</td>
<td>The hallway has to be next to an L-Unit.</td>
</tr>
<tr>
<td>Rule IC-6:</td>
<td>The opening of the B-Unit should not be connected to an L-Unit.</td>
</tr>
<tr>
<td>Rule IC-7:</td>
<td>The opening of the B-Unit should not be connected with an S-Unit or a U-Unit.</td>
</tr>
</tbody>
</table>

### 7.3.1.2 Rules for External Orientations

The second category of design rules for BAU.I, “rules for external orientation”, are listed in table 7.3. These two rules specify the possible orientations of the balcony. The architect’s reason for Rule EO-1 is that a balcony facing north cannot get any sunlight in winter. Rule EO-2 is based on the idea that proper orientation of the balcony will reduce undesirable noise from the street.

<table>
<thead>
<tr>
<th>Table 7.3:</th>
<th>Rules for external orientations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule EO-1:</td>
<td>The opening of the B-Unit should not face to north.</td>
</tr>
<tr>
<td>Rule EO-2:</td>
<td>The opening of the B-Unit should not face the street.</td>
</tr>
</tbody>
</table>
7.3.1.3 Ranking Rule of BAU.I

BAU.I produces a number of solutions. The development of ranking rules enables the architect to judge these configurations based on his own criteria. One of the ranking rules is shown in table 7-4. Instead of specifying what should or should not be, this rule describes what would be better. This rule has been translated into a set of ranking values shown at the right side of table 7-4. The program sums up the ranking value of empty cells, a configuration with a higher score indicates it is more proper according to the architect’s criteria. The score ranges from 7 to 11. Moreover, multiple ranking rules could be applied to BAU, and optimal solutions could be found based on the architect’s preferences (weighting factors and objective functions) for each ranking rule. However, as we do not intend to investigate the problem of optimization in design. At the consecutive stages of BAU, different levels of detail will be added to the units based on the results of BAU.I.

Table 7-4: Ranking rule of BAU.I.

<table>
<thead>
<tr>
<th>Rule RK-1: Locating garden at the south of site will be better.</th>
<th>2</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

7.3.1.4 Results of BAU.I

A tic-tac-toe like algorithm is applied to produce the first set of configurations based on the BE Rules. The total number of this set of configurations is 49. Figure 7-2. illustrates 14 representative configurations.

Figure 7-2: Representative configurations generated by BE rules.
A significant configuration is a combination of five units, representing the four functional units and the B-Unit, located on the 3 x 3 grid according to the internal-connection and external-orientation rules that were specified by the domain expert. BAU.1 generates 216 significant configurations, the first 96 of them are shown in figure 7-3.

![Figure 7-3: 96 of 216 significant configurations generated by BAU.1.](image)

Part of the results of Rule RK-1 are shown in figure 7-4. The configurations at the top of the figure, which have a higher score (11), are ranked higher than the configurations at the bottom.

![Figure 7-4: 60 of 216 ranked configurations based on Rule RK-1.](image)
7.3.2 Subtask II (BAU.II)

The next step, once an effective method of generating significant two dimensional configurations had been developed, is to establish another set of design rules which allows us to explore the significant configurations in three dimensions. In BAU.II, the unit is extended into a three dimensional cube. The interior level changes and slanted roofs are introduced to define a second type of unit, which is shown in figure 7-5.

![Figure 7-5: Second type of basic architectural unit.](image)

There are eight possible configurations for this set of units, which can be obtained by rotating the standard unit by ninety degree increments and shifting between the interior level changes of one meter and none. The interior level changes make the interior space and the design problems for BAU more interesting.

The complexities of a unit increase dramatically when the design tasks move from a two dimensional plane to three dimensional space. There are a few important repercussions resulting from the transformation into 3D. First, the problem space BAU encounters becomes extremely large. Each single significant configuration of BAU.I has 1024 (i.e. $4^5$) alternatives for roof combinations. Secondly, we are able to discuss the semantic level of the design problem, e.g., the visual impact of the design, which is no longer restricted to a syntactic level. Thirdly, the rules for satisfactory combinations of roofs are much more difficult to describe and to derive, and translation of the design rules into computer code becomes more complicated. These issues also suggest that design in the third
dimension is one of the most significant aspects of architectural design. These implications are explored in the following subsections.

### 7.3.2.1 Rules of BAU.II

This set of rules is based on the need to provide satisfactory water run-off between roofs and visual appearance. These rules are not easy to describe verbally. During the knowledge acquisition of BAU.II, the domain expert drew these rules graphically in order to explain his ideas to the knowledge engineer.

| Rule RF:1 | If two units are adjacent, then the L-Face of one unit should not attach to the L-Face or S-Face of another unit. |
| Rule RF:2 | If three units are in a row, then the L-Face of the middle unit should not attach to the H-Face of another unit, and the H-Face of the middle unit should not attach to a L-Face of the other unit. |
| Rule RF:3 | If two units are adjacent and one unit’s L-Face is attached to another unit’s H-Face, then the first unit should be raised 1 meter. |
| Rule RF:4 | If two adjacent roofs have the same height, then they should share either an H-Face or an S-Face. |

To make the rules more comprehensible, a notation about the second type of unit is provided at the top of table 7-5. Although Rule RF-4 is stated quite succinctly, it was the most complex and difficult rule to translate from the domain expert to the knowledge engineer and then to the machine. The architect knew what represented satisfactory and unsatisfactory roof combinations, but had trouble distilling a complete rule to describe his criteria. We worked a couple of weeks to solve this problem, and to derive Rule RF-4.
7.3.2.2 Results of BAU.II

After the completion of the cycle related to Rule RF-3, each single significant configuration of BAU.I has about 100 to 200 alternatives for roof combinations. The results of configuration No. 216 generated by Rule RF-3 are shown in figure 7.6.

![Diagram of configuration No. 216 generated by Rule RF-3]

Figure 7.6: Results of configuration No. 216 generated by Rule RF-3.
A. U-Unit   B. L-Unit   C. S-Unit   D. B-Unit

Figure 7-8: Third type of basic architectural unit.

7.3.3.1 Rules of BAU.III

Rules for BAU.III address the relationships between adjacent walls. The design decisions are based on the internal circulation and the locations of windows, doors, and interior walls.

Table 7-6: Design rules for adjacent walls.

Rule AW-1:
Two adjacent L-Units have to share an S-Face.

Rule AW-2:
U-Unit has to share an S-Face with an L-Unit.

Rule AW-3:
The opening of the S-Unit should be across from the face attached to the U-Unit.

7.3.3.2 Results of BAU.III

For each significant configuration of BAU.I, 1 or 2 three dimensional combinations are generated by BAU.III. In the first phase of BAU.III, we were interested in developing different types of roofs and skylights. Figure 7-9. presents part of the generated results. In the second phase of BAU.III, detailed information is added, e.g., columns, beams, stairs, doors, and windows. figure 7-10 presents representative combinations.
Figure 7-8: Partial results generated by BAU.III.

Figure 7-10: Detailed house samples generated by BAU.III.
7.3.3.3 Different Levels of Abstraction

We have shown the sequence of how a basic unit is transformed from a two dimensional square into a three dimensional building object, and the compositions of different levels of units in the design environment. Although the units have been compiled into a complicated building block, different levels of information about each unit have been stored in different layers during the design objects compilation. This information is available to the architect, and it allows the architect to examine the design object taking advantage of various levels of abstraction.

7.3.3.4 Current Development of BAU (BAU.IV)

Our current research interest is to investigate the concept of "a basic architectural unit" in a larger scale design environment. The design environment of BAU.IV is defined as a 9 by 9 grid, and the design task is to locate four generated bau-houses on the grid.

Figure 7-11: Current development of BAU.IV.
This extension emphasizes the relations among multiple bau-houses. In addition, new design elements are introduced, e.g., compound walls, trees, streets, street lamps, fountains, etc. The development of BAU.IV enables us to re-examine the potentials and limitations of our methodology. Retrieval and combinations of bau-houses are the most crucial parts of this stage of work. Since BAU.IV is still under development, only partial results of BAU.IV are presented. Figure 7-11 shows a perspective of a bau-cluster generated by BAU.IV.

### 7.4 Experiences of BAU

The present section discusses our experiences in the development of BAU. Knowledge representation, control strategies, programming hardware and software, and the project history of BAU are addressed.

#### 7.4.1 Knowledge Representation

The units and their relationships which comprise the design environment are defined explicitly and can therefore be translated easily into descriptions which a computer can manipulate. Two types of information are important for BAU. In order to express the information about the units at different levels and the relationships between the units, a declarative form of representation is necessary. Additionally, a uniform representation for symbolic and graphical representation is an important issue. The List representation is selected because of its flexibilities in knowledge representation.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>010</td>
<td>011</td>
<td>(0 (U (H NOT-W) (K N) (B s) S 0 L 0 B L 0) 0</td>
</tr>
<tr>
<td>B L 0</td>
<td>0 L 0</td>
<td>0 0 S</td>
<td></td>
</tr>
<tr>
<td>0 U S</td>
<td>0 U S</td>
<td>0 0 L 0 B L 0</td>
<td>(0 (U (H NOT-W) (K N) (B s) S 0 L 0 (B W) L 0)</td>
</tr>
</tbody>
</table>

Each configuration of BAU is represented as a list containing nine elements. The first element of a bau-list occupies the south-west corner of the environment, and the last element of a bau-list occupies the north-
east corner of the environment. At the first level, each element of the bau-
list contains a binary number, 0 or 1, "0" indicates an empty cell and "1" 
indicates an occupied cell. As the process continues, each element which 
is "1" in the bau-list is replaced by the symbol "B", "L", "S", or "U" to 
indicate that the occupied cell has been replaced by a functional unit. In 
the later refinement process, each symbol can be extended into a more 
complicated sub-list which contains more information about the unit.

Table 7-7 presents the manipulation of a bau-list during a BAU.I cycle, 
for a two dimensional configuration. The graphical routines read in the 
bau-list and display the graphical results on the screen (table 7-7D). The 
same concept of list representation can be extended to the third 
dimension, information about roofs, and columns, etc, is added to each 
sub-list. The knowledge representation of BAU also imposes rules about 
how an abstract design element can be converted into a real building 
entity and visualized.

7.4.2 Control Strategy

BAU separates knowledge according to the way it is to be processed. Two 
sorts of knowledge are retained: knowledge for generation and knowledge 
for testing. Both types of knowledge can be expressed by rules: 
constructive and restrictive rules, corresponding to generators and filters 
[Takala 87]. The first kind of knowledge is important for maintaining and 
handling the additional information as the levels of detail increase. The 
second kind contains the heuristics of the domain expert. The distinction 
between these two types of knowledge is similar to the distinction between 
the working memory and production memory elements of a production 
system. A generate-and-test method is applied to generate the new 
working memory element and then to test it against the production 
memory. It is however uneconomical to use an exhaustive method to 
generate the problem space, especially when each problem state contains 
not just a single symbol. To avoid an exhaustive generate-and-test 
process, a more efficient search for the significant results is obtained by 
imposing constraints on the intermediate stages as they are generated.

7.4.3 Problem Space and Solution Sets

Although it is hard to estimate how much information an architect 
should maintain within a design process and how complex a design
The design kit for BAU was first developed using COMMONLISP. AutoCAD and AutoLISP, the computer graphics software system currently being used for the second phase of our development, enable the programmer to define instances, create modules by combining transformed instances, and display the results in two or three dimensions. BAU runs on both IBM RT and SUN4 workstation, under the UNIX operating system. The results of BAU.IV are sent to a special 3D graphics program called STALKER written with the GL graphics library for Silicon Graphics Workstations. This allows the architect to study the volume, material, and color of bau-houses, and to “walk through” the building site in real time.

### 7.4.5 Project History

We started the discussion of this project in March 1990. Since then, we have had weekly meetings between the domain expert and knowledge engineer. Each meeting lasted about one hour. During the meetings, the domain expert drew his ideas on paper, and explained his ideas to the knowledge engineer. After the meetings, the knowledge engineer tried to translate the domain expert’s ideas first into rules, and then into computer code. Each subsystem of BAU took approximately two to three months to develop.

An earlier project [Roger 78] had first addressed some issues relevant to this project. Two major conceptual problems in the approach taken earlier were (1) generation of the problem space, and (2) acquisition, formalization, and integration of domain knowledge.

### 7.5 BAU and CAM

There are two important considerations for linking BAU with CAM: fabricating the building components of BAU and manufacturing process planning [Groover 80, Rembold 82]. The results of BAU are represented as a list of identifiers for each component and their locations. Since the layout possibilities of BAU are limited to the form of a grid, the locations of components and their connectivity could be stored as entries in a standard rectangular array format. The resulting geometric data may be used to produce detailed design working drawing. However, the major
concept of BAU is to transmit data via CAD/CAM system rather than with engineering drawings. Thus, BAU can be easily linked with the numerical control (NC) instructions for fabricating the part [Smithers 89]. Process planning is concerned with determining the sequence of individual manufacturing operations needed to produce a given part or product. The computer would employ a set of algorithms to progress through the various technical and logical decisions toward a final plan for manufacturing. The short term goal of this research is to assemble the part of BAU in a production line. Our long term goal is to enable the robots to construct BAU at the building site.

7.6 Summary of BAU

BAU has proven that the computer is a meaningful tool for investigating design problems which can be specified by well-defined objectives operating on pre-determined variables or features. Through the development of BAU, the design process becomes directly accessible for detailed examination.

There are several questions that remain unanswered in this research: How does a designer accept a design solution in such a large problem space? How can a designer tune a mediocre design scheme into a good design solution? How does a designer structure his design problem before he starts his design problem solving? What are the differences between experts and novices (students) in solving this type of design problem? How does a novice learn to design? How does one develop a general design tool for this type of design tasks? These questions are interesting and probably could be examined by incorporating BAU with cognitive psychology studies of design.

The final consideration is how to extend the acquired experiences of BAU to other designs problems. We'll discuss the knowledge acquisition and machine learning aspect of BAU in Chapter 8 and Chapter 9 based on inductive and analogical reasoning.

The results of this investigation demonstrate some of the possibilities offered by computers for exploring more fully the consequences of design decisions. We anticipate that BAU can be augmented to address new research issues in the incorporation of knowledge-based design systems.
in architecture, computer-aided architectural manufacturing, and robotic task planning.
Chapter 8

Inductive Reasoning in BAU

The development of the current knowledge acquisition method in the BAU design system, presented in Chapter 7, relied on information and ideas that were derived from interviews between the knowledge engineer and the domain expert. Some of the design rules of BAU (e.g., Rule RF-4) were not easy to describe or derive. This indicated the need for a more efficient knowledge acquisition method. In this chapter, an inductive learning system is presented, in which design concepts and rules are extracted from positive and negative examples provided by the designer.

8.1 Test Case: Acquiring the Rules of BAU.I

This section demonstrates the possibility of inducing BAU.I rules by using the EB-BAU.I computer program. Seven significant configurations of BAU.I are selected as positive training instances, and the program then generates a description that is true for all of the positive instances. The criteria for generalization are based on the relationships between the units. In particular, the program is implemented based on the principle of *maximally-specific conjunctive generalizations* (MSC-generalizations) [Dieterich 83].

![Diagram of shapes: S-Shape, Y-Shape, T-Shape, W-Shape, L-Shape, U-Shape, P-Shape]

Example:

<table>
<thead>
<tr>
<th>Shape</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-Shape</td>
<td>148 (setq layout-148 '(0 0 B U L S 0 0))</td>
</tr>
<tr>
<td>Y-Shape</td>
<td>152 (setq layout-152 '(0 0 S L L U 0 B 0))</td>
</tr>
<tr>
<td>T-Shape</td>
<td>156 (setq layout-156 '(0 0 L S U L 0 0 B))</td>
</tr>
<tr>
<td>W-Shape</td>
<td>158 (setq layout-158 '(0 0 B 0 L L S U 0))</td>
</tr>
<tr>
<td>L-Shape</td>
<td>172 (setq layout-172 '(0 0 B 0 L S U L))</td>
</tr>
<tr>
<td>U-Shape</td>
<td>206 (setq layout-206 '(0 0 B 0 S L L U))</td>
</tr>
<tr>
<td>P-Shape</td>
<td>216 (setq layout-216 '(0 0 0 0 L S B L U))</td>
</tr>
</tbody>
</table>

Figure 8.1: Examples of EB-BAU.I input and the underlying knowledge representations.
The results of conjunctive generalizations:
((L NEXT-TO U) (L NEXT-TO B) (L NEXT-TO L) (S NEXT-TO U)).

The results of disjunctive generalizations:
((L NEXT-TO U) (L NEXT-TO B) (L NEXT-TO L) (S NEXT-TO U)
(S NEXT-TO L)).

**Figure 8-2:** EB-BAU.I outputs.

Figure 8-1 illustrates seven positive training instances of EB-BAU.I and the internal list-representation for each training instance. The positive instances represent seven representative two-dimensional layouts of BAU. The outputs of EB-BAU.I are shown in figure 8-2. The source code of EB-BAU.I is included in Appendix I.

Further inductions can be made based on the seven lines of code in the upper part of figure 8-2, e.g. *conjunctive or disjunctive generalizations*. The results of conjunctive and disjunctive generalizations are presented in the lower part of figure 8-2. The difference between the two generalizations is that the conjunctive generalization induces one more rule than the disjunctive generalization, namely (S NEXT-TO L). However, this rule is only represented in example No. 216, and it is too specific for the other instances, therefore only the results of the conjunctive generalization are considered. The four rules derived by conjunctive generalization are incorporated as Rules IC-1 to IC-5 of BAU.I. Some other two-dimensional rules of BAU can also be derived using similar mechanisms, but more detailed information must be included in the training instances.

### 8.2 Conjunctive and Disjunctive Generalizations

Based on the example presented in the last section, induction may be considered as a search through a description space. The goal of the search and the criteria for evaluation must be pre-specified. Although these criteria depend upon domain specific knowledge, some general regularities of induction are obvious. That is, the version space of an
induction can be represented by its *maximally-specific* and *maximally-general* descriptions, both of these descriptions can be applied to discriminate the given class from all other possible classes.

More precisely, all of the positive instances satisfy a characteristic description, which is equivalent to a maximally-specific description. Furthermore, a discriminant description is not satisfied by all of the negative instances. Characteristic description and discriminant description are related to conjunctive and disjunctive generalizations [Michalski 83]. The determination of characteristic and discriminant descriptions is the subject of learning from examples.

Conjunctive generalization is typically a collection of simple properties common to all objects in a given class. From the application viewpoint, the most interesting properties are maximal characteristic descriptions that are the most specific logic products can characterize all objects in the given class. Disjunctive generalization is a logical disjunction of the properties common to all objects in the class. The most interesting are minimal discriminant descriptions that are the shortest expressions distinguishing all objects in the given class from objects of other classes.

### 8.3 Acquiring the Rules of BAU.II

Although the rules of BAU.II are stated quite succinctly, the knowledge acquisition and application tasks of BAU.II took months of effort to achieve. The basis for the rules of BAU.II were difficult for the domain expert to identify and translate to the knowledge engineer, who had further trouble translating it to the machine. The difficulty was that the architect could recognize what represented satisfactory and unsatisfactory roof combinations, but was unable to formulate a complete set of rules in the beginning. One explanation for this is the complexities of the problem space for BAU.II increased tremendously, it contained too much information for the human designer to handle in other than an intuitive way. It was not a problem at all for the domain expert to distinguish between the instance which he liked and did not like. Figure 8-3 shows the positive and negative examples identified by the domain expert of BAU. The proposed system EB-BAU.II tries to recognize the underlying reasoning mechanisms behind the domain expert's
preferences according to the positive and negative examples. The results of EB-BAU.I support this assumption in part.

A. Positive Examples.

B. Negative Examples.

Figure 8.3: Positive and negative examples (Nos. 216 and 152) identified by the expert.

Similar tasks have been addressed in computer vision, making use of similar techniques of machine learning. One of Winston's programs was mentioned in Section 3.2.1. Another related work was conducted by Huffman and Clowes. It reconstructs a three-dimensional scenes from a two-dimensional projections by labeling line drawings in the trihedral world [Cohen 82]. Huffman and Clowes attempted a systematic approach to polyhedral scene analysis using constraint propagation to determine the possible labels for the edges.

8.3.1 Knowledge Representation of EB-BAU.II

Most of the high-level processing problems in computer vision deal with finding the proper representation for translating iconic or analogical representations down to a symbolic level which computers can manipulate. Since the knowledge to be acquired in EB-BAU.I and EB-BAU.II are different, the knowledge representation of EB-BAU.I is not appropriate for EB-BAU.II. Another knowledge representation has to be developed for EB-BAU.II.
Figure 8-4: Knowledge representation and acquisition in EB-BAU.II.

Figure 8-4A illustrates the knowledge representation of EB-BAU.II, which emphasizes the individual orientation of each roof. The sequence of notation in the layout list follows the convention described in Section 7.4.1. An "O" in the list represents an unoccupied cell of BAU, and the sublists, e.g. (R W O), represent the orientation and interior level change of a BAU.
unit. For instance, (R W 1) describes a BAU unit that has a roof which slants to the west and an interior level change of one meter.

8.3.2 Knowledge Acquisition in EB-BAU.II

Once the knowledge representation was selected, the problem was how to derive useful abstractions of these types of BAU roofs. It was not clear how to solve the problem in the beginning. One possibility (the brute-force approach) was to parse each layout into five sub-layouts, each sub-layout specifying a BAU unit and its neighbors. Figure 8-4B shows how each sub-layout is examined in order to identify the allowable combinations rather than the whole layout, which contains more complex information. We can even sub-divide the sub-layouts down to a level which contains only the information about relationships between two roofs. Figure 8-4C shows 10 of the simplest roof combinations found in the initial layout. Accordingly, we can also parse the initial layout list into 10 sub-lists which contain more detailed information. However, there are 64 possible combinations of two adjacent roofs, therefore, an economical way to derive the final positive combinations should be considered in advance. In this implementation, to identify and eliminate multiple instances of uniform relationships between roofs, we rotate each roof so that it slants to the east, and rotate the corresponding adjacent roof accordingly (see figure 8-4C). By comparing the results the number of possible combinations is reduced to 1/4 of the original, which means, that for the maximum case of 64, only 16 combinations will need to be considered. After the transformation shown in figure 8-4C, we start to parse the initial layout list into a list which contains pairs of roof relations. The final step is to remove the duplicate information from the derived list (see figure 8-4D).

8.3.3 Rules Derived by EB-BAU.II

The derived relations (rules) between roofs are presented both symbolically and graphically in figure 8-4E. The derived rules are not as compact as corresponding rules of BAU.II, however, they relate to Rules RF-1, RF-3, and RF-4 (Rule RF-2 is excluded, since it is a constructive rule). Although only one graphical knowledge acquisition example of BAU.II is shown in figure 8-4, we can make the generalizations based on several results of this kind of analysis. But there is a significant difference between the generalizations of EB-BAU.I and EB-BAU.II. The
former is implemented using conjunctive generalization, and the latter is implemented using disjunctive generalization.

8.3.4 Knowledge Application in EB-BAU.II

Even though the mechanism behind the expert’s reasoning seems to be derived, it is hard to tell whether the solutions make sense. To test whether the application of the derived rules produces practical results or not, the results of EB-BAU.II were compared with the results produced by BAU.II. The 1,024 possible BAU.II combinations (which include positive and negative examples) were divided into 4 sample sets containing 256 samples each. The 1,024 samples were produced from layout No. 216 of BAU.I.

Each sample set was run to determine which samples represented positive, negative, or ambiguous examples. Based on these partitions, 4 sets of rules for designing roofs were derived. Then, seven representative layouts from BAU.I were chosen (as in Section 8.1), which contain 1,024 different roof combinations. Each of the 1,024 combinations was tested against each of the sets of rules derived by EB-BAU.II, and the graphical output for the “good” combinations of roofs for each representative layout were produced. The graphical outputs enabled us to visually examine the results of the EB-BAU.II algorithms. Intuitively, we feel satisfied with the agreement between the input positive examples and output positive instances. The results of different sample runs are shown in figure 8-5 to figure 8-8. Those figures present comprehensive graphical results of this approach. Additionally, the significant results of BAU.II are all covered by the solution sets of EB-BAU.II.

Table 8-I: Comparisons of different sample space and solution sets.

<table>
<thead>
<tr>
<th></th>
<th>Sample I</th>
<th>Sample II</th>
<th>Sample III</th>
<th>Sample IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Size</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>No. Positive</td>
<td></td>
<td>3</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. Ambiguous</td>
<td>10</td>
<td>17</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. Results</td>
<td>148</td>
<td>4</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>152</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>156</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>158</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>172</td>
<td>16</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>206</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>216</td>
<td>4</td>
<td>15</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>
A. Positive examples of Sample I.

B. Sample results of Sample I.

Figure 8-5: Positive examples and sample results of Sample I.
A. Positive examples of Sample II.

B. Sample results of Sample II.

Figure 8-6: Positive examples and sample results of Sample II.
A. Positive examples of Sample III.

B. Sample results of Sample III.

Figure 8-7: Positive examples and sample results of Sample III.
A. Positive examples of Sample IV.

B. Sample results of Sample IV.

Figure 8-8: Positive examples and sample results of Sample IV.
Table 8-1 records the number of positive and ambiguous examples of each sample set, and the number of results after the required knowledge was applied to different BAU layouts. There is no significant connection between the number of positive examples and the number of results produced. The possible interpretations which can be made include: Layout No. 156 is similar to Layout No. 172, and Layout No. 206 is similar to Layout No. 216 (since they have three identical values in a row). These interpretations can be made based on the information in a table such as table 8-1 and can be confirmed by analyzing similarities in two-dimensional layouts of Layouts No. 156 and 172, or 206 and 216.

8.4 Summary of Inductive Reasoning

In the initial stages of the implementation of ideas described in this chapter, the use of the ID3 shell, a concept clustering shell, was considered. However, we found these tasks more related to concept formation rather than concept clustering, and a graphical interface was required to present the results, so programs were developed in LISP.

One question which is difficult to answer is why conjunctive generalizations is used in EB-BAU.I and disjunctive generalizations in EB-BAU.II. One of the answers is that a hierarchical design system can be represented as an AND-OR search tree. The decision to use either conjunctive or disjunctive generalizations to acquire domain knowledge is really dependent on whether the AND or the OR part of a design decision tree is emphasized as the criteria for knowledge acquisition.

Although the mechanisms behind EB-BAU are quite simple, they enable us to acquire knowledge about BAU in a different way, and they also establish the fundamental mechanisms of CB-BAU, which is described in the next chapter. With this kind of a tool, knowledge engineers would still be able to construct knowledge-based design systems with information from the domain expert which may be incomplete or uncertain. The extensions of this application will be discussed in the next chapter in more details.
Chapter 9

Analogical Reasoning in BAU

The purpose of this chapter is to describe the mechanisms with which the user of a knowledge base can enter information about attributes that can in turn be used to recognize similarities in other cases. Analogical inference is used to determine which similarities should be considered relevant, which will lead to purpose directed analogical inference\(^\text{12}\). Thus, the unstated, implicit assumptions of the knowledge source become explicitly stated, thereby offering the user of the system greater flexibility and more control. The explicitly stated relevancy criteria can be easily examined and modified.

Based on the nature of knowledge transferred from previous experiences to the new problem, the analogical reasoning approach is classified into two categories: transformational and derivational analogy [Carbonell 85]. Transformational analogy adapts the solutions to the past problems to the new problem. Derivational analogy applies the problem solving methods used to solve the previous problems to solve the new one [Gero 91]. Analogical reasoning could support a design system based on an extension of the version of BAU presented in Chapter 7.

9.1 Adaptations of Previous Design Solutions

Similar distinctions of case adaptations have been described in case-based reasoning (CBR) literature. First, there is structural adaptation, in which the adaptation rules apply directly to the solution stored in a case. Second, there is derivational adaptation, in which the rules that generated the original solution are re-run to generate a new solution [Carbonell 83]. In CBR, both kinds of mechanisms are present. This

---

\(^{12}\) Since any two examples have many ways of being mapped to each other, one needs to know the purpose for which the analogy is being made in order to select the appropriate mapping.
section focuses on structural adaptation. Derivational adaptation will be addressed in Section 9.2.

### 9.1.1 Null adaptation

The fundamental level of adaptation is null adaptation. Null adaptation, correctly speaking, is neither structural nor derivational [Riesbeck 89]. The simplest technique is to alter nothing and simply apply whatever solution is retrieved to the new solution. Null adaptation also applies to tasks for which, even though the reasoning may be complex, the solution itself is very simple.

![Examples of null adaptation](image)

**Figure 9-1:** Some other possible environments other than the 3 by 3 site for BAU.

Simple examples of null adaptation in BAU can be shown by using other design environments for BAU (figure 9-1). Although the design environments of BAU change, some of the solutions for the 3 by 3 site can still be applied to different sites. For example, most of the U-Shape and F-Shape layouts generated by BAU.I (see figure 8-1) can be applied to the 4 by 2 site in figure 9-1A, without being modified. A simple matrix matching is used to derive the solutions for the new sites. For shapes of BAU.I layouts, which cannot fit into the new site properly, a simple move-unit function may be defined that is able to transform the unsatisfactory solutions into satisfactory ones. The advantage of using null adaptation is that the whole generative process of BAU does not have to be rerun to derive new solutions, instead simple case retrieval techniques can be used to assist the designer in finding solutions more easily.

The number of significant configurations, generated by BAU.I rules, for the 4 by 2 site is 115. 79 of those 115 solutions can be derived by using null adaptation, which means that about 70% of the solutions for the 4 by 2 site can be retrieved from the set of solutions previously found for the 3 by 3 site.
The other 36 solutions for the 4 by 2 site which cannot be derived from previous solutions are shown in figure 9.2.

![Diagram showing 36 solutions for a 4 by 2 site](image)

**Figure 9.2:** The solutions for the 4 by 2 site which cannot be derived by null adaptation.

The major problem of null adaptation is that after applying the previous solutions to the new design environment, the semantics of the previous solutions change, for example, in some cases, the number of gardens changes. Because null adaptation deals only with the syntactic level of a solution, it does not guarantee the maintenance of the design semantics. However, null adaptation does provide a shortcut for solving new problems. Additional techniques, e.g., analogical means-ends analysis, can be incorporated easily to derive more complete and semantically correct adaptations.

### 9.1.2 Case-Based Reasoning in BAU

In reality, human designers do not rely on simple null adaptation techniques to solve new design problems, rather, they use knowledge extracted from design experience to tackle new design situations. One of the best known structural adaptation techniques is called *parameterized solutions*, it compares the parameters of old and new problem descriptions when a previous case is retrieved for a new situation. The differences between the two problem descriptions are then used to make
appropriate modifications to the solution parameters. Such parameterized solutions can be thought of as products of a type of interpolation technique that is similar to the method used for finding logarithms in a table using only basic arithmetic [Riesbeck 89].

9.1.2.1 Comparing RB-BAU and CB-BAU

The concept of parameterized solutions enables us to develop the BAU system with the case-based (CB-BAU) approach rather than the initial rule-based (RB-BAU) approach. Figure 9-3 shows the major differences between these two approaches. Both of RB-BAU and CB-BAU contain a design generator and a design filter (see the detailed discussion of generator and filter in Section 7.4.2).

A. The framework of RB-BAU.

B. The framework of CB-BAU.

Figure 9-3: A comparison between RB-BAU and CB-BAU frameworks.
There are three major differences between RB-BAU and CB-BAU. The first difference is that the filter of RB-BAU contains a set of design rules defined by the domain expert, while the filter of CB-BAU contains a set of positive examples provided by the same domain expert. Secondly, in CB-BAU, both the instances in the generator and the cases in the filter have to be interpreted by the "case-parser", before they are compared to each other so that their similarities can be identified. Finally, RB-BAU uses the generate-and-test technique which can only accept or reject generated instances, but CB-BAU uses the "matching" technique to determine how "good" the generated instances are. Thus, RB-BAU explains its answers by providing a trace of the rules used to derive them, and CB-BAU offers a alternative solution to the problem of knowledge acquisition and explanation. Since our primary interest is in different ways of building knowledge-based design systems, the discussion of case modification is not included in this research.

9.1.2.2 CB-BAU: System Overview

Expertise is more like a library of past experience than a set of rules; hence cases better support knowledge transfer and explanation. Generally speaking, a case-based reasoner has a library of cases, a method of storing new cases that allows them to be found again when needed, an indexing scheme that reflects processing that goes on while a case is initially considered, a method for partial matching that allows new cases to be considered in terms of similar ones, and a method of adaptation that allows information gathered from one case to be applied to another [Riesbeck 89]. The expertise that CB-BAU has is more than just a collection of cases, some domain dependent knowledge must be included too. The organizational parts of the CB-BAU case library are:

1. The representative BAU cases which are stored in both 2D and 3D form.
2. A case-parser which contains the rules for analyzing the cases.
3. A case-frame representation for storing the analyzed case data.
4. A case-matcher which contains the rules for matching cases.

The knowledge representation for both the instances in the generator and the cases in the filter is a list (CB-list) which contains two sub-lists. The two sub-lists are related to the list representation of EB-BAU.1 and EB-
BAU.II respectively. The CB-list is converted into machine-readable form which is represented as a case-frame. The desired behavior of the CB-lists are retrieved by the case-parser and put into different fields of the case-frame. Different fields may have different hierarchies of abstraction, and each field represents BAU information according to different classes and ranges.

Table 9-1: A case-frame representation of a BAU case.

<table>
<thead>
<tr>
<th>Frame Attribute-ID</th>
<th>Attribute</th>
<th>Features</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Frame-ID</td>
<td>N/A</td>
<td>Integer</td>
</tr>
<tr>
<td>01</td>
<td>Symbolic Repres.</td>
<td>N/A</td>
<td>List</td>
</tr>
<tr>
<td>02</td>
<td>Numeric Repres.</td>
<td>N/A</td>
<td>List</td>
</tr>
<tr>
<td>03</td>
<td>Interior Space</td>
<td>Space of 2 L-Units</td>
<td>A, Al, I, S, V, T</td>
</tr>
<tr>
<td>04</td>
<td>Hallway &amp; S-Unit</td>
<td>Prep, Para</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>Hallway &amp; U-Unit</td>
<td>Prep, Para</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>Layout Type</td>
<td>N/A</td>
<td>L, M, P, S, T, U, Y</td>
</tr>
<tr>
<td>07</td>
<td>Garden</td>
<td>No. of Gardens</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>08</td>
<td>Main Garden Orie.</td>
<td>E, S, W, N</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>Orientation</td>
<td>Entrance</td>
<td>E, S, W, N</td>
</tr>
<tr>
<td>10</td>
<td>Privacy</td>
<td>E, S, W, N</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>View</td>
<td>E, S, W, N</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Roof Layout</td>
<td>N/A</td>
<td>List</td>
</tr>
<tr>
<td>13</td>
<td>Roof</td>
<td>Main Roof Orient.</td>
<td>E, S, W, N</td>
</tr>
<tr>
<td>14</td>
<td>No. of Roof Orient.</td>
<td>No. of Roof Orient.</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>15</td>
<td>No. of High Roofs</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Relations of Roofs</td>
<td>List</td>
<td></td>
</tr>
</tbody>
</table>

The case-frame is made up of 17 fields, which contain the significant information about a case or an instance of the generator. Each field contains an attribute name, important features to be retrieved, and possible values for each field. The fields No. 0 to 2 of a case-frame contain the general representation information of a retrieved case. The other fields can be divided into 6 important sub-fields, which contain information about the cases: interior space, layout type, garden, orientation, roof layout, and roof configuration.

Each selected case can be converted into a source case-frame, and the instances of the generator can be converted into target case-frames. Both the source and target case-frames will be filled with specific values determined by the rules of the case-parser. The case-parser’s rules are more general than the rules of the RB-BAU filter. How well the source

---

13. Similar usages of case-frames and case-parsers can be found in natural language processing.
case-frame and target instance-frame match is determined by how well corresponding field values in the two frames match.

Table 9-2: Sample scores and weighting factors from a comparison of source and target case frames.

<table>
<thead>
<tr>
<th>Frame Attribute-ID</th>
<th>Scores</th>
<th>Weighting Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>01</td>
<td>0-9</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>0-9</td>
<td>2</td>
</tr>
<tr>
<td>03</td>
<td>0.5</td>
<td>3</td>
</tr>
<tr>
<td>04</td>
<td>0.1</td>
<td>4</td>
</tr>
<tr>
<td>05</td>
<td>0.1</td>
<td>4</td>
</tr>
<tr>
<td>06</td>
<td>0.2, 3, 4, 6</td>
<td>3</td>
</tr>
<tr>
<td>07</td>
<td>0, 0.5, 2</td>
<td>4</td>
</tr>
<tr>
<td>08</td>
<td>0, 1, 2, 3</td>
<td>8</td>
</tr>
<tr>
<td>09</td>
<td>0.1</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>0.1</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>0, 1, 2, 3</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0-9</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>0, 0.5, 2</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>0, 0.5, 2</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>0-5</td>
<td>2</td>
</tr>
</tbody>
</table>

For the matching of the source case- and target instance-frames, an abstraction is developed to evaluate how well two field values match. The abstraction for matching two case-frames, called case-matcher, is also represented in the form of a frame. The user may assign a weighting factor to each field of case-matcher (see table 9-2). A similar approach has been developed by Winston, which employs importance-weighted mappings of roles in one situation onto roles in another [Winston 86]. His method of determining "importance" is based on counting the number of constraints in which the given entity participates. In this research, the definition of the case-frame, the scores and the weighting factor assignments are based on the author's knowledge and preferences. Part of the source code of CB-BAU can be found in Appendix III.

9.1.2.3 Results of CB-BAU

The CB-BAU's generator contains 5675 instances generated by the BAU.II program. A summary of these instances can be found in the Appendix II. To improve the CB-BAU's computational efficiency, the unsatisfactory examples of BAU.II are not included in the following sample runs. The CB-BAU's filter stores the descriptions of 7 representative BAU layouts which include 2D and 3D information. For the purpose of consistency, these seven cases in the case-library are
identical to those shown in Sections 8.1 and 9.1. There are many different ways to select cases from the case library. A user may select up to 7 source cases, and specify the 24 most similar and dissimilar cases to be presented. If only one single source case is selected, then the user can also specify whether the instances, which have layouts identical to that of the source case, should be shown or not. This alternative makes the implementation more comprehensive. A summary of different sample runs is described in Table 9.3. Figures 9-4 through 9-14 illustrate the results of different sample runs of CB-BAU.

A single source case was selected from the case library for Sample run 1-A, 1-B, and 1-C (see figure 9-4, 9-5, and 9-6). The results of 1-A and 1-B present the similar instances in the generator for case No. 148, and 1-C presents the dissimilar instances. The difference between 1-A and 1-B is that the instances in the generator which have the identical 2D layouts as case No. 148 are not included in the computation.

<table>
<thead>
<tr>
<th>Table 9.3: Summary of CB-BAU's sample runs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table" /></td>
</tr>
<tr>
<td>Source Case(s)</td>
</tr>
<tr>
<td>Sample Run 1-A</td>
</tr>
<tr>
<td>Sample Run 1-B</td>
</tr>
<tr>
<td>Sample Run 1-C</td>
</tr>
<tr>
<td>Sample Run 2-A</td>
</tr>
<tr>
<td>Sample Run 2-B</td>
</tr>
<tr>
<td>Sample Run 3-A</td>
</tr>
<tr>
<td>Sample Run 3-B</td>
</tr>
<tr>
<td>Sample Run 4-A</td>
</tr>
<tr>
<td>Sample Run 4-B</td>
</tr>
<tr>
<td>Sample Run 5-A</td>
</tr>
<tr>
<td>Sample Run 5-B</td>
</tr>
</tbody>
</table>

Sample runs 2 to 5 show the similar and the dissimilar instances produced by different case selections from the case-library. The results of sample run 2-A seem at first to be incomplete because only the instances which are similar to case No. 156 are displayed but not those similar to No. 152. That is because the weighting factors that were used upgraded the importance of case No. 156. In the sample run 3, the discriminating weighting factor has been eliminated, allowing the instances similar to case No. 152 and 172 to be displayed evenly. Sample run 4 shows more complicated case-matching.
A. Case 148 is selected as the only source case.

B. The similar target instances (including the similar 2D layouts).

Figure 9-4: The results of CB-BAU’s sample run 1-A.
A. The selected source case is 148.

B. The similar target instances (excluding the similar 2D layouts).

Figure 9-5: The results of CB-BAU’s sample run 1-B.
A. The selected source cases are 152 and 156.

B. The similar target instances.

Figure 9-7: The results of CB-BAU's sample run 2-A.
A. The selected source cases are 152 and 156.

B. The dissimilar target instances.

Figure 9-8: The results of CB-BAU’s sample run 2-B.
A. The selected source cases are 152 and 172.

B. The similar target instances.

Figure 9-9: The results of CB-BAU's sample run 3-A.
A. The selected source cases are 152 and 172.

B. The dissimilar target instances.

Figure 9-10: The results of CB-BAU’s sample run 3-B.
A. The selected source cases are 158, 172, 206 and 216.

B. The similar target instances.

Figure 9-11: The results of CB-BAU's sample run 4-A.
A. The selected source cases are 158, 172, 206 and 216.

B. The dissimilar target instances.

Figure 9-12: The results of CB-BAU's sample run 4-B.
A. All of the cases are selected as source cases.

B. The similar target instances.

Figure 9-13: The results of CB-BAU’s sample run 5-A.
A. All of the cases are selected as source cases.

B. The dissimilar target instances.

Figure 9-14: The results of CB-BAU's sample run 5-B.

109
The most interesting sample run is 5, which selects all of the 7 cases in the case-library. The scores for sample run 5-A range from 1,062.5 to 1,096.5. The result in the lower-left corner of figure 9-13 received the lowest score, and the result in the upper-right corner the highest. The scores for sample run 5-B range from 324.0 to 345.0. The result in the lower-left corner of figure 9-14 returned the highest score, and the one in the upper-right corner the lowest.

Based on the results of sample run 5, it may be concluded that the instance which gets the highest score (1,096.5) is the most representative BAU layout according to the current weighting factors of case-matcher. Concurrently, the instance which gets the lowest score (324.0) is the most insignificant layout based on the current criteria. However, the human designer may sometimes prefer the most insignificant layout.

9.1.2.4 Limitations of the Current CB-BAU

The case-frame in of the current CB-BAU is represented using fixed field attributes, values, and weighting factors. A more user-friendly interface could be developed which would allow the users to modify the field values, change the importance of some fields, and add new fields to the basic case frame. In such a way, the preferences for matching the source and target cases could be adjusted easily. The topic of case-repairing is not included in our current research\textsuperscript{14}, e.g., the repair of retrieved cases which do not fit into a new site well. It requires further research before it can be incorporated into the current system.

9.2 Adaptations of Previous Design Problem Solving Process

One of the major advantages of prefabricated buildings like BAU is the flexibility with which new design elements, materials, rules or environments can be introduced into the existing design system. In such a way, it provides more alternatives for the customers, e.g. build a BAU house with a new garage unit in a 4 by 3 site. However, the problem itself is not entirely new, since some fundamental problem-solving strategies for the new BAU are still the same.

\textsuperscript{14} Repair is similar to adaptation, in that a solution has to be modified to fit a situation. The difference is that adaptation starts with an old situation and a new case and adapts the solution to the new situation, while repair starts with a solution, a failure report, and may be an explanation, and modifies the solution to remove the failure (Riesbeck 89).
The idea here is to store not only a solution with a case, but the planning sequence that constructed the solution. In other words, the stored solution is adapted not by changing it directly, but by re-executing parts of the original solution process. A major potential advantages of rule-based system is called additivity. Ideally, adding a new piece of behavior means adding a new rule or modifying an existing one. Several models of learning have been proposed based on the notion of adding rules to a production system or optimizing the ones that exist [Laird 86, Winston 86]. Additionally, "re-instantiation" in case-based reasoning is also related to this topic, which is a "derivational" adaptation method. It operates not on the original solution, but on the method that was used to generate that solution. In this sense, re-instantiation means replacing a step in a solution by taking the plan that generated that step and executing it again in the context of current situation.

In this context, it is important to ask if a system is capable of re-arranging the new design assumptions and justifications for the new design requirements. A design system may be considered as a system which maintains the true assumptions and processes self-organization. A truth-maintenance system (TMS) [Doyle 79] and an assumption-based truth-maintenance system (ATMS) [de Kleer 86] are developed for representing knowledge about resolving contradictions, updating justifications, and managing the process of retracting and modifying assumptions, which manage temporary inconsistencies during the reasoning process. An ATMS serves as a good starting point for reasoning about the contradictions and consistences when the new design requirements are introduced into an existing design environment. In this sense, ATMS is an important engine for future design support systems [Smithers 89]. However, at this moment, this research does not provide a comprehensive programming example for this problem.

9.3 Summary of Analogical Reasoning

There are two important experiences gained from the development of RB-BAU and CB-BAU: Firstly, the domain expert spent more efforts in the development of RB-BAU, while the knowledge engineer spent more efforts in the development of CB-BAU. Secondly, the required computation power for the RB-BAU is less than for the CB-BAU. The reason is that RB-BAU
has been pre-compiled into several levels of hierarchy, and the unsatisfied solutions have been eliminated in each level of generation. Consequently it is more efficient than CB-BAU which only contains two levels of hierarchies at this moment. It is possible to modify the CB-BAU to make it computationally more efficient.

Incorporation of case-based reasoning and the assumption-based truth maintenance systems are important for the future design support systems in architecture, especially, for the future design critic systems in architectural design education. By using these kinds of tools, students may contrast their design with the "good" design cases in the case library, figure out the differences between the assumptions they made and the assumptions of those good design cases, adjust their design assumptions, and modify the design.
Chapter 10

Conclusions and Future Work

This research intends to extend the current architectural design computation from a sequential reasoning techniques (sequential in the sense of forward and backward reasoning) to lateral thinking. We'll discuss what has been achieved and the future research directions of this study in this chapter.

We started from the study of the underlying structures of existing CAAD programs and knowledge-based design systems, current research results of machine learning, and the characteristics of architectural design. Based on the results of our survey, we examined the limitations of current design computation, and possible solutions to overcome such limitations are proposed. However, due to the difficulties of acquiring synthetic design knowledge, we incorporated causal reasoning to acquire the analytic domain knowledge to derive case prototype. In our implementation, the RB-BAU system demonstrates a traditional approach of building a knowledge-based design system, while CB-BAU takes a case-based approach, and the fundamental ideas for the development of CB-BAU are based on the results of EB-BAU.

10.1 Conclusions

The development of BAU extends the traditional two-dimensional layout synthesis problems into 2.5-D based on the rule-based representation. There are several important lessons that we have learned from BAU:

1. There are some areas of architectural design, in which problems could be clearly defined, goals could be relatively fixed, and phenomena lend themselves to the categories of available theory and techniques.
2. The design requirements for solutions can be expressed by some combination of constraints (design rules) and criteria (design goal).
In this case, the design problem-solving process can be viewed as a state-space search process.

3. In design, an exhaustive search can be avoided through a proper ordering of constraints. This is an important factor of design expertise.

4. Through the study of the architect's problem solving behavior, the acquired design model and strategies can be used in automating parts of the design process.

BAU can be considered as a knowledge compilation task. The experience of BAU leads us to reconsider the fundamental problems of knowledge acquisition in knowledge-based design systems. In real-world design problems, criteria and goals sometime cannot be specified clearly, but the previous design solutions similar to the current problem are available. EB-BAU and CB-BAU emphasize the utilization of solutions to previous problems which are known to be similar to the current problem. EB-BAU acquires knowledge of general principles from collections of individual design cases, and CB-BAU transfers knowledge from one situation to another, similar one.

In EB-BAU, we demonstrated that induction is the process by which we reason from the particular to the general. The primary idea behind induction is a measure of the similarity defined as a proximity measure in a multi-dimensional space spanned by selected object attributes. In spite of generalization, it is never perfect and there is always the danger of losing some quite important information. Nevertheless generalization still provides a good means to resolve the problems mentioned.

Case-based reasoning is an alternative to rule-based reasoning. The use of the case-based approach will be more suitable for the design problems where rules are very difficult to formalize or too large in number. A rule-based system produces nearly optimal answers, but it will be slow to develop. A case-based system produces approximate answers, but it will be quick to develop. Some of the differences between rule-based and case-based systems can be understood by analogy to human problem solving. In case-based system, it seems much more plausible that knowledge is organized so that related items are "near" each other.
10.2 Parting Reflections

The differences between analogy and induction are subtle. Some research started from the study of analogy, but ended up with the discussions of induction, e.g., [Russel 89]; or vice versa, e.g., [Holland 86]. To describe why these changes in focus happen, I'll use a term, dimensions of reasoning, to discuss the relationships between analogy and induction. This concept is not intended to be exhaustive or exclusive, nor it is intended to reflect exactly the human thinking process. However, it may provide another way to understand the motivation behind this research.

In broad terms, from the machine's view (or my personal view), reasoning can be classified into one, two, and multiple dimensions. Each dimension of reasoning has two complementary directions. In the first dimension of reasoning (or sequential reasoning), quite obviously, there are forward and backward reasoning, which are related to deduction and abduction respectively. In the two-dimensional reasoning, one direction is analogy, and the complementary direction is metaphor. In the multi-dimensional reasoning, one direction is induction, and the complementary direction is creativity.

The similarity between analogy and induction is that both of them are associated with positive directions, reasoning in more than one single dimension. For example, we can apply several analogies to a problem simultaneously, or a single instance induction is also possible sometime. From a restricted viewpoint, both analogy and induction can be considered as an extension of logical deduction. However, deductive thinking is considered as truth-preserving, while analogy and induction are considered as plausibility-preserving and falsity-preserving.

Based on the main characteristic of the computer, which is sequential processing, deduction and abduction are well-implemented on the computer through the studies of theorem proving and fuzzy logic. Analogy and induction have been studied since the 1980's in machine learning. However, the two and multi-dimensional reasoning modes in the complementary direction, metaphor and creativity, are still missing in the current computing research so far. The major difficulties to represent metaphor and creativity are that the similarities between the
The last interesting problem for the future research, which I ask myself all the time, is "Can metaphor and creativity be understood and demonstrated by the computer?" Just as Karl Popper said in Section 2.3.2: "P2 is a solution to the current problem P1 which inevitably represents a new problem. This new problem is, in general, not created intentionally."

The end of this thesis is just the beginning of another research. Popper's view is also the epilogue of this thesis.
Appendix I

Source Code of EB-BAU.I

;; PROGRAM EB-BAU.I
;; This program generates the spatial relations of each input BAU list.
;; The input lists are based on the positive training instances of BAU.I.
;; (setq layout-168 '000BBLLSS00)
;; (setq layout-152 '000SLLU0B0)
;; (setq layout-156 '000LSSUL00B)
;; (setq layout-158 '000BLLLS0U0)
;; (setq layout-172 '000000LSU0)
;; (setq layout-206 '000000BLU0)
;; (setq layout-216 '000000LSSBLU)

;; Main function.
(defun EB-BAU.I ()
  ;; Generates the description of unit-relations for the positive training instances.
  ;; Input - nil.
  ;; Output - A description of unit-relations.
  (print "Relations")
  (find-relations layout-168 layout-152 layout-156 layout-158
                  layout-172 layout-206 layout-216)
  (print "Done")
)

;; Interpreter.
(defun find-relations (&rest let)
  ;; Input - A list of positive training instances.
  ;; Output - The spatial relations of each input layout.
  ;; This is a recursive function.
  (print (relations car let))
  (cond (null (cdr let)) (relations (car let))
        (t (intersection (relations (car let))
                        (apply #find-relations (cdr let)))))
)

(defun relations layout)
  ;; Finds the relations between units based on the input BAU list.
  ;; Input - A BAU layout.
  ;; Output - Returns the conjunction of unit-relations.
  (remove-duplicates (relation-of-units 0 layout) :test #reverse-equal)
)

(defun relation-of-units (n layout)
  ;; Inspects the input list, find all the relations between units.
  ;; Input - An initial flag n, and a BAU list.
  ;; Output - A list of unit-relations (duplications are not removed).
  ;; This is a recursive function.
  (cond (> n (length layout)) nil
        (t (append (find-neighbors n layout)
                   (relation-of-units (+ n 1) layout))))
)
(define find-neighbors (u-num layout)
  ; Specifies the neighbors of each BAU cell.
  ; Input: A unit-ID, and a BAU layout.
  ; Output: A list of units which are the neighbors of input unit.
  (cons (u-num 0) (neighbors-of layout u-num 1 3)
    ((= u-num 1) (neighbors-of layout u-num 0 2 4))
    ((= u-num 2) (neighbors-of layout u-num 1 3 5))
    ((= u-num 3) (neighbors-of layout u-num 0 4 6))
    ((= u-num 4) (neighbors-of layout u-num 1 2 5))
    ((= u-num 5) (neighbors-of layout u-num 2 4 8))
    ((= u-num 6) (neighbors-of layout u-num 3 7))
    ((= u-num 7) (neighbors-of layout u-num 4 6 8))
    ((= u-num 8) (neighbors-of layout u-num 5 7))
    (nil))
)

(define neighbors-of (layout u-num n1 n2 &optional n3 n4)
  ; Specifies the neighbors of a particular BAU cell.
  ; Input: A unit-ID, a BAU layout, and a set of neighbor-ID.
  ; Output: A list of units related to neighbor-ID.
  (if (not-equal (nth u-num layout) 0)
    (delete nil
      (list (if (not-equal (nth n1 layout) 0)
        (list (nth u-num layout) 'next-to (nth n1 layout)))
        (if (not-equal (nth n2 layout) 0)
          (list (nth u-num layout) 'next-to (nth n2 layout)))
          (if (not-equal (nth n3 layout) 0)
            (list (nth u-num layout) 'next-to (nth n3 layout)))
            (if (not-equal (nth n4 layout) 0)
              (list (nth u-num layout) 'next-to (nth n4 layout)))))
    (nil))
)

;; Functions for list manipulations.
(define not-equal (n1 n2)
  ; Input: Two symbols n1 n2.
  ; Output: T, if n1 is not equal n2; nil, otherwise.
  (not (eq n1 n2))
)

(define reverse-equal (lst1 lst2)
  ; Input: Two lists.
  ; Output: T, if lst1 = lst2, or lst1 = reverse(lst2); nil, otherwise.
  (if (or (equal lst1 lst2) (equal (reverse lst1) lst2))
    T
    (nil)))

119
### Appendix II

#### Number of Solutions Generated by Rule-RF4.

<table>
<thead>
<tr>
<th>ID No.</th>
<th>SC Num.</th>
<th>ID No.</th>
<th>SC Num.</th>
<th>ID No.</th>
<th>SC Num.</th>
<th>ID No.</th>
<th>SC Num.</th>
<th>ID No.</th>
<th>SC Num.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>37</td>
<td>15</td>
<td>100</td>
<td>29</td>
<td>145</td>
<td>16</td>
<td>183</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>38</td>
<td>15</td>
<td>72</td>
<td>30</td>
<td>110</td>
<td>29</td>
<td>146</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>39</td>
<td>15</td>
<td>73</td>
<td>30</td>
<td>111</td>
<td>29</td>
<td>147</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>40</td>
<td>49</td>
<td>76</td>
<td>30</td>
<td>112</td>
<td>29</td>
<td>148</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>41</td>
<td>49</td>
<td>77</td>
<td>30</td>
<td>113</td>
<td>29</td>
<td>149</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>42</td>
<td>49</td>
<td>78</td>
<td>30</td>
<td>114</td>
<td>29</td>
<td>150</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>43</td>
<td>21</td>
<td>79</td>
<td>30</td>
<td>115</td>
<td>29</td>
<td>151</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>36</td>
<td>44</td>
<td>23</td>
<td>80</td>
<td>15</td>
<td>116</td>
<td>29</td>
<td>152</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>45</td>
<td>17</td>
<td>81</td>
<td>28</td>
<td>117</td>
<td>29</td>
<td>153</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>36</td>
<td>46</td>
<td>16</td>
<td>82</td>
<td>28</td>
<td>118</td>
<td>29</td>
<td>154</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>36</td>
<td>47</td>
<td>16</td>
<td>83</td>
<td>28</td>
<td>119</td>
<td>29</td>
<td>155</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>30</td>
<td>48</td>
<td>16</td>
<td>84</td>
<td>29</td>
<td>120</td>
<td>29</td>
<td>156</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>30</td>
<td>49</td>
<td>16</td>
<td>85</td>
<td>29</td>
<td>121</td>
<td>29</td>
<td>157</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>30</td>
<td>50</td>
<td>15</td>
<td>86</td>
<td>29</td>
<td>122</td>
<td>29</td>
<td>158</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>30</td>
<td>51</td>
<td>16</td>
<td>87</td>
<td>29</td>
<td>123</td>
<td>29</td>
<td>159</td>
<td>29</td>
</tr>
<tr>
<td>16</td>
<td>30</td>
<td>52</td>
<td>16</td>
<td>88</td>
<td>29</td>
<td>124</td>
<td>29</td>
<td>160</td>
<td>29</td>
</tr>
<tr>
<td>17</td>
<td>30</td>
<td>53</td>
<td>25</td>
<td>89</td>
<td>29</td>
<td>125</td>
<td>28</td>
<td>161</td>
<td>29</td>
</tr>
<tr>
<td>18</td>
<td>30</td>
<td>54</td>
<td>29</td>
<td>90</td>
<td>29</td>
<td>126</td>
<td>28</td>
<td>162</td>
<td>29</td>
</tr>
<tr>
<td>19</td>
<td>30</td>
<td>55</td>
<td>12</td>
<td>91</td>
<td>29</td>
<td>127</td>
<td>28</td>
<td>163</td>
<td>29</td>
</tr>
<tr>
<td>20</td>
<td>29</td>
<td>56</td>
<td>12</td>
<td>92</td>
<td>29</td>
<td>128</td>
<td>28</td>
<td>164</td>
<td>29</td>
</tr>
<tr>
<td>21</td>
<td>29</td>
<td>57</td>
<td>29</td>
<td>93</td>
<td>29</td>
<td>129</td>
<td>28</td>
<td>165</td>
<td>29</td>
</tr>
<tr>
<td>22</td>
<td>29</td>
<td>58</td>
<td>29</td>
<td>94</td>
<td>29</td>
<td>130</td>
<td>28</td>
<td>166</td>
<td>29</td>
</tr>
<tr>
<td>23</td>
<td>29</td>
<td>59</td>
<td>29</td>
<td>95</td>
<td>29</td>
<td>131</td>
<td>28</td>
<td>167</td>
<td>29</td>
</tr>
<tr>
<td>24</td>
<td>29</td>
<td>60</td>
<td>29</td>
<td>96</td>
<td>29</td>
<td>132</td>
<td>28</td>
<td>168</td>
<td>29</td>
</tr>
<tr>
<td>25</td>
<td>29</td>
<td>61</td>
<td>29</td>
<td>97</td>
<td>29</td>
<td>133</td>
<td>28</td>
<td>169</td>
<td>16</td>
</tr>
<tr>
<td>26</td>
<td>29</td>
<td>62</td>
<td>29</td>
<td>98</td>
<td>29</td>
<td>134</td>
<td>28</td>
<td>170</td>
<td>16</td>
</tr>
<tr>
<td>27</td>
<td>29</td>
<td>63</td>
<td>29</td>
<td>99</td>
<td>29</td>
<td>135</td>
<td>28</td>
<td>171</td>
<td>16</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>64</td>
<td>29</td>
<td>100</td>
<td>29</td>
<td>136</td>
<td>28</td>
<td>172</td>
<td>16</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>65</td>
<td>12</td>
<td>101</td>
<td>11</td>
<td>137</td>
<td>28</td>
<td>173</td>
<td>16</td>
</tr>
<tr>
<td>30</td>
<td>29</td>
<td>66</td>
<td>12</td>
<td>102</td>
<td>11</td>
<td>138</td>
<td>28</td>
<td>174</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td>29</td>
<td>67</td>
<td>15</td>
<td>103</td>
<td>15</td>
<td>139</td>
<td>28</td>
<td>175</td>
<td>30</td>
</tr>
<tr>
<td>32</td>
<td>29</td>
<td>68</td>
<td>15</td>
<td>104</td>
<td>15</td>
<td>140</td>
<td>28</td>
<td>176</td>
<td>30</td>
</tr>
<tr>
<td>33</td>
<td>29</td>
<td>69</td>
<td>15</td>
<td>105</td>
<td>15</td>
<td>141</td>
<td>28</td>
<td>177</td>
<td>30</td>
</tr>
<tr>
<td>34</td>
<td>29</td>
<td>70</td>
<td>15</td>
<td>106</td>
<td>29</td>
<td>142</td>
<td>28</td>
<td>178</td>
<td>30</td>
</tr>
<tr>
<td>35</td>
<td>29</td>
<td>71</td>
<td>15</td>
<td>107</td>
<td>29</td>
<td>143</td>
<td>28</td>
<td>179</td>
<td>30</td>
</tr>
<tr>
<td>36</td>
<td>29</td>
<td>72</td>
<td>15</td>
<td>108</td>
<td>29</td>
<td>144</td>
<td>28</td>
<td>180</td>
<td>30</td>
</tr>
</tbody>
</table>

Total Number of 3D Significant Configurations: 5675.

**ID No.:** ID No. of 2D configurations generated by BAU.I.

**SC Num.:** Number of 3D significant configurations generated by BAU.II.

---

[The rest of the paper follows, including the section titled "Results of 3D Significant Configurations by Rule-RF4." and the conclusion page "Section 7. Conclusion." These sections are not transcribed here as they are not relevant to the provided task.]
Appendix III

Source Code of CB-BAU

(defun cb-bau (/ source-list source-case-frame sample-size n
              start-case target-frame good-bad)
  ; Initialise.
  (setq source-list nil)
  (setq source-case-frame nil)
  (setq sample-size nil)
  (setq n 0)
  ; Specify the number of results.
  (setq "result-num" 24)
  ; Read CB.
  (setq source-list
        (read (getstring T "Source cases [1..7]: ")))
  ; Determines the number of source cases.
  (if (> (length source-list) 1) (find-targets source-list))
  ; Decide the nearest or farthest results to be displayed.
  (setq good-bad (getstring "Nearnest neighbors, or farrest neighbors [0, 1]: "))
  ; Show results.
  (show-results "result-far-list")
  (show-results "result-near-list")
)

(defun find-targets (list / sample-size m n)
  ; CBR of BAU based on multi-cases.
  ; Determines the sample size.
  (setq sample-size (getint "Sample size [0 .. 5674]: ")
  (setq m (getint "Start from [0 .. 5674]: ")
  (setq n 0)
  (while (< n sample-size) (< m 5674))
    (print m)
    (setq tmp (nth m "scores")
    (insert-result-list m list tmp)
    (setq m (+ m 1))
    (setq n (+ n 1))
  )
)
(defun find-target-1 (num / sample-size m n layout-type)
  ;; CBR of BAI based on one single case.
  (cond ((= num 1) (seq layout-type 'S))
        ((= num 2) (seq layout-type 'Y))
        ((= num 3) (seq layout-type 'T))
        ((= num 4) (seq layout-type 'W))
        ((= num 5) (seq layout-type 'L))
        ((= num 6) (seq layout-type 'U))
        ((= num 7) (seq layout-type 'P))
        (t nil))
  ;; Determines the sample size.
  (seqq sample-size (getint "Sample size [0 - 5674]: ")
   (seqq m (getint "Start from [0 - 5674]: ")
   (seqq flag (getint '([0 all]) (different layout types only): ")
   (seqq n 0)
   (while (and (<= n sample-size) (<= m 5674))
     (print m)
     (print n)
     (seqq temp (nth m *scores*))
     (seqq result-list (insert-result-lst-1 m num temp))
     (if (= flag 0) (insert-result-lst-1 m num temp))
     (if (= flag 1) (if (not (equal layout-type (last temp)))
                      (insert-result-lst-1 m num temp))
     (seqq m (+ m 1))
     (seqq n (+ n 1)))
  )
)

(defun near ()
  ;; Displays the most similar results.
  (initialize)
  (show-results *result-near-lst*)
)

(defun far ()
  ;; Displays the most dissimilar results.
  (initialize)
  (show-results *result-far-lst*)
)

(defun insert-result-lst (num lst data / tmp)
  ;; Updating the score list for multi-cases comparisons.
  (seqq temp (add-scores lst data))
  (print (list temp num))
  (seqq *result-near-lst* (insertion-sort-nc (list temp num) *result-near-lst*))
  (seqq *result-far-lst* (insertion-sort-dc (list temp num) *result-far-lst*))
)

(defun insert-result-lst-1 (num n data / tmp)
  ;; Updating the score list for one single case comparisons.
  (seqq temp (nth n data))
  (print (list temp num))
  (seqq *result-near-lst* (insertion-sort-nc (list temp num) *result-near-lst*))
  (seqq *result-far-lst* (insertion-sort-dc (list temp num) *result-far-lst*))
)

(defun add-scores (lst data)
  ;; Adds the scores of each attribute of frames matched.
  (apply + (mapcar (lambda (x) (nth x data)) lst))
)

122
LIST MANIPULATIONS

(defun insertion-sort-ac (a lst)
    ; Insertion sort based on an accedent order.
    (cond ((null lst) (list a))
      ((< (car a) (car lst)) (cons (car lst) (insertion-sort-ac a (cdr lst))))))

(defun insertion-sort-dc (a lst)
    ; Insertion sort based on a deecedent order.
    (cond ((null lst) (list a))
      (>= (car a) (car lst)) (cons (car lst) (insertion-sort-dc a (cdr lst)))))))

SOURCE LISP

Each case list contains a 2D layout, 2D ID, and 3D layout and 3D ID.

(seqq *C147* (0 0 0 T W (U (H NOT-S) (K E) (B W)) L L S 0 0 147)
  0 0 0 (R E) (B W) (R E +) (R E) (R W) 0 0 0 130))

(seqq *C151* (0 0 0 S L L (U (H NOT-N) (K W) (B E)) 0 (T W) 0 0 151)
  0 0 0 (R S) (R N) (R N +) (R N) 0 (R N) 0 0 511))

(seqq *C155* (0 0 0 L S (U (H NOT-N) (K E) (B W)) L 0 0 (T W) 155)
  0 0 0 (R S) (R S) (R S +) 0 0 0 (R N) 0 0 343))

(seqq *C157* (0 0 0 (T W) 0 L L S (U (H NOT-E) (K S) (B N)) 0 0 157)
  0 0 0 (R S) 0 (R W) (R W +) (R W) 0 0 426))

(seqq *C171* (0 0 0 (T W) 0 0 L S (U (H NOT-T) (K E) (B W)) L 171)
  0 0 0 (R W) 0 0 (R N +) (R N) (R N) (R N) 767))

(seqq *C171* (0 0 0 L S (U (H NOT-N) (K W) (B E) 0 0 205)
  0 0 0 (R N +) 0 (R S) (R N) (R S +) 0 0 426))

(seqq *C215* (0 0 0 0 L S (T S) L (U (H NOT-N) (K W) (B E) 215)
  0 0 0 (R E +) (R E) (R W) (R E +) (R E) 32)))

123
(defun matcher (/ source-list source-case-frame sample-size n)
  start-case target-frame)
"Initialization of source and target-frame."
  (setq source nil)
  (setq case-frame nil)
  n 0)
  (setq source-list
    (read (getstring T "Source cases (1..?):")))
  (if (member 7 source-list) (setq source (cons "C115" source)))
  (if (member 6 source-list) (setq source (cons "C205" source)))
  (if (member 5 source-list) (setq source (cons "C171" source)))
  (if (member 4 source-list) (setq source (cons "C157" source)))
  (if (member 3 source-list) (setq source (cons "C155" source)))
  (if (member 2 source-list) (setq source (cons "C151" source)))
  (if (member 1 source-list) (setq source (cons "C147" source)))
  (setq case-frame (mapcar 'make-case-frame source))
"Determines the sample size."
  (setq sample-size (getint "Sample size [0 - 5674]:")
  (setq start-case (getint "Start from [0 - 5674]:")
  (while (and (< n sample-size) (> (< n start-case) sample-size))
    (setq score 0)
    (foreach x-case-frame (compare-frames x target-frame))
    (print (list "score" (car target-case) (+ n start-case))
    (setq n (+ n 1))))
"Case-frame matching."
  (defun weighting-factors ()
  "Assign weighting-factors for each field of the matching-frame."
  (setq W1 1 W2 2 W3 10 W4 4 W5 4 W6 8 W7 4 W8 8 W9 10 W10 7 W11 5 W12 1 W13 3 W14 6 W15 8 W16 2)
  (defun compare-frames (frame1 frame2)

  "Compared two case-frames, and derives a score."
  (weighting-factors)
  (setq score 0)
  (setq s1 (match-1-frame (nth 1 frame1) (nth 1 frame2))
  s2 (match-2-frame (nth 2 frame1) (nth 2 frame2))
  s3 (match-3-frame (nth 3 frame1) (nth 3 frame2))
  s4 (match-4-frame (nth 4 frame1) (nth 4 frame2))
  s5 (match-5-frame (nth 5 frame1) (nth 5 frame2))
  s6 (match-6-frame (nth 6 frame1) (nth 6 frame2))
  s7 (match-7-frame (nth 7 frame1) (nth 7 frame2))
  s8 (match-8-frame (nth 8 frame1) (nth 8 frame2))
  s9 (match-8-frame (nth 9 frame1) (nth 9 frame2))
  s10 (match-10-frame (nth 10 frame1) (nth 10 frame2))
  s11 (match-11-frame (nth 11 frame1) (nth 11 frame2))
  s12 (match-12-frame (nth 12 frame1) (nth 12 frame2))
  s13 (match-13-frame (nth 13 frame1) (nth 13 frame2))
  s14 (match-14-frame (nth 14 frame1) (nth 14 frame2))
  s15 (match-15-frame (nth 15 frame1) (nth 15 frame2))
  s16 (match-16-frame (nth 16 frame1) (nth 16 frame2))
  (setq score (+(< s1 s2 s3 s4 s5 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16) score))
)
(defun match-1-frame (last1 last2)
  ;; Matches the 1st field of two case-frames, and calculate the scores.
  (* (apply + (mapcar 'lambda (x y) (if (equal x y) 1 0)) last1 last2)) "W1")
)

(defun match-2-frame (last1 last2)
  ;; Matches the 2nd field of two case-frames, and calculate the scores.
  (* (apply + (mapcar 'lambda (x y) (if (= x y) 1 0)) last1 last2)) "W2")
)

(defun match-3-frame (last1 last2)
  ;; Matches the 3rd field of two case-frames, and calculate the scores.
  (* (cond (and (equal last1 'no-good) (equal last2 'no-good)) 0
            (and (member last1 'A1) (member last2 'A1)) 5
            (and (member last1 'I5) (member last2 'I5)) 5
            (and (member last1 'A1T) (member last2 'A1T)) 5
            (t 0)) "W3")
)

(defun match-4-frame (last1 last2)
  ;; Matches the 4th field of two case-frames, and calculate the scores.
  (* (if (equal last1 last2) 1 0)) "W4")
)

(defun match-5-frame (last1 last2)
  ;; Matches the 5th field of two case-frames, and calculate the scores.
  (* (if (equal last1 last2) 1 0)) "W5")
)

(defun match-6-frame (last1 last2)
  ;; Matches the 6th field of two case-frames, and calculate the scores.
  (* (cond (equal last1 last2) 10
            (and (member last1 'P-I) (member last2 'P-I)) 4
            (and (member last1 'P-B) (member last2 'P-B)) 2
            (and (member last1 'P-T) (member last2 'P-T)) 2
            (and (member last1 'P-U) (member last2 'P-U)) 6
            (and (member last1 'P-W) (member last2 'P-W)) 4
            (and (member last1 'P-Y) (member last2 'P-Y)) 4
            (and (member last1 'L-I) (member last2 'L-I)) 2
            (and (member last1 'L-B) (member last2 'L-B)) 4
            (and (member last1 'L-U) (member last2 'L-U)) 4
            (and (member last1 'L-W) (member last2 'L-W)) 6
            (and (member last1 'L-Y) (member last2 'L-Y)) 3
            (and (member last1 'S-I) (member last2 'S-I)) 4
            (and (member last1 'S-B) (member last2 'S-B)) 4
            (and (member last1 'S-U) (member last2 'S-U)) 4
            (and (member last1 'S-W) (member last2 'S-W)) 2
            (and (member last1 'S-Y) (member last2 'S-Y)) 6
            (and (member last1 'T-I) (member last2 'T-I)) 6
            (and (member last1 'T-B) (member last2 'T-B)) 2
            (and (member last1 'T-U) (member last2 'T-U)) 2
            (and (member last1 'T-W) (member last2 'T-W)) 2
            (and (member last1 'T-Y) (member last2 'T-Y)) 2
            (and (member last1 'U-I) (member last2 'U-I)) 6
            (and (member last1 'U-B) (member last2 'U-B)) 3
            (and (member last1 'U-U) (member last2 'U-U)) 6
            (and (member last1 'U-W) (member last2 'U-W)) 6
            (t 0)) "W6")
)

125
(defun match-7-frame (lst1 lst2 / tmp)
  ;; Matches the 7th field of two case-frames, and calculate the scores.
  (setq tmp (abs (- lst1 lst2)))
  (cond ((and (>= tmp 0) (< tmp 0.5)) (setq tmp 2))
    ((and (> tmp 0.5) (< tmp 1.5)) (setq tmp 0.5))
    ((and (> tmp 1.5) (< tmp 2.5)) (setq tmp 0))
    (t (setq tmp 0)))
  (* tmp "W7")
)

(defun match-8-frame (lst1 lst2)
  ;; Matches the 8th field of two case-frames, and calculate the scores.
  (* (length (intersection lst1 lst2)) "W8")
)

(defun match-9-frame (lst1 lst2)
  ;; Matches the 9th field of two case-frames, and calculate the scores.
  (* (if (equal lst1 lst2) 1 0) "W9")
)

(defun match-10-frame (lst1 lst2)
  ;; Matches the 10th field of two case-frames, and calculate the scores.
  (* (if (equal lst1 lst2) 1 0) "W10")
)

(defun match-11-frame (lst1 lst2)
  ;; Matches the 11th field of two case-frames, and calculate the scores.
  (* (length (intersection lst1 lst2)) "W11")
)

(defun match-12-frame (lst1 lst2)
  ;; Matches the 12th field of two case-frames, and calculate the scores.
  (* (apply 'a (mapcar
                        (lambda (x y) (if (and (not (x y))
                                        (not (y x)) (equal x y)) 1 0)) lst1 lst2)) "W12")
)

(defun match-13-frame (lst1 lst2)
  ;; Matches the 13th field of two case-frames, and calculate the scores.
  (* (length (intersection lst1 lst2)) "W13")
)

(defun match-14-frame (lst1 lst2 / tmp)
  ;; Matches the 14th field of two case-frames, and calculate the scores.
  (setq tmp (abs (- lst1 lst2)))
  (cond ((> tmp 0) (setq tmp 2))
    ((> tmp 1) (setq tmp 0.5))
    (t (setq tmp 0)))
  (* tmp "W14")
)

(defun match-15-frame (lst1 lst2 / tmp)
  ;; Matches the 15th field of two case-frames, and calculate the scores.
  (setq tmp (abs (- lst1 lst2)))
  (cond ((> tmp 0) (setq tmp 2))
    ((> tmp 1) (setq tmp 0.5))
    (t (setq tmp 0)))
  (* tmp "W15")
)
(defun match-16-frame (lst1 lst2)
  ; Matches the 16th field of two case-frames, and calculate the scores.
  (* (length (intersection lst1 lst2)) "W16")
)

;; List manipulations.

(defun intersection (lst1 lst2)
  ; Produces the intersection of two lists.
  (cond ((null lst1) nil)
        ((member (car lst1) lst2)
          (cons (car lst1) (intersection (cdr lst1) lst2)))
        (t (intersection (cdr lst1) lst2)))
)
;; PROGRAM  CASE-PARSER.LSP
;; This program parses an input BAU case into a case-frame form.

;; Main Function.
(defun make-case-frame (lst)
  (case_parsers lst)
  (list *frame-id* ; nth 00
        *symbol-configuration* ; nth 01
        *numeric-configuration* ; nth 02
        *living-inter* ; nth 03
        *sleeping-inter* ; nth 04
        *utility-inter* ; nth 06
        *layout-type* ; nth 06
        *mem-garden* ; nth 07
        *garden-orient* ; nth 08
        *entrance-orient* ; nth 09
        *privacy* ; nth 10
        *view-orient* ; nth 11
        *verd* ; nth 12
        *northeast-orient* ; nth 13
        *north-orient* ; nth 14
        *northeast-orient* ; nth 15
        *roof-relations* ; nth 16
  )
)

;; Case-frame parser.
(defun case_parsers (lst)
  (make-frame-id lst)
  (make-configuration-frame (car lst))
  (make-internal-frame lst)
  (make-outdoor-frame *numeric-configuration*)
  (make-orient-frame *symbol-configuration* (car lst))
  (make-roof-frame (cadr lst))
)

;; Makes case-frame elements.
(defun make-frame-id (lst) (setq *frame-id* (list (last (car lst)) (last (cadr lst))))
)

(defun make-configuration-frame (lst)
  (setq *symbol-configuration* (first n_elements 9 (mapcar 'decode-configuration-1 lst)))
  (setq *numeric-configuration* (first n_elements 9 (mapcar 'decode-configuration-2 lst)))
)

(defun make-internal-frame (lst / tmp)
  (setq tmp *symbol-configuration*)
  (setq *living-inter* (decode-internal-1 tmp (cadr lst))
    *sleeping-inter* (decode-internal-2 tmp lst)
    *utility-inter* (decode-internal-3 tmp lst))
)

(defun make-outdoor-frame (lst)
  (setq *layout-type* (decode-outdoor-2 lst (decode-outdoor-1 lst))
    *mem-garden* (cond (equal *layout-type* 'L) 1)
    (equal *layout-type* 'M) 2)
    (equal *layout-type* 'P) 1.5)
    (equal *layout-type* 'S) 2)
    (equal *layout-type* 'T) 2)
    (equal *layout-type* 'U) 1.5)
    (equal *layout-type* 'T) 3)
    (t nil))
  (setq *garden-orient* (decode-outdoor-3 lst *layout-type*)))

128
(defun make-orient-frame (lst1 lst2)
  (seq "entrace-orient*" (decode-orient-1 lst1 lst2)
    *privacy* (decode-orient-2 lst1)
    *view-orient* *garden-orient*))

(defun make-roof-frame (lst)
  (seq *roof* (mapcar 'decode-roof-0 (first_n_elements 9 lst))
    *major-roof-orient* (decode-roof-1 *roof*)
    *no-roof-orient* (decode-roof-2 *roof*)
    *no-high-orient* (decode-roof-3 *roof*)
    *roof-relations* (decode-roof-4 *roof*)))

;; Decodes a case.
(defun decode-configuration-1 (a)
  (cond (null a) nil
        (atom a) a
        (listp a) (car a)
        (t nil)))

;; Decodes configurations.
(defun decode-configuration-2 (a)
  (if (numberp a) 0 1))

;; Decodes interior spaces.
(defun decode-interior-1 (lst1 lst2 / tmp roof1 roof2)
  (seq tmp (find-element-2 location 1 lst1))
  (seq roof1 (nth (car tmp) lst2)
     roof2 (nth (cadr tmp) lst2)
     (identify-interior-1 tmp roof1 roof2)))

(defun decode-interior-2 (lst1 lst2 / tmp1 tmp2 unit rf)
  (seq tmp1 (find-element-location 8 lst1))
  (seq tmp2 (find-element-location 'U lst1))
  (seq unit (nth 1 (nth tmp2 (car lst1)))
     rf (nth 1 (nth tmp1 (cadr lst1)))
     (identify-interior-2 unit rf)))

(defun decode-interior-3 (lst1 lst2 / tmp unit rf)
  (seq tmp (find-element-location 'U lst1))
  (seq unit (nth 1 (nth tmp (car lst1)))
     rf (nth 1 (nth tmp (cadr lst1)))
     (identify-interior-2 unit rf)))

;; Identifies the interior relations.
(defun identify-interior-1 (id r1 rf2 / tmp orient)
  (seq tmp (- (cadr id) (car id)))
  (cond (= tmp 1) (E-inter-living-roof r1 rf2)
        (= tmp 3) (N-inter-living-roof r1 rf2)
        (t nil)))

129
(defun E-inter-living-roof (rf1 rf2)
  (cond (and (equal rf1 'E) (equal rf2 'W)) 'V
        (and (equal rf1 'S) (equal rf2 'S)) 'I
        (and (equal rf1 'E) (equal rf2 'R E +)) 'A
        (and (equal rf1 'W) (equal rf2 'R W +)) 'T
        (and (equal rf1 'N) (equal rf2 'R N)) 'I
        (and (equal rf1 'I) (equal rf2 'R W +)) 'T
        (and (equal rf1 'R E) (equal rf2 'R E)) 'T
        (and (equal rf1 'R W) (equal rf2 'R W)) 'T
        (and (equal rf1 'R S') (equal rf2 'R S')) 'T
        (and (equal rf1 'R N') (equal rf2 'R N')) 'T
        (and (equal rf1 'R E +) (equal rf2 'R E +)) 'A
        (and (equal rf1 'R W +) (equal rf2 'R W +)) 'A
        (and (equal rf1 'R S) (equal rf2 'R S)) 'A
        (and (equal rf1 'R N) (equal rf2 'R N)) 'A
        (and (equal rf1 'R E +) (equal rf2 'R E +)) 'A
        (and (equal rf1 'R W +) (equal rf2 'R W +)) 'A
        (and (equal rf1 'R S') (equal rf2 'R S')) 'A
        (and (equal rf1 'R N') (equal rf2 'R N')) 'A
        (and (equal rf1 'E) (equal rf2 'E)) 'I
        (and (equal rf1 'S) (equal rf2 'S)) 'I
        (and (equal rf1 'I) (equal rf2 'I)) 'I
        (and (equal rf1 'W) (equal rf2 'W)) 'I
        (and (equal rf1 'N) (equal rf2 'N)) 'I
        (and (equal rf1 'R E) (equal rf2 'R E)) 'I
        (and (equal rf1 'R W) (equal rf2 'R W)) 'I
        (and (equal rf1 'R S) (equal rf2 'R S)) 'I
        (and (equal rf1 'R N) (equal rf2 'R N)) 'I
        (and (equal rf1 'R E +) (equal rf2 'R E +)) 'I
        (and (equal rf1 'R W +) (equal rf2 'R W +)) 'I
        (and (equal rf1 'R S') (equal rf2 'R S')) 'I
        (and (equal rf1 'R N') (equal rf2 'R N')) 'I
        (t 'no-good))
)

(defun N-inter-living-roof (rf1 rf2)
  (cond (and (equal rf1 'E) (equal rf2 'E)) 'I
        (and (equal rf1 'E) (equal rf2 'W)) 'I
        (and (equal rf1 'E) (equal rf2 'S)) 'I
        (and (equal rf1 'E) (equal rf2 'N)) 'I
        (and (equal rf1 'S) (equal rf2 'S)) 'I
        (and (equal rf1 'S) (equal rf2 'N)) 'I
        (and (equal rf1 'S) (equal rf2 'W)) 'I
        (and (equal rf1 'S) (equal rf2 'E)) 'I
        (and (equal rf1 'N) (equal rf2 'N)) 'I
        (and (equal rf1 'N) (equal rf2 'W)) 'I
        (and (equal rf1 'N) (equal rf2 'E)) 'I
        (and (equal rf1 'W) (equal rf2 'W)) 'I
        (and (equal rf1 'W) (equal rf2 'E)) 'I
        (and (equal rf1 'E) (equal rf2 'E)) 'I
        (and (equal rf1 'E) (equal rf2 'S)) 'I
        (and (equal rf1 'E) (equal rf2 'N)) 'I
        (and (equal rf1 'S) (equal rf2 'S)) 'I
        (and (equal rf1 'S) (equal rf2 'N)) 'I
        (and (equal rf1 'S) (equal rf2 'W)) 'I
        (and (equal rf1 'S) (equal rf2 'E)) 'I
        (and (equal rf1 'N) (equal rf2 'N)) 'I
        (and (equal rf1 'N) (equal rf2 'W)) 'I
        (and (equal rf1 'N) (equal rf2 'E)) 'I
        (t 'no-good))
)

(defun identify-interior-2 (unit rf)
  (cond (and (equal unit 'H NOT E) (or (equal rf E) (equal rf W) 'para)
            (equal unit 'H NOT E) (or (equal rf S) (equal rf N) 'para)
            (equal unit 'H NOT S) (or (equal rf E) (equal rf W) 'para)
            (equal unit 'H NOT S) (or (equal rf S) (equal rf N) 'para)
            (equal unit 'H NOT W) (or (equal rf E) (equal rf W) 'para)
            (equal unit 'H NOT W) (or (equal rf S) (equal rf N) 'para)
            (equal unit 'H NOT N) (or (equal rf E) (equal rf W) 'para)
            (equal unit 'H NOT N) (or (equal rf S) (equal rf N) 'para)
            (t nil))
)

(defun decode-outdoor-1 (lat)
  (cond (eq (nth 1 lat) (nth 5 lat)) (eq (nth 2 lat) (nth 4 lat))
        (eq (nth 3 lat) (nth 6 lat)) (eq (nth 4 lat) (nth 8 lat)))
)

(= (nth 1 lat) (nth 5 lat) (nth 2 lat) (nth 4 lat) (nth 3 lat) (nth 6 lat) (nth 7 lat) (nth 8 lat))

;; Decodes the outdoor space.

(defun find-garden-U (lat / tmp)
  (setq tmp (find-row-column 0 lat))
  (cond ((equal tmp 'C1) 'W)
        ((equal tmp 'C3) 'E)
        ((equal tmp 'R1) 'S)
        ((equal tmp 'R3) 'N)
        (t nil))
)

(defun find-garden-Y (lat)
  (cond ((or (and (= (nth 0 lat) 0) (= (nth 1 lat) 0))
             (and (= (nth 2 lat) 0) (= (nth 1 lat) 0)))))
    (for (and (= (nth 3 lat) 0) (= (nth 3 lat) 0))
             (and (= (nth 4 lat) 0) (= (nth 6 lat) 0)))
    (for (and (= (nth 7 lat) 0) (= (nth 6 lat) 0))
             (and (= (nth 5 lat) 0) (= (nth 8 lat) 0))
     (t nil))
)

;;; DECODE ORIENTATION

(defun decode-orient-1 (lat1 lat2)
  (cdr (nth (find-element-location 'T lat1) lat2))
)

(defun decode-orient-2 (lat)
  (cond ((or (and (equal (nth 0 lat) 'S) (equal (nth 1 lat) 'U))
                (and (equal (nth 0 lat) 'U) (equal (nth 0 lat) 'U)))
               'S)
        (for (and (equal (nth 0 lat) 'S) (equal (nth 2 lat) 'U))
             (and (equal (nth 0 lat) 'U) (equal (nth 1 lat) 'U)))
        (for (and (equal (nth 0 lat) 'S) (equal (nth 3 lat) 'U))
             (and (equal (nth 0 lat) 'U) (equal (nth 0 lat) 'U)))
        (for (and (equal (nth 0 lat) 'S) (equal (nth 6 lat) 'U))
             (and (equal (nth 0 lat) 'U) (equal (nth 6 lat) 'U)))
        (for (and (equal (nth 0 lat) 'S) (equal (nth 7 lat) 'U))
             (and (equal (nth 0 lat) 'U) (equal (nth 7 lat) 'U)))
        (for (and (equal (nth 0 lat) 'S) (equal (nth 8 lat) 'U))
             (and (equal (nth 0 lat) 'U) (equal (nth 8 lat) 'U)))
        (t nil))
)

;;; Decodes the roof configurations.
(defun decode-roof-0 (a)
  (cond ((equal a '(RE) '(RE 0))
         (equal a 'R S) '(R S 0))
         (equal a '(RE W) '(RE W 0))
         (equal a '(RN) '(RN 0))
         (equal a 'RE +) '(R E 1))
         (equal a '(RS +) '(RS 1))
         (equal a '(R W +) '(R W 1))
         (equal a '(RN +) '(RN 1))
         (t nil))
)
(defun symbol-equal (x y)
  (and (not (numberp x)) (not (numberp y)) (equal x y))
)

(defun find-element-location (a let)-> (length let) (length (member a let)))

(defun find-element-2-location (a let)
  (list (- (length let)) (length (member a let)))
)

(defun find-row-column (bw let / tmp)
  (setq tmp nil)
  (if (and (= (nth 0 let) bw) (= (nth 1 let) bw) (= (nth 2 let) bw))
      (setq tmp (cons 't tmp)))
  (if (and (= (nth 3 let) bw) (= (nth 4 let) bw) (= (nth 5 let) bw))
      (setq tmp (cons 't2 tmp)))
  (if (and (= (nth 6 let) bw) (= (nth 7 let) bw) (= (nth 8 let) bw))
      (setq tmp (cons 't3 tmp)))
  (if (and (= (nth 0 let) bw) (= (nth 3 let) bw) (= (nth 6 let) bw))
      (setq tmp (cons 'c1 tmp)))
  (if (and (= (nth 1 let) bw) (= (nth 4 let) bw) (= (nth 7 let) bw))
      (setq tmp (cons 'c2 tmp)))
  (if (and (= (nth 2 let) bw) (= (nth 5 let) bw) (= (nth 8 let) bw))
      (setq tmp (cons 'c3 tmp)))
  (if (and (= (nth 0 let) bw) (= (nth 4 let) bw) (= (nth 8 let) bw))
      (setq tmp (cons 'd1 tmp)))
  (if (and (= (nth 2 let) bw) (= (nth 4 let) bw) (= (nth 6 let) bw))
      (setq tmp (cons 'd2 tmp)))
  tmp
)

(defun remove-duplicate (let)
  (cond ((null let) nil)
        ((member (car let) (cdr let)) (remove-duplicate (cdr let)))
        (t (cons (car let) (remove-duplicate (cdr let))))))

;; Generalizes the row relations.
;; This part of code is similar to EBAU.II.LISP.
(defun unify-orient (list / a b c)
  (setq a (encode-orient (nth 1 (car list))))
  (setq b (encode-orient (nth 2 (car list))))
  (setq c (encode-orient (last list)))
  (cond (= a b) (setq b (decode-orient (- b 1)) c (decode-orient (- c 1)))
        (= a 2) (setq b (decode-orient (- b 2)) c (decode-orient (- c 2)))
        (= a 3) (setq b (decode-orient (- b 3)) c (decode-orient (- c 3)))
        (t nil))
  (list (list (list R E (last (car list))))) (last R b (last (cdr list)))))
)

(defun encode-orient (orient)
  (cond ((equal orient 'E) 1)
        ((equal orient 'W) 2)
        ((equal orient 'N) 4)
        (t nil))
)

134
(defun ror-6 (list)
  (if (listp (nth 7 list))
      (setq *ror* (cons (list (nth 6 list) (nth 7 list) 'E) *ror*))
      (if (listp (nth 3 list))
          (setq *ror* (cons (list (nth 6 list) (nth 3 list) 'S) *ror*))
          ))
)

(defun ror-7 (list)
  (if (listp (nth 4 list))
      (setq *ror* (cons (list (nth 7 list) (nth 4 list) 'S) *ror*))
      (if (listp (nth 6 list))
          (setq *ror* (cons (list (nth 7 list) (nth 6 list) 'W) *ror*))
          (if (listp (nth 8 list))
              (setq *ror* (cons (list (nth 7 list) (nth 8 list) 'E) *ror*))
              ))
  ))

(defun ror-8 (list)
  (if (listp (nth 7 list))
      (setq *ror* (cons (list (nth 8 list) (nth 7 list) 'W) *ror*))
      (if (listp (nth 5 list))
          (setq *ror* (cons (list (nth 8 list) (nth 5 list) 'S) *ror*))
          ))
  )
References

[Adams 86] Adams, James I.  

[Akin 86] Akin, Ömer.  
*Psychology of Architectural Design*.  

[Akin 88] Akin, Ömer.  
*Expertise of the Architect*.  
Tech. Rept. EDRC-88-09-88, Engineering Design Research Center,  

[Alexander 77] Alexander, Christopher; Ishikawa, S. and Silverstein, M.  
*A Pattern Language*.  

*Notes on the Synthesis of Form*.  

[Amsler 86] Amsler Mark (ed.).  
*The Languages of Creativity*.  

[Anderson 83] Anderson, J. R.  
The Architecture of Cognition.  

[Anderson 86] Anderson, John R.  
"Knowledge Compilation: The General Learning Mechanism".  
In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (eds),  

*Visual Thinking*.  

[Barr 81] Barr Avron and Feigenbaum Edward A.  
The Handbook of Artificial Intelligence, Volume I.  
Morgan Kaufmann, Los Altos, CA, 1981.

[Bijl 85] Bijl, Aart.  
"Graphical Input: Can Computers Understand People?".  

[Bobrow 84] Bobrow, D.  
"Qualitative Reasoning about Physical Systems: an Introduction".  
*Artificial Intelligence*, 24(1).

[Boden 77] Boden, Margaret A.  
*Artificial Intelligence and Natural Man*.  

[Boose 89] Boose, John R. and Gaines, Brian R.  
"Knowledge Acquisition for Knowledge-Based Systems: Notes on the State-of-the-Art".  

[Borning 79] Borning, Alan H.  
*Thing Lab: A Constraint-Oriented Simulation Laboratory*.  
Tech. Rept. SSL-79-3, Xerox Palo Alto Research Center, Palo Alto,  
[Broadbent 77] Broadbent, Geoffrey. 
"Design in Architecture.

[Brown 89] Brown, David C. 
"Making Design Routine". 

[Brownston 85] Brownston, L. Farrell, R., Kant, E., and Martin N. 
"Programming Expert Systems in OPS5". 

"Constructing an Expert System". 

Rule-Based Expert Systems. 

[Buchanan 89] Buchanan, Bruce G. 
"Can Machine Learning Offer Anything to Expert Systems?". 

[Canter 74] Canter, David. 
Psychology for Architects. 

[Carbonell 83] Carbonell, Jaime G. 
"Learning by Analogy: Formulating and Generalizing Plans from Past Experience". 

[Carbonell 85] Carbonell, Jaime G. 

[Chan 90] Chan, Chiu-Shui. 
"Psychology of Style in Design. 

[Clancey 86] Clancey, William J. 
"Heuristic Classification". 

[Chen 90] Chen, Chen-Cheng and Schmitt, Gerhard. 
"Design as Learning". 

"Learning by Inductive Inference". 
[Coyne 87] Coyne, R.D., Roseman, M.A., Radford, A.D. and Gero, J.S.
"Innovation and Creativity in Knowledge-based CAD".
In Proceedings of IFIP WG 5.2 Working Conference on Expert
Systems in Computer-Aided Design, IFIP, Sydney, Australia,

[Cross 84] Cross, Nigel (ed).
Developments in Design Methodology.

[de Kleer 86] de Kleer, Johan.
"An Assumption-Based TMS".
Artificial Intelligence, 28, 1986.

[Dietterich 83] Dietterich, Thomas G. and Michalski, Ryszard S.
"A Comparative Review of Selected Methods for Learning from
Examples".
In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (eds),
Machine Learning - An Artificial Intelligence Approach. Tioga

Orlup, M. F.
"A Proposed Taxonomy of Mechanical Design Problems".
Proceeding of the 1988 International Computers in Engineering

[Dixon 89] Dixon, John R.
"On Research Methodology Toward A Scientific Theory of
Engineering Design".
In Design Theory 88', Newsome S., Spillers, W. R., and Finger S.
(eds), Springer-Verlag, New York, NY, 1989.

[Doyle 79] Doyle, Jon.
"A Truth Maintenance System".
Artificial Intelligence, 12, 1979.

[Eastman 72] Eastman, Charles M.
Logical Methods of Building Design: A Synthesis and Review.
Tech. Rept. CMU-IPS-72-28, Institute of Physical Planning,

[Eastman 75] Eastman, Charles M.
Spatial Synthesis in Computer-Aided Building Design.

[Eastman 80] Eastman, Charles M.
The Design of Assemblies.
Technical Report IBS-11, Department of Architecture, Carnegie-
Mellon University, Pittsburgh, PA, 1980.

[Eastman 85] Eastman, Charles M.
"Abstractions: A Conceptual Approach for Structuring Interaction
with Integrated CAD Systems".

[Ernst 86] Ernst, M. L. and Ojha, H. E.
"Business Applications of Artificial Intelligence KBS’s".
FGCS, 2, 1986.

[Evans 63] Evans, T. G.
A Heuristic Program to Solve Geometric-Analogy Problems.
Ph.D. Thesis, Department of Mathematics, Massachusetts Institute

[Falgings 88] Faltings, Bepi.
"Qualitative Kinematics and Computer-Aided Design".
In Proceedings of the Second IFIP WG 5.2 Workshop on Intelligent


140
Gips, James.  
*Shape Grammars and Their Uses.*  

Gips, James and Stiny, George.  
"Production Systems and Grammars: A Uniform Characterization".  

Gordon, William J.  
*Syntactics.*  

Gosling, James.  
*Algebraic Constraints.*  

Gross, Mark Donald.  
*Design as Exploring Constraints.*  

Groover, Mikell P.  
*Automation, Production Systems, and Computer-Aided Manufacturing.*  

Groover, Mikell P. and Zimmers, Emory W.  
CAD/CAM: Computer-Aided Design and Manufacturing.  

Gruber, Thomas R.  
*The Acquisition of Strategic Knowledge.*  

*Building Expert Systems.*  

Hilbert, James and Lefevre, Patricia.  
"Conceptual and Procedural Knowledge in Mathematics: An Introductory Analysis".  

Hirsch, M.B.  
"CIM for one of a kind production".  
*Keynote Speech, the Second European Conference on Application of Artificial Intelligence and Robotics to Building Architecture and Civil Engineering,* University of Liège, Belgium, Mar., 1996.

Holland, J. H., Holyoak, K. H., Nisbett, R. E., and Thagard, P. R.  
*Induction.*  

Hopfield, John J. and Tank, David W.  
"Computing with Neural Circuits: A Model".  

Indurkhya, Bipin.  
"Modes of Analogy".  

*Design : Science : Method.*  
Westbury House, Guildford, 1981.

Jantke Klaus P. (ed).  
*Analogical and Inductive Inference.*  
[Kalay 85] Kalay, Yuhada E.
"Redefining the Role of Computer in Architecture: from Drafting/Modeling Tools to Knowledge-Based Design Assistants".
CAD, Volume 17, No. 7, September, 1989.

[Keane 88] Keane, Mark T.
Analogical Problem Solving.

[Kedar-Cabelli 85] Smadar, Kedar-Cabelli.
"Purpose-Directed Analogy".

[Keller 88] Keller, R.M.
"Defining Operationality for Explanation-Based Learning".
Artificial Intelligence, 35 (2), 1988.

[Kittay 84] Kittay, E. F.
"The Identification of the Metaphor".

[Laird 86] Laird, J. E., Rosenbloom, P. S., and Newell A.
"Chunking in Soar: the Anatomy of a General Learning Mechanism".

[Langley 87] Langley, Pat.
"Machine Learning and Grammar Induction".

[Laseau 80] Laseau, Paul.
Graphic Thinking for Architects and Designers.

[Lawson 80] Lawson, Bryan.
How Designers Think.


[Marcus 86] Marcus, S., Stout, J. and McDermott J.
VT: An Expert Elevator Designer that Uses Knowledge-Based Backtracking.

[McDermott 80] McDermott, J.
RT: A Rule-Based Configurer of Computer Systems.

[Michalski 80A] Michalski, R. S.
"Pattern Recognition as Rule-Guided Inductive Inference".

[Michalski 80B] Michalski, R. S. and Chilausky, R. L.
"Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System Soybean Disease Diagnosis".
[Michalski 83] Carbonell, J. G., Michalski, R. S., and Mitchell, T. M.
"An Overview of Machine Learning".
In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (eds),
Machine Learning - An Artificial Intelligence Approach. Tioga

[Miclalski 86] Michalski, Ryszard S.
"Understanding the Nature of Learning: Issues and Research
Directions".
In Michalski, R, S., Carbonell, J. G., and Mitchell, T. M. (eds),
Machine Learning - An Artificial Intelligence Approach, Volume II.

Computer-Aided Architectural Design.

[Mitchell 78] Mitchell, Tom M.
Ph.D. Thesis, Computer Science Department, Stanford University,
1978.

"The Uses of Inconsistency in Design".
In Kalay, Y. E. (ed) Evaluating and Predicting Design
Performance, John Wiley & Sons, New York, NY, 1990
(Forthcoming).

[Mittal 86] Mittal S.
"Pride: An Expert System for the Design of Paper Handling

Native Genius in Anonymous Architecture.

"Toward Better Models of the Design Process".

[Mostow 87] Mostow, J. and Barley M.
Automated Reuse of Design Plans.
In Proceedings of the International Conference on Engineering

"Information System for Component Building". In Eastman,
Charles M. (ed), Spatial Synthesis in Computer-Aided Building

"On the Role of Analogy in Engineering Design: An Overview".
In Sriram, D. and Adley B. (eds), AI in Engineering. Proceedings
of the 2nd International Conference, Boston. Computational

[Newell 72] Newell Alan and Simon Herbert A.
Human Problem Solving.

[Nilsson 80] Nilsson Nils J.
Principles of Artificial Intelligence.

[Piaget 78] Piaget, J.
Success and Understanding.

143
[Poetschke 89] Poetschke, Dieter.

[Popper 72] Popper, Karl R.
Objective Knowledge.

[Quinlan 82] Quinlan, J.R.

[Radford 88] Radford, Antony D., and Gero, John S.
Design by Optimization in Architecture, Building, and Construction.

[Riesbeck 89] Riesbeck, Christopher K. and Schank, Roger C.
Inside Case-Based Reasoning.

[Rogers 78] Rogers, Gary.
BAU.

[Rosenman 91] Rosenman, M. A., Gero, J. S. and Ozman, R. E.

[Rudofsky 88] Rudofsky, Bernard.
Architecture without Architects: A Short Introduction to Non-Pedigreed Architecture.
University of New Mexico Press, NM, 1988.

[Russell 89] Russell, Stuart J.
The Use of Knowledge in Analogy and Induction.

[Rychener 88] Rychener, Michael (ed).
Expert Systems for Engineering Design.

[Schank 82] Schank, Roger C.
Dynamic Memory - A Theory of Reminding and Learning in Computers and People.

[Schank 89] Schank, Roger C. and Leake, David B.
"Creativity and Learning in a Case-Based Explainer".

[Scheffler 65] Scheffler, I.
Conditions of Knowledge: An Introduction to Epistemology and Education.

[Schmitt 88] Schmitt, Gerhard.
Microcomputer Aided Design.

[Schmitt 88a] Schmitt, G., Chen, C. C., and Isabelle P.
Schmitt, G., Chen, C. C. and Shih, S. G.
"Structuring Graphical Knowledge and Intelligent Design Environments".

Searle, John R.
"Minds, Brains, and Programs".

Shapiro, Alan D.
Structured Induction in Expert Systems.

Simon, Herbert A.
"The Structure of Ill-Defined Problems".

Simon, Herbert A.
"Style in Design".

Simon, Herbert A.
"The Architecture of Complexity".
The Sciences of the Artificial.

Smithers, T.
"AI-Based Design versus Geometry-Based Design or Why Design cannot be Supported by Geometry Alone".
CAD, Volume 21, No. 3, April, 1989.

Solomonoff, R.
"A New Method for Discovering the Grammars of Phrase Structure Languages".

Sowa, John F.
Conceptual Structures: Information Processing in Mind and Machine.

Springer, Sally P. and Deutsch, Georg.
Left Brain, Right Brain.

"The Nature Constraints and Their Effect on Quality and Satisfying".

Steadman, J. P.

Steinberg, L. I.
"Design as Refinement Plus Constraint Propagation: The VEXED Experience".

Stiny, George.
"Introduction to Shape and Shape Grammars".

Stiny, George.
"A New Line on Drafting Systems".
Sussmann, G.J. and Steele, G.L.
"Constraints - A Language for Expressing Almost Hierarchical Descriptions".

Sutherland, Ivan Edward.

Takala, Tapio.
"Theoretical Framework for Computer Aided Innovative Design".

Tulving, E.
Elements of Episodic Memory.

Turing A. A.
"Computing Machinery and Intelligence".
Mind 59, 1950.

Wade, John W.
Architectural, Problems, and Purposes.

Winograd, Terry.
"Frame Representations and the Declarative/Procedural Controversy".

Winston, P. H.
"Learning Structural Descriptions from Examples".

Winston, Patrick H.
"Learning by Augmenting Rules and Accumulating Censors".

Wolstencroft, John.
"Restructuring, Reminding and Repair: What's Missing from Models of Analogy."

Wulf, W. A., Shaw, M., Hilfinger, P. N. and Flon, L.
Fundamental Structures of Computer Science.

Yoshikawa, Hiroyuki.
"General Design Theory and Artificial Intelligence".