Towards Meaningful Computational Descriptions of Architectural Form

by

Manoel Rodrigues Alves
Bacharel em Arquitetura
Universidade Mackenzie
Sao Paulo Brazil
1980

Miguel Ruano
Título de Arquitecto Superior
Escola Tècnica Superior d'Arquitectura
Barcelona Spain
1985

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS OF THE DEGREE
MASTER OF SCIENCE IN ARCHITECTURE STUDIES
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 1987

Copyright (c) Manoel Rodrigues Alves and Miguel Ruano 1987
The authors hereby grant to M.I.T. permission to reproduce and to distribute publicly copies of the thesis document in whole or in part

Signature of the Authors

Manoel Rodrigues Alves
Miguel Ruano

Certified by

Aaron Fleisher
Professor of Urban Studies and Planning
Thesis Supervisor

Accepted by

JUL 08 1987
LIBRARIES

JUN 08 1987

Departmental Committee for Graduate Students
Towards Meaningful Computational Descriptions of Architectural Form

by

Manoel R. Alves and Miguel Ruano

Submitted to the Department of Architecture on May 8, 1987
in partial fulfillment of the requirements for the
Degree of Master of Science in Architecture Studies

ABSTRACT

Computers have irrupted in the domain of architectural design, still with uncertain results. The impetus are driven by two major forces: First, the momentum originated by extensive application of computational tools to any imaginable human activity, not necessarily related to design. Second, the experience derived mainly from engineering fields, which has by-produced systems for computer-aided architectural design. Yet, architects complain. The magic is missing. The goals, mistyfied. Computerized design tools are product-oriented, not process-oriented. In Architecture, the how's are as significant as the what's.

This thesis presents a computational environment for architectural design. The environment aims to overcome some critical limitations of current CAD systems. Architectural design is neither drafting, nor simulation, nor modelling. Architectural design is a process of progressive consciousness acquisition. The proposed environment helps architects communicate better with their design objects. It is an environment where design worlds can be expressed and explored, where architects can manifest their preferences to approach each design problem, what instruments are to be used and how they are to be manipulated. It is specifically intended to be a tool to design better, not directly a tool to produce better designs.

Thesis Supervisor: Aaron Fleisher
Title: Professor of Urban Design and Planning
Acknowledgements

We are sincerely grateful to Prof. Aaron Fleisher, our thesis advisor, for his faith, friendship, open mindedness and for being always there when needed. Without his encouragement and constructive arguments, this work could not have been produced. Aaron, we are very much indebted with you.

We would also like to thank:

Mark Gross, for taking us seriously, for patient reading through boring drafts, precise comments and sharp criticism.

Stephen Ervin, for helping us through the delights of debugging, posing difficult questions, and for giving us moral and physical support when it was most needed.

William Mitchell, for intelligent feedback and willingness to help, despite adverse circumstances.

Nabeel Hamdi, whose clarity and structure helped to put scattered pieces of work together at early stages.

Patrick Purcell, for kind unconditional support.
towards meaningful computational descriptions of architectural form
"I have yet to see a problem, however complicated, which, when you looked at it in the right way, did not become still more complicated."

- Paul Anderson
Contents

Abstract 2
Acknowledgements 3
Contents 6

1. Introduction
  1.1 Preface 8
  1.2 Computers and Architecture 9
  1.3 Computing Architectural Worlds 14

2. Design as an Understanding Process
  2.1 Understanding Space 16
  2.2 The Power of the Representation 19
  2.3 Design as a Least-Commitment Search 23

3 Computational Representations of Architectural Form
  3.1 Intelligent Representations 25
  3.2 Architectonic Elements, Architectural Primitives 26
  3.3 Architectural Knowledge and Default Values 31
  3.4 Representations of Relationships between Architectonic Elements 35
  3.5 Designing in a Programming Environment 39
  3.6 Perspective. 41

towards meaningful computational descriptions of architectural form
## 4 A Computational Environment for Architectural Design

- 4.1 Architectural Representations ................................................. 43
- 4.2 Definition of Architectural Primitives ............................... 49
- 4.3 Organization of Architectural Primitives ......................... 54
- 4.4 Generation of Architectural Representations ................... 63
- 4.5 Manipulation of Architectural Representations ................. 69
- 4.6 Inheritance of Default Settings ........................................ 75

## 5 Discussions

- 5.1 Considerations ..................................................................... 90
- 5.2 A Tool for Managing Architectural Knowledge ............. 91
- 5.3 Association of Graphic Description with Architectural Primitives 93
- 5.4 Multiple Descriptions ..................................................... 114
- 5.5 Summary .......................................................................... 119

### Appendix 1

- Scheme Code ................................................................. 122

### Appendix 2

- The Graphic System ....................................................... 135

### Bibliography

- 146

### List of illustrations

- 155

*towards meaningful computational descriptions of architectural form*
Chapter 1
Introduction

"An expert is one who does not have to think. He knows."
- Frank Lloyd Wright

1.1 Preface

Nowadays, a major concern within western societies is to find ways to formulate, systematize, manipulate, preserve, reproduce, and transmit knowledge. Currently, computers are one of the mainstreams providing basis and support for activities related to knowledge acquisition. If knowledge can be basically defined as an acquired familiarity with facts and/or relationships, we are concerned with the contribution computers can bring to Architecture.

This work demonstrates how computers can provide assistance in architectural design decision-making. The ideas presented here do not try to offer a finished product, nor to be a universal solution. Rather we propose ways to overcome limitations of present so-called 'computer-aided architectural design' systems. The basic idea sustained in this thesis to overcome some of these limitations is to present a computer environment within which architects can create their own architectural worlds, an environment for conception, representation, refinement and evaluation of design ideas.
1.2 Computers and Architecture

We have decided to assume the point of view of an architect in order to define the role of computers within the architectural design process. Our approach represents a fundamental shift in the way that has traditionally been chosen to tackle this issue. Computer programmers and system analysts have been trying to explore the needs of design professions in order to build systems that could be commercialized as computer-aided design tools. While this strategy has been successful in engineering design and other related fields, it still faces much criticism from many architects, mostly because such systems would be better defined as 'computer-aided drafting'\(^1\). Software developers seem to understand that architects are 'blueprints producers', so they have provided us with powerful rendering tools\(^2\). Their usefulness as authentic design tools is very limited, since most, if not all, of design decisions have to be made before such systems can be used. Clearly the architectural design

---

\(^1\) Currently, much effort is devoted to the development of Expert Systems as well as CAD Systems. "The conceptual worlds that are modelled and manipulated are different, but they share the need to represent information about both the architect's ideas and also the domain in which the ideas exist. In CAD, the architect's ideas are expressed as lines, curves, polygons, text, and symbols. The domain is the world of drafting, and the information here is about how drawings are made and modified. Meanwhile, in expert systems, the architect's ideas are expressed as a description of a problem situation. The domain is the field that deals with the problem, and the information here is what is needed to solve the problem in terms of expert knowledge." [Jurgensen 1987, pg 80]. Expert Systems are able to evaluate situations within a narrow knowledge area with the accuracy of a human expert and have been characterized to include knowledge acquisition capabilities, knowledge representation schemes, inference strategies and explanation features.

\(^2\) An extensive overview and excellent bibliographical references of the work done so far in CAD systems may be found in [Vanier and Grabinsky 1987].
process has been repeatedly misunderstood. All the crucial design decisions take place specifically between the first sketch of the site and the final draft before the production of the construction drawings.
The fundamental characteristic of architectural designs is the simultaneity and relevance of both form and function and their corresponding descriptions. In spite of functional axioms of 'rational architecture' limiting the symbolic value of the architectural form, duality and multiplicity are characteristics deeply embedded in architecture. The architect shifts back and forth from form to function and from function to form aiming at the production of a functional, symbolic and expressionist object. So, "architectural design although exhibiting some of the features of engineering design possesses a different character. The emphasis on the specification of function is lacking and we cannot say that architectural design has a clear input and output structure. Many parts of architectural composition are not bound by well-defined sets of components and component properties arising from a particular functional specification...Approaches to CAD...are predominantly function-led design worlds, which enable the generation and evaluation within a knowledge base which is largely functional." [Earl 1986, pg 178].

Standard graphic editors in CAD systems are 'blockheaded' precisely because they will do whatever they are told to - since they do not 'understand' differences between things. Flexibility and intelligence are not synonym. CAD systems present a world of spatial consistency, due to the fact that most of them where originally engineering design tools. Engineers prefer consistent design worlds, whereas architects frequently work with non-consistent space to stimulate the generation of design ideas to move, later on, to a consistent ('buildable') architectural space. Under such conditions, many times the effort made to generate a computer model of a design object is not worth its
usefulness or the information it conveys. Attempts to provide more 'realistic' representations fail to understand that architects need to represent what is relevant, and not necessarily in a realistic fashion.

The complexity of architectural design has not been fully taken into account in previous attempts to apply computers. Many variables are involved in any architectural design problem, and the certainty of their values changes constantly during the design process. In addition, architectural design is based, to a large extent, on subjective judgements. "Since architectural design has been shown to proceed from first approximation solutions (schematic design) to higher order approximations (preliminary and final design), mathematically one may think of using methods of successive approximation....but these techniques may be at times not even theoretically applicable....It would be conceptually possible to consider design as an iterative process with feedback, for which there are analytical and numerical techniques, were it not for the type of feedback to be expected, which depends on non-probabilistic responses due to value judgements." [Salvadori 1974, pg 98]. Although the design process is not solely an intuitive one, intuition is a major component.

Computers are already contributing to increase the efficiency of design processes by providing the means to produce better models of design objects than the ones achieved through traditional means. "Static standard procedures, such as plans, elevations, and sections may fail to convey certain spatial qualities....whereas other media - computer graphic displays, for instance, may not be so limited." [Rowe 1987, pg 98]. More and more frequently we find...
CAD systems as part of the working environment in architectural offices. However, most of these systems are appropriate to carry out solutions only within the domain of well-defined and well-structured problems. One more severe limitation is that CAD systems present a predefined environment where many critical decisions that architects need to control have already been made by somebody else. Even if some such systems offer some room for modification by means of limited programming capabilities, they are far from the necessary stage where an architect can tailor the system to suit his needs. Tailoring the system to individual needs means, to us, something else than alternative graphic representations, manipulable geometric transformations or linkage between geometric and symbolic data.

On the other hand, it can be argued that programming languages offer to anybody the possibility of writing programs to suit his particular needs. However, there are two major hindrances to this approach. The first one is the fact that not everybody can be expected to have the necessary knowledge to program a computer or the resources to support somebody else to do it. The second obstacle, but most important, is the fact that programming languages are oriented towards problems that can be precisely described in algorithmic terms, which is not always the case in architectural design. Design reasoning, in general, is not algorithmically organized as a whole process, even if architects often use structured thinking to deal with many aspects of design problems.
Still, computers are here, becoming progressively more accessible to anybody who wants to use them. An increasingly cheaper technology is nowadays available, as well software of any imaginable kind. We believe computers still have a lot more to offer beyond their current applications, and that they will determine decisive changes in the practice of architectural design.

1.3 Computing Architectural Worlds

To define personal architectural worlds computationally, architects must be able to express the way a particular design problem is to be approached, what means are to be used, and how they will be manipulated. Since architectural worlds change not only from architect to architect, but also from project to project, we are looking for is a computational environment that is adaptable to continuously changing needs. To specify such an environment, we have looked into some of the crucial features of designing. We have incorporated such features in an embryonic version of a computational environment that will be introduced in Chapter 4.

Such an environment provides architects with the capability to store knowledge about architectural design, so this knowledge can be explicitly manipulated, increased, modified and organized by the architect with the help of the computer. The proposed environment deals with 'intelligent' representations of architectural form that allow for operations not limited to predefined schemes. It offers structures to store architectural information in terms of towards meaningful computational descriptions of architectural form.
architectonic elements, their properties and their relationships. We have also provided ways to transfer information from element to element, and systems to organize hierarchies of elements. We propose ways to deal with ambiguity, deferred decision-making and unclearly defined situations. Our environment supports and manages architectural knowledge other than purely graphic descriptions.

This thesis focuses on computational representations of architectural information, knowledge and ideas. It is not anymore a matter of what computers can do, but rather of defining what they should do and how they should do it to decisively contribute to the production of architectural design objects.

This thesis presents the following organization: Chapter 2 presents an interpretation of architectural design understood as a process of acquiring a progressive consciousness of the design object. Chapter 3 analyzes the main issues involved in a computational representation of architectural knowledge. Chapter 4 introduces an embryonic computational environment intended to help architects design. Finally, chapter 5 proposes a general outline for further work in order to verify the validity and potentiality of the ideas presented in this thesis.
Chapter 2  
Design as an Understanding Process

"For me it is important to learn that there are different ways of design process."
- Lucien Kroll

2.1 Understanding Space

The premise of this work is that architectural designing is a continuous process of interaction between the architect and the design object - process, in this context, means "a series of actions or operations conducing to an end".

Traditionally, such interaction is conveyed through different types of representations: sketches, plans, models, lists of alphanumerical data, various kinds of calculations, and so on. These different representations serve diverse purposes (economic, volumetric, structural, thermal, lighting, or spatial analyses and syntheses), but they are all intended to help the architect understand the consequences of his actions. Things can be represented in many different ways. Every representational system does not want to explain the whole world, but just a specific aspect. By articulating design processes through different representations, architects explore and understand attributes of the design object that could not be easily anticipated, otherwise. "Design

---

1 "A symbol is something which stands for something else. A representation is a particular kind of a symbol, in which the structure of the symbol itself is perceived to correspond in some way with the structure of the thing that the symbol stands for." [Smith 1978, pg 8].
problems are immense in terms of the different variables that must be determined. The functional, logical and aesthetic relations that must be satisfied between the variables are dense. There is no one representation that allows detailed consideration of such diverse concerns as spatial composition, structural performance, material selection and construction scheduling.” [Eastman 1982, pg 127]. The ultimate goal of an architectural representation is to objectify space by representing it. Current CAD systems offer different views but not different representations of the same design object.

Through this representational process, the architect develops a formal understanding of how the projected space is going to be perceived. The architectural design process presents, at least, three different domains of work, the first of which we will call the ‘project’. It encompasses the actions necessary to translate the architect’s thinking process into a physical entity, the design object. The second one, the ‘product’, is the outcome of the ‘project’, the architectural space. The third one is the ‘use’ of the product; it is within this

---

2 ‘Objectify’: "to make objective".

3 "Although the power of representation in shaping perceptions has only been recently broached in architectural thought its role in architectural history is obvious. Changes in conceptions of space brought about by the invention of projective and analytic geometry, perspective or axonometric drawings certainly affected the practice of building design. Likewise, representations can be seen as a tool for conceptualization ....The overlaying of representations lends richness and understanding to our conceptions. Major alternatives will generally be modelled several ways before a decision is made since the more comprehensive description depends not only on each partial representation but also on interaction among representations." [Isenstadt 1985, pg 28].

4 Formal here is understood as "clear and definite" as opposed to 'intuitive': "understanding without reasoning".

towards meaningful computational descriptions of architectural form
domain that spatial experiences are manifested. Within these domains, we will identify four different kinds of space: 'actual', 'perceived', 'mental' and 'represented'. 'Actual space' is the three-dimensional environment that human beings inhabit, experience and apprehend. 'Perceived space' is the unstructured heap of perceptions and sensations an observer experiences when affected by the 'actual space', and the simultaneous phenomenon by which the observer affects the 'actual space' by creating his own 'mental space'. 'Mental space' is an abstract conception that human beings develop to apprehend three-dimensional environments by structuring perceptions and sensations. When such conceptualizations are described, we have 'represented space'. Descriptions can take different media: spoken or written word, two dimensional graphics, three-dimensional models....

'Understanding space' is understanding the relations between humans and their space through the four mentioned facets and how links between the facets are established. Architects act directly upon the 'actual space', however intending to understand and control how humans are going to experience it ('perceived space'), what meanings they are going to associate to it, and what sort of 'mental space' they are going to develop. Understanding space becomes, thus, the key question in architectural designing process(es).” ....Many studies on the concept of space in relation to architecture have either tended to leave man out by discussing abstract geometry, or have made man enter by reducing space to impressions, sensations and studies of effects. Space has to be

5 Abstract is here understood as “thought of apart from any particular instances or material objects; not concrete.
understood as an existential dimension, as a relation between man and his environment." [Norberg-Schulz 1971, pg 14].

During the design process, architects work on the four kinds of space. In 'mental space' architects develop, understand and evaluate design objects. Architects materialize their 'mental space' in the 'represented space', which acts as the link between the 'mental space' and the 'actual space'. Thus, through this represented space the architect accomplishes a formal understanding of how the 'actual space' will be perceived. Such understanding acts in the realm of his 'mental space' to induce new decisions. These new decisions affect the design object and this cyclic mechanism takes place again.

2.2 The Power of the Representation

Architects get feedback from their own ideas when these ideas are represented - 're-present': "to present again". New decisions are produced as reactions to what the architect 'sees' in the representations. Applying knowledge and experience, the architect introduces tentative decisions and explores their hypothetical consequences. "Design expertise is not in solving problems, but in exploring for solutions" [Gross 1985, pg 136]. New decisions are based on their plausibility and reasonability, but they need not to be correct, and can be discarded later on. Designing is based on a generation-test-evaluation cycle that can be expressed as:
1) proposal of a design hypothesis;
2) experiment to understand the consequences of its application to the design;
3) evaluation and conclusion leading to a new hypothesis that can be a refined version of the previous one, or a completely different one.

Many times architects just experiment with ideas: they may cause alterations or make changes to the design object to see how it ‘behaves’ under ‘what-if?’ conditions. Understanding this ‘behavior’ helps to understand the idiosyncrasies and features of the object being designed. Design objects have their own internal logic structure, and, sometimes, the architect is just trying to understand what it is that he is creating, to get clues in order to progress consistently. Some decisions incorporate abstractions intended only to be used during the design process. Abstractions may eventually disappear and not be present in the final design, either because they are just ‘tools’ (like grids, construction lines, etc.) or because they are temporary decisions that will be replaced by better ones. When designing, architects have to abandon many ideas, many possibilities; they make choices.

---

6 “Design can be understood as a process of successive refinement. Refinement proceeds in two alternative steps; describing constraints and exploring alternatives, or variants. The describing-step adds new constraints to the design; the exploring-step examines variants in the constraint region. These variants suggest changes and additions to the constraints. The cycle then repeats; the new context of constraints is explored, generating a new set of variants. This process of refining constraints and exploring alternatives is repeated until it converges on a small region of acceptable variants, or alternatives.” [Gross 1987, pg 14]

7 Architects also deal with ‘abstractions’ as visionary ideas of design objects, an idea of quality apart from its material accompaniments.
Every move in the design process is the product of a deliberate action whose ultimate purpose is to find a solution to a specific problem. However, there is not a sole solution for an architectural design problem, and it is difficult to agree about what the best possible solution between all the possibles alternatives would be. There is not agreement between architects even about the methods to find the solutions. For these reasons, design decisions are neither deterministic nor predetermined. Despite their deliberateness, the results of such actions may not be exactly what the architect had in mind. An architect may be surprised by an 'accident', an unexpected consequence of a decision. Nevertheless, even if a design exploration follows a random pattern of generation of ideas, it will still be a process oriented towards the accomplishment of a desired aims. These aims materialize in the built form itself, which could be seen as the final representation of a design solution for a particular set of requirements. The goal of the design process, from this viewpoint, would not be the built form, but a design solution.

8 "All early models of the design process have one characteristic in common; they all view the design process as a sequence of well defined activities and are based on the assumption that the ideas and principles of the scientific method can be applied to it...It is interesting that all early models of the design process advocate the definition and use of formal techniques, frequently borrowed from operations research. They all imply that, by following systematic procedures and by using prescribed techniques in design, the design itself can be improved: optimum solutions to design problems can be more readily determined...They all view the design process as a step-by-step procedure in which analysis can be separated from synthesis, and procedures can be prescribed which will improve the product: the design solution." [Bazjanac 1974, pg 5-6]. An introduction to the early models of design may be found in [Bazjanac 1974]. A more detailed analysis may be found in [Broadbent 1973] and in [Rowe 1987].

9 "The design solution is a plan for accomplishment of the solution, not the solution itself." [Bazjanac 1984, pg 11].
Design can be understood as the process of going from representation to representation by reacting to the emerging ideas, especially forms, that each representation makes visible [Mitchell 1986a]. A different representation may contribute uniquely to the architect's understanding, and suggest new ways for design explorations. Each representation is the result of a new experiment, the consequence of the inclusion of a new idea, a change, an increase of information about the design object.

The design process is essentially based on a progressive accumulation of information about the characteristics of the design object. This accumulative process does not usually follow a linear structure, but it is rather characterized by forward and backward moves, reflecting the results of successive manipulations and experiments with architectural ideas. "The architect continuously goes through alternating sequences of generation of variety and reduction of variety. During the generation of variety he searches for relevant possibilities; during the reduction he evaluates and selects the most desirable or the most feasible." [Bazjanac 1974, pg 14]. During this process, proposals become more and more precise - and designing becomes a search to reduce ambiguity in the representations of the objects being designed. Inconsistencies are accepted - they are, indeed, a characteristic component of the architectural design process -, but they are to be progressively reduced. Reduction of ambiguity and inconsistencies in the representations of the design object does not preclude the significant role that ambiguity plays in architecture. Control of ambiguous or contradictory forms or meanings in the design object requires, precisely, a deep formal understanding and the appropriate communication towards meaningful computational descriptions of architectural form.
tools. Ambiguous representations can transmit confusing messages, difficulting the development of formal understandings and the communication between the different parties interested in the design.

2.3 Design as a Least-Commitment Search

When designing, the architect usually wants to remain as uncommitted as possible; that is, he tries to concentrate on what at each phase are the crucial aspects of the design, leaving many variables undefined. Hence, design can also be understood as the process of deferring decisions, until they clearly have to be made. Using his expertise\textsuperscript{10}, the architect can analyze the design object at any stage of development and draw conclusions, without needing to add information other than that already provided. Sometimes he will make assumptions about undefined variables (dimensions, positions, materials, etc) in order to analyze the product from different points of view (thermal, structural, volumetric, etc.). In other words, architects deal frequently with conflict and complexity by making assumptions\textsuperscript{11}. This method, however, works adequately only under conditions of limited complexity. Most frequently, architects must deal with far too large a number of variables to control them all. Computers can help architects deal with the increasing complexity of decision-

\textsuperscript{10} Expertise is related to knowledge acquisition and is domain specific. It is made of knowledge about a particular domain, understanding most of the domain problems, and the ability to solve these problems.

\textsuperscript{11} An 'assumption' is "a first move that becomes a commitment only gradually".
making processes affecting the modern built environment by providing tools to manipulate assumptions and understand potential consequences.

The control of complexity leads to complex designs, while the lack of control leads to complicated designs. Any architectural design involves its particular set of variables that need to be considered; if they are not, the architect loses control over the design object, since all the variables are going to be involved, anyway. The higher the control, the better the product.
Chapter 3

Computational Representations of Architectural Form

"You make your tools; then your tools make you."

- William Mitchell

3.1 Intelligent Representations

In order to use computational tools to manage complexity, formal representations of complex structures are necessary. Computers do not do things the same way people do, even if the results look alike. So far, machines cannot understand space directly, but they can be instructed to simulate this understanding; this can be accomplished by programming them to understand formalized architectural representations.

Architects generally represent space using conventions (a kind of embedded 'intelligence'). Conventions help architects and other people sharing the same code understand the qualities of the projected built spaces. At the beginning of the design process, ideas are expressed vaguely, and architectonic elements materialized schematically. However, even at these early stages, such representations encode a relevant amount of 'intelligence'. Whereas a layman sees just lines and polygonal shapes in a sketch, a skilled reader understands 'walls', 'rooms', 'stairs'. Information about sizes, proportions, materials,
colors, shapes and volumes can be obtained from such representations. This implies not only an understanding about what the conventions to represent things are, but also about what the represented entities 'mean'. For instance, two parallel lines filled with a hatched pattern may represent a 'wall', but the concept 'wall' means specific materials, certain dimensions, probably opacity and possibly also load-bearing capabilities.

Computers can be instructed to understand these 'intelligent representations', so they can deal with the design object the same way as a skilled observer would. Therefore, the computer must be capable of producing and understanding different representations of a design object. Such representations relate to different stages of the design process, to different graphic scales, and to different design intentions.

3.2 Architectonic Elements, Architectural Primitives

Architects deal not only with primary graphic elements (lines, circles, squares, polygons, etc.), but mainly with architectonic elements (walls,
columns, windows, and also rooms, courtyards, kitchens, etc.). For this reason, the machine should have 'architectural primitives' that will enable the architect to generate representations of the corresponding architectonic elements\(^3\). The architect creates his architectural primitives according to his needs. He also defines variables that will represent the properties of new or existing objects, especially those that are relevant to particular design situations.

The ways architectonic elements are graphically represented can be defined to serve different purposes. For example, a wall may be represented as a single line, as two parallel lines, as a three dimensional prismatic body or as any other graphic entity that the architect feels comfortable with. This provides a flexible environment and allows for the capability of different approaches to different design problems. The architectural primitives will reside at a higher level than the already traditional 'graphic primitives' that will also be implemented and accessible to the architect.

Within the proposed environment, architects create their own architectural primitives as computational entities\(^4\) associated to specific properties. This 'create-your-own' capability is essential to our approach, since it provides the basic way for architects to computationally express and store architectural knowledge. The representation of knowledge is indispensable to

---

\(^3\) We use the concept 'primitive' as something to be used to generate a series of similar things using the primitive as a model, a template or a rubber stamp.

\(^4\) 'Entity': "something that has a real and separate existence either actually or in the mind".
make knowledge computationally manageable and manipulable. An architectural primitive is created, essentially, by associating a name (e.g., 'wall', 'window', 'column', etc.) to a list containing the variables that reflect the properties of each architectonic element that the architect thinks is relevant (e.g., 'position', 'height', 'shape', 'material', 'color', etc.). The list also names the components that will be specified for the elements. These components will be, as well, architectonic elements with properties and, perhaps, components of their own (e.g., in a 'column' the components could be 'capital', 'shaft' and 'base').

Every instance generated from a specific architectural primitive shares with other instances of its class the same set of properties and components. Additional properties or components can later be added to the architectural primitive, and all the instances of that class will gain the new property; properties or components can also be added to a particular instance of a class in order to store particular information about that instance - information which will not appear in other instances of its class. The different values assigned to the properties and the different specifications for the components will differentiate one instance from another, so 'column1' and 'column2' are both columns but not the same one. Although all instances generated from an architectural primitive share its very same essence, there are means to differentiate between instances of the same class. This distinction is similar to that proposed by Mitchell in terms of 'essential properties' (properties shared by all instances of an object of a certain class) and 'accidental properties' (properties that distinguish an instance of an object of a certain class from the other instances) [Mitchell 1986b].
By using these architectural primitives, the machine stores the information necessary to 'understand' what kinds of architectonic elements are represented; through understanding the qualities of each element, the machine can analyze the design object according to diverse criteria (volumetric, structural, thermal, lighting and so on). This way, the architect can translate his architectural 'expertise' and 'heuristics' into a logic understandable by the machine.

Within the architectural primitives, variables are used to represent the properties of architectonic elements. During the design process the values assigned to such variables - i.e., the specifications for the properties - may be subjected to many changes. One value that a variable can have, more likely at the beginning of the process, is that of 'unassigned', meaning the architect knows the variable has eventually to be fixed but he does not want to assign a value for it. As we have seen, variables can also be added during the design process, as descriptions of the design object become more precise. This way, descriptions of architectonic elements take different degrees of completeness.

"Partial specification of variables is a useful device for representing ambiguity"
When the architect draws a line that represents a wall, the computer must understand 'WALL' and not 'LINE', even though the wall is represented graphically as a line. Through the designing, the architect increases the information about that wall by adding new data about materials, dimensions, colors, structural role, etc. These are the 'variables' that describe a wall, the essential properties that two different walls A and B both have in common. The different values of these variables and accidental properties cause A and B to be different walls, but both are still WALLS and not, for example, WINDOWS (the computer 'understands' that a window has a different set of variables that reflects its different properties and behavior, even if both the wall and the window have the same kind of graphic representation - for instance, a rectangle). A wall is not anymore just a set of lines, but an entity with qualities. For instance, a wall is often opaque, load-bearing and made of, say, bricks, while a window is commonly transparent and has a different thermal conductivity.

The set of variables defined by the architect to represent, in an architectural primitive, the properties of a specific architectonic element are used by the computer to perform calculations and analyses. To do so, the machine checks tables where the information about the qualities of the materials are stored. When the computer is asked to analyze the thermal behavior of a wall, for instance, it goes to the appropriate table to get the insulation factor corresponding to each material constituting the wall and, combining these
values with the dimensions (thickness, in this case), it makes the calculation. In the same way the computer performs structural, natural lighting, or any other kind of analysis.

With such a system, it will be easy to elaborate alternative design solutions, and compare them in terms of cost, structural behavior, thermal efficiency or spatial qualities. If designing is making experiments, access to a freer manipulation of the design object opens new possibilities to compare more alternatives faster and, hence, less expensively. The proposed system is, in fact, an environment to realize design experiments.

3.3 Architectural Knowledge and Default Values

How does the computer know what variables are relevant in a particular analysis? If it must ask the architect for a long list of values before being able to make any calculation, the system will be so clumsy that will be useless. The computer must make assumptions about the values that the variables involved in the calculations must take if their status is 'unassigned'. 'Default values' are introduced in the system by the architect to represent information about plausible and probable specifications regarding materials, dimensions, environmental parameters, costs, structural behavior or relationships between architectural elements. Some examples would be: 2.5 feet as the width of a standard door, $200,00/square feet for double-decker single family houses, 68 degrees Fahrenheit for a comfortable temperature, etc. This is another way to
introduce 'expertise' as part of the 'intelligence' of the system. Default values relieve the architect of the tedious task of assigning parameters to all design variables. Such a feature reflects the trade-off between keeping design flexibility and the need to progressively specify the characteristics of design objects. Default values are used as temporary assumptions until all the specifications have been determined by the architect.

The introduction of default values enables the machine to perform analytic tasks at any stage of the design process without waiting for the architect's complete definition of the design object. Nevertheless, if the status of a variable is not 'unassigned', that is, if the architect has already assigned a value to that variable (e.g., 'wall made of brick'), this assignment must have priority over any default value for that variable. The machine can also be instructed to make some inferences on the basis of the information it already has, and may keep the architect informed of the inferences that have been made. For instance, if a window is drawn on a facade, the machine may be instructed to infer, according to a hierarchical structure of systems and subsystems, that the window will be a component of that facade, as long as the architect does not establish a different relationship. This way, the computer makes sure the window will not be left floating in air when the facade is moved. But, the computer can also be instructed to make more sophisticated inferences. For instance, a rectangular opening is likely more suitable than a hexagonal one for

---

6 "Upon gaining experience, the novice designer begins to build a set of preferences. In contrast, the expert designer has already built up a large set of default constraints and preferences." [Gross 1985, pg 20]
a wall made of bricks, and its dimensions will probably be subjected to some constraints; an I-shaped column will probably be made of steel, and so on. Default values provide the computer with the capability to make inferences about appropriate decisions to satisfy functional problems. For instance, the slope of a chair or the minimum area for a bathroom.

Final decisions are always the architect's responsibility; machine inferences are only temporary moves to provide nimbleness to the design process. This capability provides the proposed system with a qualitative advantage over others that can just 'check' dimensions and materials when the design is finished, but that are not really useful during the design process, since they can not help the architect make better decisions. At early stages of the design process, it is not necessary to perform extremely accurate analyses; they can be performed by means of rules of thumb and approximations. Although numbers have a role in the evaluation of functional performances, architects often rely on intuition and experience rather than on thorough analysis. However, if thermal, acoustic, structural, cost-benefit and other technical analyses are not more frequently performed at early stages of the design process, it is not because they are considered irrelevant, but because they are time-consuming and, hence, expensive. When these kinds of analyses can be performed almost instantaneously by a machine, the architect will get a better control over the design object.

The default values may be conveniently grouped as 'sets' within the architectural primitives, each of these sets probably related to a specific building
system, to previously experimented combination of values, or, maybe, to architectural languages previously defined by the architect. For instance, a set might be called 'skyscraper' and would contain values for steel structure, curtain wall and so on; another set could be 'row-house', with load-bearing walls and tile roofs. In fact, sets of default values can be used for any kind of architectural primitive (e.g., default values for an 'external-wall' could be material: architectural concrete; thickness: 8 inches; height: 10 feet). The architect can, of course, arrange new combinations or modify the existing ones, adding new sets of default values to suit his changing needs.

The capability to alter is crucial because any useful tool must leave the design process open. Every design process has its own structure that will be reflected in the sequence of decisions to be made. One possible way by which diverse architectural design process structures could be supported by the computer would be by instructing it to make plausible and probable assumptions through the use of inference rules. However, since such design structures are organized and modified by the architect according to his needs, any computer system that prejudices rigidly either the kind or the sequence of decisions should be regarded as a failure.

The default values are assigned to the variables associated to each architectural primitive. When the architect requests the machine to perform a calculation, the computer looks for the variables that are involved. If the variables have a value fixed by the architect, it will use such a value. Otherwise, it will use the default value that the architect may have stored in the primitive. If
it cannot find any value for an involved variable, the machine will turn to the architect asking for the information missing. Once all the variables needed for a particular analytic task have been assigned a value, the computer turns to the already mentioned tables and proceeds to give an answer to the architect's request. The machine can also store knowledge to perform 'active' calculations to generate solutions to specific problems, besides 'checking' existing conditions. For instance, the computer can infer the material of a wall for a given thickness, a defined height and a specific load; or it can assign dimensions to a reinforced concrete column under a certain stress, and so on.

3.4 Representations of Relationships between Architectonic Elements

We have seen how the computer can be instructed to store structured information about properties of architectonic elements including, optionally, default values for the properties. Properties may describe either intrinsic characteristics of each particular architectonic element, or of a general class of elements. Specifications about the relationships between architectonic elements can also be stored in the machine by means of the architectural primitives, as well as default settings for such specifications. Thus, the computer will understand not only about architectonic elements, but also about relationships between elements. Different kinds of relationships can be defined, modified or suppressed, providing the architect with a very effective tool to deal with complexity.
Whenever a hierarchical organization is needed, it has to be possible to express it within the system; for instance, 'windows are in walls and walls are made of bricks', so when the wall is moved, the windows and the bricks move accordingly. This hierarchical organization provides the system with the basis to understand dependencies in the built environment. In the previous example, we have elements that belong to different levels of the architectural form. "A level is a class of elements that have the same inherent dependency behavior....The dependence between elements is inherent, indicating their relative permanence or changeability" [Gross 1985, pg 76]. In order to control the propagation of changes through design, the system should understand dependency in its practical terms: "if element A is moved, what other elements should move accordingly?". Using this hierarchical structure it is possible to create higher levels of architectural primitives, as long as they can be formally described. This way, 'room' and 'house' can also become architectural primitives, as they are already architectonic elements. This arrangement proposes a hierarchy among architectural primitives, each level acting at a different scale, and fixing the relationships among architectonic elements: houses are composed of rooms which are shaped by walls that are made of bricks, and windows are on external walls of the rooms of a house that is built in a block that is part of a neighborhood that belongs to a city connected to a region.....

---

7 An introduction to the discussion about dependencies in the built environment can be found in [Gross 1985]. A more exhaustive discussion is found in [Habraken 1983a].
Hierarchical relationships are not the only ones that can be specified within the system. Spatial relationships can be defined both in terms of positions and dimensions, providing an environment for experimentation on spatial organizations. As long as an architectural form can be expressed in terms of elements and their relationships, the proposed system provides the architect with the environment to make both elements and relationships explicit and, hence, manageable and manipulable with the help of the computer. In order to materialize his design intentions, the architect can use different instruments within the system.

Geometry is such an instrument. Even if the proposed system's concerns go beyond the plain issue of graphic representations, it is important to clarify the role geometry will play within such system, as a tool to synthesize and analyze space. Geometry is not just an 'editing' tool to freely manipulate form and dimensions. Geometry is essential in the process of form definition and, hence, will determine some of the variables describing the architectonic elements. A 'wall' can be rectilinear or have a 'B-spline' shape, or be parallel or perpendicular to another 'wall', for instance. Proportions can also be helpful: "wall A" is twice as tall and half as long as "wall B". The architect does not need to deal with dimensions in numeric terms, but just in relation to other dimensions. Only one reference is necessary, either the scale of the drawing, the dimensions of the site, or the size of an architectonic element.

Parametric Variations are useful to manipulate dimensions and proportions, providing new ways for methodological exploration of form.
computational representations of architectural form

variations. Parametric Variations affect relative dimensions of elements and relative positions between elements. To do so, the dimensions and the positions need to be expressed as functions of one or more parameters (e.g., 'the height of column-15 is seven times its radius', 'the width of the window-11 is 1/3 of the length of the wall-4 and the height of its sill equals the height of table-3 plus two inches', etc.).

Shape grammars can be expressed as sets of rules affecting elements and their spatial relationships. Shape grammars are rule-based systems which, basically, consists of an initial state, a set of finite vocabulary elements, a set of production (IF-THEN) rules to be applied to the elements₈, and a terminal condition. The architect defines its own grammar to the machine, and then uses it to explore the generation and transformation of architectural forms according to his interests. Shape grammars "can be used to explain and describe a given corpora of designs with common features, or to develop new designs and to test the rules on which they are based....Shape Grammars lead to more comprehensive results, which are stated with precision and rigor, and to a deeper understanding of the issues involved." [Flemming 1986]. The architect can, at any time, modify the vocabulary elements or the production rules in order to conduct form explorations.

Notational systems, like 'Write Form' [Habraken 1983b], [Habraken and Gross 1984] describe textually the built environment, in terms of elements

₈ In the case of shape grammars, IF-THEN rules are composed of a left-hand side representing the condition and a right-hand side representing the action part.
and their spatial relationships. Textual and graphic descriptions map each other, with the advantage that pattern-matching is easier on text than on images. This capability can be useful to explore transformations of form by recognizing configurations (patterns), applying transformational rules and producing new configurations. The architect expresses the configurations to be recognized, the rules to be applied and the new configurations to be produced; changes can be introduced in any of them.

Constraints [Gross et al 1986], are used within the proposed system to manage geometrical, topological and dimensional relationships between architectonic elements, to verify the satisfaction of established specifications (materials, dimensions, building regulations, etc.), and to implement inferential procedures by means of relationships based on logical reasoning. Chapter 5 elaborates on the implementation and use of such instruments.

### 3.5 Designing in a Programming Environment

Organizing the represented space through elements and relationships in order to deal with complexity requires an accessible framework where every architect can develop his personal architectural languages making them appropriate for every specific design situation. Access to free manipulation of the computational environment conflicts with current commercial policies tending to offer finished, closed, non-accessible tools. Computers may be an architectural design tool only if architects can keep designing their architectural
worlds at the same time they are giving shape to design objects as with traditional design tools. Architects explore and redefine their commitments as long as they design. They usually do not establish aprioristic rules to proceed to design afterwards. More likely, they will break the rules, define new ones and break them again during the design process - that if they use any rules at all\(^9\). In any case they need a flexible environment where these activities can be supported. Clearly, this can only take place within the context of a programming environment, where the architect can explore and manipulate the design object in his own terms, and where free access to all data structures is available\(^10\).

The computer can offer a fertile framework to materialize ideas, so that they can be freely manipulated to open new ways for design exploration. As Abelson and Sussman have pointed out: ".. a (computer) language is a framework within which we organize our ideas about processes. Thus, when we describe a language we should pay particular attention to the means that the language provides for combining simple ideas to form more complex ideas. Every powerful language has three mechanisms for accomplishing this."

\(^9\) "Design is more than following rules; it is making rules as well. Design concerns inventing and adapting systems of form organization as well as generating specific forms within a given rule system. By making new rules and combining and modifying existing ones, designers invent new styles and occasionally new building types. Moreover, the rules are not all decided before the design begins; rather they are adopted and invented throughout the design process. Rule making may even continue into the process of building." [Gross 1985, pg 5].

\(^10\) "The intention (of a programming environment) is to create a flexible environment in which contextual information, knowledge, and system processes are visible and amenable to modification and development." [Gero and Coyne 1986, pg 88].
Primitive expressions, which represent the simplest entities with which the language is concerned. Means of combination, by which compound expressions are built from simpler ones. Means of abstraction, by which compound objects can be named and manipulated as units." [Abelson and Sussman 1985, pg 4]. The three mechanisms are already present in architectural design environments, in a more or less conscious way. Nevertheless, to take full advantage of such a computational framework, a structured understanding of the goals of the design process is required. Appropriate formulations of architectural design methodologies and problem-solving strategies must be developed, so architectural knowledge can be stored in the form of precise, explicit computer-processable structures.

3.6 Perspective

It is important to point out that there is a difference between the architectural knowledge we can computerize and the one we may want to computerize. The describability of a problem and the corresponding solution method(s) are a necessary and sufficient condition for its computability. However, for whatever reason, we may decide not to computerize a piece of architectural knowledge that could have been computerized. On the other hand, we may not be able to instruct the computer to do something just because we do not know how to do it. It will be difficult - and of questionable design usefulness - to instruct the computer, for instance, to understand space in terms of perception, using subjectivity and cognitive psychology, or to understand
aesthetics. To computerize a piece of knowledge may have its advantages, but definitively it has also its price; some people may not be willing to pay such price if they think the advantages are not worth it. Notwithstanding, we believe computers can represent space in ways that improve the present communication flow between the architect and the design object. Given the increasing complexity of decision-making processes affecting the modern built environment, an electronic memory can help to keep track of all the variables that have to be considered during the design process. The machine relieves the architect's mind from ordinary time-consuming tasks, enabling him to concentrate on the specific problem of controlling the qualities of the space being designed.

By providing architects with better feedback, with better explanations of the consequences of their decisions, computers will compete with conventional representation tools and give architects a degree of control over their products heretofore unknown.
Chapter 4
A Computational Environment for Architectural Design

"Never speak more clearly than you think."
- Jeremy Bernstein

4.1 Architectural Representations

How might the ideas presented so far be reflected in a computational environment? We believe that the architect must be able to organize his architectural knowledge around conceptual entities (architectonic elements) with associated representations. The description of an architectonic element must be able to represent information about an entity with different degrees of completeness, as well as accommodate multiple representations which can describe the associated entity from different points of view.

Properties of the representations can be added, altered or specified during the design process. The architect may introduce new elements, delete or redefine the existing ones. He may also work on the relationships between elements, which will evolve, and in general tend to become more precise. The issues involved here are related to flexibility and ambiguity, to the conflict between the capability to defer decisions and the need to increase the information required to materialize a design solution.
To design our architectural knowledge representation scheme we decided not to provide pre-defined structures as built-in primitives representing architectonic elements, but rather to work towards a representation scheme capable of handling ambiguity, variability and other characteristics intrinsic to architectural design. To do so, the structure of the representation scheme must correspond to the architect's personal understanding of architectonic elements and their relationships, and must provide the means for modification whenever the architect's understanding changes. To serve this purpose, we decided to create an object-oriented environment where all data are contained in record structures called 'objects'; the environment accounts for operations like individuation of description (architectonic elements of the same kind can be described differently), internal concept structure in terms of objects (each architectonic element is described and manipulated as a distinct object) and interrelations between objects, and structured inheritance (a specific feature of the organization to be proposed) [Brachman 1979]. The proposed system encompasses an environment that helps the architect to manipulate information regarding architectonic elements, by modelling each architectonic element into a corresponding computational object.

One of the goals of the proposed environment is to provide means for the architect to express architectural worlds, both in terms of formal and functional knowledge. As such, the system reflects the capability to develop and investigate theoretical issues. Formal knowledge describes what the properties of an object are, while functional knowledge describes what an object does, what the relationships between the components of the object are. Regarding
formal descriptions, a design object may be described in terms of geometrical
elements, and topological, dimensional or equivalence relations. On the other
hand, in terms of functional descriptions it is necessary to describe the action
elements, the useful actions, the functional connections and the hierarchy of
elements. According to Earl [Earl 1986], formal knowledge deals with the
shapes of components and their spatial relations in assemblies, and functional
knowledge is concerned about how individual components behave and about
their functional relations.

The architect is provided with the capability to create his own
architectural primitives\(^1\) associated to specific properties. Such capability
constitute the basic means to instruct the machine about how to convey
intentions regarding different design situations. The architectural primitives are
defined with the help of a flexible set of underlying tools, in such a way that the
architect is committed neither to processing strategies nor to preestablished
ways to represent his architectural knowledge. The tools used to define the
primitives are neutral, meaning the architect implements his own primitives
without having to use predefined models. Therefore, the architects builds the
data-base using his architectural knowledge. Through the exercise of such
capability, the architect can describe architectonic elements by defining their
properties or by combining other architectonic elements. Hence, he makes
explicit the kind of elements that will configure his design environment and,
perhaps more important, what are the properties associated to those elements

---

\(^1\) The term 'primitive' is here understood as a computational object that, when instantiated,
will generate representations of architectonic elements.
that are considered to be relevant to tackle a specific design problem. The proposed environment does not oblige the architect to structure his architectural knowledge in a pre-established way; it is the architect’s responsibility to structure the elements and their relationships.

The manipulation of architectonic elements, their properties and their relationships is possible by means of PCScheme\textsuperscript{2} procedures, which can be fully expanded. At the present stage of development, the available procedures allow for operations of the following kind:

**creation:**

procedure CREATE-ELEMENT; defines the architectural primitives.

**instantiation:**

procedure MAKE-ELEMENT; generates representations of architectonic elements from the architectural primitives.

\textsuperscript{2} Scheme is a dialect of LISP, a programming language appropriate to work with procedural and data abstraction, the description of processes and symbol manipulation. LISt Processing language is widely used in programs that involve processing of symbols, from mathematical symbols to the symbols that form natural (spoken) languages. It is one of the main languages used in Artificial Intelligence research. Being an interpreted language, it allows for incremental expansion of the program while running it, what makes it very appropriate to support design processes. Scheme offers an open, dynamic, and accessible way to build up complex structures out of very simple ones. An introduction to LISP unique features can be found in [Abelson and Sussman 1985, pg 2-4] or in [Winston and Horn 1984, pg 5-8].
a computational environment for architectural design

insertion:

- procedure INCLUDE-PROPERTIES; is used to add a property(ies) to an architectural primitive, and all future instances generated from the primitive will share the(se) property(ies), or to add a property(ies) to the representation of a specific architectonic element.

- procedure INCLUDE-COMPONENTS; stores in the architectural primitives the names of the components that will be specified for the architectonic elements; it can also be used to add a particular component to the representation of a specific architectonic element.

- procedure INCLUDE-PROPERTY-RETROPROPAGATION; differently from INCLUDE-PROPERTIES, this procedure adds the new property not only to the architectural primitive but also to its existing instances that have already been generated.

- procedure INCLUDE-COMPONENT-RETROPROPAGATION; it is similar to the previous procedure, but it deals with components instead of properties.

assignment:

- procedure ASSIGN-PROPERTY; it is used to assign, reassign or unassign the value or the description of a property in a representation of an architectonic element.

- procedure SPECIFY-COMPONENTS; it is used to specify, respecify or unspecify components in the representations of architectonic elements.
deletion:
  procedure DELETE-PROPERTY; it deletes a property from a representation.
  procedure DELETE-COMPONENT; it deletes a component from a representation.

retrieval:
  procedure GET-PROPERTY; this procedure returns the value or the description of a specific property of a representation.
  procedure LIST-OF-PROPERTIES; it returns the list of properties in a representation of an architectural primitive or of an architectonic element, with their corresponding current values or descriptions.
  procedure LIST-OF-COMPONENTS; it returns the list of components of a representation, with their corresponding current specification.

inquiry:
  procedure LIST-OF-OFFSPRINGS; it returns a list of all the instances of architectonic elements generated from a specific architectural primitive.
  procedure STATE?; it returns the current state of a representation.
4.2 Definition of Architectural Primitives

Architectural primitives are defined by three lists, the first of which contains the list of all the representations of architectonic elements that have been generated from the primitive. The second list enumerates the properties that the representations of architectonic elements generated by the primitives will share. The third list names the components to specify in the architectonic element. Properties may have either values or descriptions, components have objects (architectonic elements). There are four possible types of architectural primitives: a) architectural primitives representing architectonic elements that have just a name but no properties and no components; b) architectural primitives that have properties, but does not present any component; c) architectural primitives with components but no properties; d) architectural primitives with properties and components.

Let us elaborate on one of the possible options for an architectural primitive. Suppose the architect wants to define an architectural primitive whose name will be "room". Here is what has to be done:

```plaintext
===> CREATE-ELEMENT
###
    identifier: room
    element-type: class-prototype
    list-of: room
    list-of-properties
        area: unassigned
        height: unassigned
    list-of-components: none
```
For clarity purposes, the real Scheme syntax has been translated to the following conventions that will be used along the examples: 1) bold capital letters stand for the name of the procedure; 2) the sign "===>" stands for the user input, i.e. what the user types; 3) the sign "##>" stands for the machine output, i.e. what the machine returns. Those who are interested in the Scheme code should refer to Appendix 1.

Now the machine 'knows' that "room" is associated to a 'frame' expressing certain characteristics. A frame is a standard information storage device. "Each frame is a table (possibly empty) of bindings, which associate variables with their corresponding values (a single frame may contain at most one binding for any variable)." [Abelson and Sussman 1985, pg 184]. The description contained in the slots is the information that will be used to generate instances of "room". A frame system provides the capability of coordinating information in such way that instances of the same class can be identified from different points of view. The referred frame begins with a header containing a

---

3 In Scheme the basic entities are word-like objects called atoms, which can be numbers (e.g., 4, 40, etc.) or symbols (e.g., *, + column, etc.). Groups of atoms form sentence-like objects called lists. A list consists of a left parenthesis, followed by any integer number of atoms or lists, and a right parenthesis. What goes within the parenthesis are the elements of the list - e.g., (2 3) has three elements: 2, 3 and 3. We call the leftmost element in the list the 'operator', all other elements we call 'operands'. In Scheme, the procedure, that is, the thing to do, is always specified first, followed by the things the procedure is supposed to work with, the arguments. This is the reason for the prefix notation, as opposed to the infix notation commonly used in ordinary mathematics - for instance, in Scheme the expression 'three plus five' is represented as follow (+ 3 5). In Scheme, the sentence 'to square a number is to multiply it by itself' is written as follow:

(define (square x) (* x x))

4 Ideally, what the user types should be minimized. Procedures, or commands, should be picked from dynamic on-screen menus that would appear only when the user wants them.
tag called 'identifier', the name given to the architectonic primitive - "room" in this example. The header also contains a taxonomical label 'element-type' which identifies the class an architectural primitive belongs to, and another label, 'list-of', which enumerates all instances generated from the architectural primitive. Each architectonic element is represented only once as a single object, but its 'identifier' may appear as a component in an element of a higher hierarchical level. The structure of an architectonic element is not repeated every time the architectonic element is referenced. 'Element-type' and 'list-of' are related to the hierarchical organization of elements in classes, subclasses and individuals, as we shall see.

Since an architectonic element may be described by a number of different properties, an architectural primitive must have the capability of being associated to a variable number of properties. By the same token, an architectural primitive can be associated with a variable number of components, so the procedure 'CREATE-ELEMENT' can take any number of arguments beyond one. The first argument is understood by the machine as the name of the primitive; the remaining arguments are considered to be the properties associated to the primitive.

'CREATE-ELEMENT' is implemented this way to reflect the fact that the knowledge the architect has about an architectural primitive varies during the design process. Therefore, properties can be added to architectural primitives later on during the design process.
Suppose the architect wants to define what components are going to be included in "room".

```plaintext
==>
INCLUDE-COMPONENTS  room  wall  wall  wall
                     wall  door  window

##>  list-of-components
    wall:  unspecified
    wall:  unspecified
    wall:  unspecified
    wall:  unspecified
    door:  unspecified
    window: unspecified
```

Now that the list of components has been included, the machine has the capability to 'know' that the architectural primitive "room" has six components (four walls, a door and a window) whose settings will be specified later. Names of components are no more than a label for the slot(s) where the information will be stored - such names are, in fact, names of variables. However, the variables require previously defined architectonic elements as potential values; that is, every architectonic element assigned to one slot in the list of components has to be known already by the machine. A type-matching device makes sure that an architectonic element specified as a component of another element belongs to the corresponding class defined by the name in the slot ("wall-1" cannot be specified to the slot for a "door"). As a consequence, the architect needs not to be concerned with the problem of misassignment. When explaining the insertion procedures, we will see that the architect is not allowed to specify not yet existent architectonic elements as components.
As illustrated, if a variable has no value assigned to it, or if a component has no element specified to it, the terms 'unassigned' and 'unspecified' are associated to them, respectively. The terms 'unassigned' and 'unspecified' reflect the fact that, even if knowing when creating the primitive that certain sets of properties and configurations\(^5\) of components are relevant to the design problem, the architect does not know yet what particular values or specific components will be preferred to define the proposed solutions. 'Unassigned' and 'unspecified' not only occupy the places that later on will be filled in by values and components, but also reflect the structure the architect has defined to progress through the design process. As we shall see, 'unassigned' and 'unspecified' are just particular settings of variables whose value may change during the design process.

\(^5\) "Configurations are sets of elements with certain position relations.... A configuration is described by its selection and by its distribution. The selection identifies the elements in the configuration. The distribution consists of position constraints on the element named in the selection. Each element in the selection may itself be a configuration of smaller elements. An element may be either a space, or virtual element, or it may be a physical, or material one. An element can also belong to more than one configuration." [Gross 1985, pg 60-61].
4.3 Organization of Architectural Primitives

So far for the definition of architectural primitives. But how are the architectural primitives organized? What can we do with such primitives? Architectural primitives are used to generate computational representations of architectonic elements. Such representations will inherit sets of properties and configurations of components from their class-prototypes through a pyramidal lattice-like organization of the architectural primitives.

The proposed organization for the architectural primitives is based on the concepts of class, subclass and individual\(^6\) (Figure 4.1). Classes, subclasses, and individuals are organized in a 'kind-of' pyramidal lattice. This way, relationships between classes and subclasses may be expressed, and properties may be inferred through such relationships. Moreover, properties, as well as components may be inherited. This structure implies that all the representations generated from an architectural primitive will belong to a class because they share a common description, since the architect wanted it so.

\(^6\) It is possible to find the names class, subclass and instance, or prototype, instance and individual being used to define the same concepts. Classification hierarchy is an important topic in knowledge representation systems. For a more comprehensive introduction to this topic the reader should refer to [Gross 1985], [Bobrow and Winograd 1977], or [Smith 1978]. For issues related to knowledge representation systems the reader should refer to [Brachman and Levesque 1985], specially to the papers "Epistemological problems of artificial intelligence" [McCarthy 1977], "On the epistemological status of semantic networks" [Brachman 1979] where a comprehensive survey and analysis of semantic network schemes is presented, and "What's is in a link: foundations for semantic networks" [Woods 1975] where basis for the development of semantic network schemes are introduced.
towards meaningful computational descriptions of architectural form
Properties of a class will be as well be properties of all its subclasses, except for exceptions. As we have seen in Chapter 3, representations may have essential and accidental properties. Accidental properties supplement and/or overrule the properties of the class-prototype. In such a case, we say that the instance is an exception to its class. *The entire set of properties of any instance is the union of its private properties and the properties it inherits from its prototype*. [Gross 1985, pg 70].

Within the proposed structure, there are two possible kinds of instances: subclasses and individuals. Although it may be a little bit confusing regarding the common use of the word 'instance'⁷, it is this distinction that will allow the architect to establish a taxonomy by means of subclasses of subclasses, or instances of instances. This distinction represents the combination of two important features, which are made available to the architect: a) the capability to define more specific and detailed classes of architectonic elements during the design process; b) the capability to alter the specifications of an architectonic element while designing⁸. In Figure 4.1, "gypsum-wall" is a subclass of "The general idea (of instantiation) is the association of a particular individual with the class of which it is a member, and in most notations, this is reflected by the construction of an individual description based on a generic description that the individual satisfies. Thus, while we primarily think of instances as things in the world that are manifestations of our abstract concepts, the term instantiation is very often used to refer to the production of a description of an individual based on a more general description." [Brachman 1979, pg 195].

⁷ "This capability reflects what Brachman calls 'manifestation'. According to him, object's epistemology are permanent. However, they do change over time (e.g., 'a wall may, at different times, be made of a different material, with different color, thickness, height, length, etc.). "Manifestations are different concepts of the same object at different places or
"partition-wall", but it could have been the case that it was an instance of "wall". By the same token, properties could have been added to "wall-2", and it could have been instantiated. Then, "wall-2" would not be an individual anymore, but a subclass.

Subclasses are classes of a lower level and, as such, establish categories within a class. Subclasses behave similarly to classes in that all the elements of a particular subclass share the same description (properties and components). In the diagram in Figure 4.1 "wall" is the class and "partition-wall" and "load-bearing wall" are subclasses. "Gypsum-wall" and "wooden-wall" are subclasses of "partition-wall"; in fact, "partition-wall" performs the role of class for both "gypsum-wall" and "wooden-wall". Subclasses are different from classes in the inheritance of properties; being at the highest level of the lattice, classes do not inherit descriptions.

Within the proposed organization, the inheritance mechanism is crucial. Without such a mechanism, architectural primitives of a lower-level class could not access the information stored in other architectural primitives of a higher-level class. What this means is that the architect would have to declare the properties and the components for every single architectonic element, independently of its level. The architect would be faced with the impossible task of specifying all the characteristics of the design object, without taking any...
advantage of the 'intelligence' of the machine (Figure 4.2). Enabling each representation of lower-level architectonic elements to say how they differ from their superiors, the inheritance mechanism provides the lattice structure with the necessary means to restrain a large overhead of redundant information. As such, inheritance provides the means for the implementation of default values.

Individuals are instances that can not be further instatiated. In general, it is possible to say that an individual is such because it is placed at a specific position in space. For example, the eight cross-shaped columns in the Barcelona Pavillion (Figure 4.3) by Mies van der Rohe could be understood as eight different individuals of a particular subclass of columns: same material, same dimensions, different positions.

A diagram may clarify the concept (Figure 4.4). At the top of the diagram we can see the class prototypes: column, wall, room, etc. Going down in the lattice we see a particular taxonomy for the subclasses of the class "column", arranged according to material, shape, strength, height and color. At the lowest level, there are the individuals which are differentiated from each other by their position, and which share certain properties inherited from their class and subclasses.
a computational environment for architectural design

Figure 4.2. towards meaningful computational descriptions of architectural form
Figure 4.4.

towards meaningful computational descriptions of architectural form
Of course, this is just an example of one way to arrange one such structure. The fact that in this example the properties shape, strength, height and color are arranged hierarchically does not mean that properties have always to be arranged in this fashion (in fact, generally they are not). Responding to different interests, every architect can systematize his own hierarchy which does not need to follow any kind of predefined arrangement, neither regarding the properties used, nor the hierarchical order in which they are organized, nor even the grain or degree of fragmentation of the division.

A subclass can be defined by more than one property. For instance, an I-shaped column made of steel can constitute a subclass if the designer knows he is not going to use other shapes for steel or other materials for I-shape. Notice that the diagram presented in figure 4.4 reflects only the inheritance of properties; a similar diagram could be drawn to show the inheritance of sets of components. For instance, the one in figure 4.5.
a computational environment for architectural design

Figure 4.5.

towards meaningful computational descriptions of architectural form
4.4 Generation of Architectural Representations

Now, let us see how to generate instances of architectonic elements from the class-prototypes. The procedure to be used is called 'MAKE-ELEMENT', and the architect must declare to which class the new element belongs, and its name. Suppose the architect decides to define some subclasses of the previously defined architectural primitive "room". For instance, a "bedroom" which will be a subclass-prototype:

```
==> MAKE-ELEMENT  room  bedroom
##> identifier:  bedroom
element-type:  room
list-of:  bedroom
list-of-properties
  area:  unassigned
  height:  unassigned
list-of-components
  wall:  unspecified
  wall:  unspecified
  wall:  unspecified
  door:  unspecified
  window:  unspecified
```

'MAKE-ELEMENT' takes a variable number of arguments, the first of which is the class to be instantiated. The remaining arguments are the new representations to be generated from the class. The new primitive 'knows' that it belongs to the class of architectural primitive "room", from which it has inherited its description both in terms of properties and components.
Now, if the architect asks the machine:

```plaintext
=== LIST-OF-OFFSPRINGS room
### list-of: room
#### bedroom
```

Notice that "bedroom" has been included in the 'list-of: room' allowing the machine to understand relationships in both directions according to the diagram presented in Figure 4.6.

---

Figure 4.6.
Furthermore, this behavior is consistent when we make instances of instances. For example:

```plaintext
==> MAKE-ELEMENT bedroom master-bedroom
##>
identifier: master-bedroom
element-type: bedroom
list-of: master-bedroom
list-of-properties
  area: unassigned
  height: unassigned
list-of-components
  wall: unspecified
  wall: unspecified
  wall: unspecified
  wall: unspecified
  door: unspecified
  window: unspecified
```

Now, let us examine both "bedroom" and "room":

```plaintext
==> STATE? bedroom
##>
identifier: bedroom
element-type: room
list-of: master-bedroom
list-of-properties
  area: unassigned
  height: unassigned
list-of-components
  wall: unspecified
  wall: unspecified
  wall: unspecified
  wall: unspecified
  door: unspecified
  window: unspecified
```
Now we can understand the role of the taxonomical labels 'list-of' and 'element-type' through the diagram in Figure 4.7. "Room" does not 'know' about the existence of "master-bedroom". By the same token, if the architect instantiates "master-bedroom" different "master-bedrooms" will be created. These new "master-bedrooms" will be known by their parent "master-bedroom" but not by "bedroom" or "room". This may perhaps look clearer through the diagram shown in Figure 4.8.

Master-bedroom-1 will be an individual if it is associated to a definite position in the space. It could also become a subclass itself and generate new offsprings that would inherit its structure of properties and components. Again, this depends on the intentions of the architect using the system.
a computational environment for architectural design

Figure 4.7.

towards meaningful computational descriptions of architectural form
Figure 4.8.

towards meaningful computational descriptions of architectural form
4.5 Manipulation of Architectural Representations

We have seen how to store information in the machine to describe properties and components of architectural primitives. We have also seen how to make instances of these elements and how the instances can be arranged in classes, subclasses and individuals. Furthermore, we have seen how the structures defining architectonic elements in terms of properties and components can be inherited, and how the concept of inheritance is linked to a taxonomy that can be used to organize and control complexity.

However, there is another classification that architects may want to use in order to deal with complexity, that of hierarchies of dependency [Habraken 1983a]. Architectural representations can be organized in a hierarchical structure reflecting the way architectonic elements are commonly arranged. "Proper resolution requires an adequate theory of design decomposition, and the embodiment in CAD systems of knowledge about how architects will want to think about design as hierarchies of elements and subsystems" [Mitchell 1986a, pg 157]. Architectonic elements can be constituted of components, being these components, as well, architectonic elements themselves. For instance, a "column" constituted of a 'base', a 'capital' and a 'shaft' - these last three being also architectonic elements. The representations of architectonic elements have an internal structure organized around slots, some of which are to be filled by 'component-elements'. This provides the system with the capability of representing a 'PART-OF/HAS-A' tree-like lattice structure that the architect can arrange and modify according to his design intentions. In Figure 4.9 we can see towards meaningful computational descriptions of architectural form
a representation of such a lattice structure combined with the diagram of Figure 4.1 representing the lattice of classes, subclasses and individuals.

The proposed organization allows the architect to define not only different classes and subclasses but also to develop different relationships between classes, subclasses and individuals. We may have:

** a class that is a subclass of another class (e.g., 'gypsum-wall is a subclass of partition-wall');
** individuals that are elements of a class (e.g., 'wall-1 and wall-2 are architectonic elements of the class partition-wall');
** a relationship between individuals where one is a component of the other (e.g., 'door-1 is a component of wall-2');
** individuals of diverse classes referring to other individuals of higher level classes (e.g., 'window-1 has to be, at least, one meter apart column-1').

It is time to demonstrate that the structures the proposed environment supports can deal with continuously changing information, so they can be used dynamically during the design process. Let us see, first, how to assign values to the properties of an element and how to specify its components. To illustrate this point suppose that "master-bedroom-1" has been previously defined:
a computational environment for architectural design

Figure 4.9.

towards meaningful computational descriptions of architectural form
'ASSIGN-PROPERTY' takes as arguments the element, the property and the value to be assigned, in this order. The machine returns the variable with the new value. To specify a component the architect uses the procedure 'SPECIFY-COMPONENTS', which may also take a variable number of arguments, being the first one the element and all the remaining the components to be specified. The machine returns the list of components for the element.
It is important to point out that "door-1" has to be known to the machine, what means that it has to have been previously defined as to represent an architectonic element. If that were not the case, the machine would return an error message. We can also specify more than one component at a time, since the procedure 'SPECIFY-COMPONENTS' accepts a variable number of arguments. Suppose that the architect wants to specify the previously defined architectonic elements "wall-1", "wall-2", and "wall-3" to "master-bedroom-1":

```plaintext
==>
SPECIFY-COMPONENTS    master-bedroom-1    door-1
#
list-of-components
# wall:    unspecified
wall:    unspecified
wall:    unspecified
wall:    unspecified
door:    door-1
window:  unspecified
```

The three walls fill in the first three slots available for "wall" according to the order they have been specified. But, what if it is necessary to specify an architectonic element for which space has not been provided? For instance,

```plaintext
==>
SPECIFY-COMPONENTS    master-bedroom-1    wall-2    wall-1
#
list-of-components
# wall:    wall-2
wall:    wall-1
wall:    wall-3
wall:    unspecified
door:    door-1
window:  unspecified
```
suppose that the architect needs two more walls in "master-bedroom-1": "wall-4" and "wall-5" (which were also previously defined):

```plaintext
 ==> SPECIFY-COMPONENTS  master-bedroom-1  wall-4  wall-5
 ##>  list-of-components
     wall:  wall-2
     wall:  wall-1
     wall:  wall-3
     wall:  wall-4
     wall:  wall-5
     door:  door-1
     window:  unspecified
```

Notice that the machine identifies what type of element "wall-5" is (a "wall") and creates the appropriate slot for it in the proper position, allowing the architect to increase the amount of information about the design object. Before the architect wanted the 'bedroom' to have four walls, now he wants five. The design object may be altered later by means of addition or deletion of properties and/or components. Besides, as the example demonstrates, the machine has also to be 'intelligent' enough to identify where the new slot should be placed. It would not have made any sense if the machine had placed "wall-5" in the list of components before "wall-4", or between the components 'door' and 'window', or after the component 'window'.
4.6 Inheritance of Default Settings

Assigned values for properties and specified components in elements can be used as 'default' settings for their instances since they are also inherited. For example, refer once more to the already defined architectural primitive "room" and suppose also that now the architect wants to define some default settings (the height of the room and the type of the door) to be shared through inheritance by the instances of the class "room". Before setting any value the architect inquires about the properties and components of "room".

```plaintext
==>
LIST-OF-PROPERTIES  room

### list-of-properties
area: unspecified
height: unspecified

==>
LIST-OF-COMPONENTS  room

### list-of-components
wall: unspecified
wall: unspecified
wall: unspecified
wall: unspecified
door: unspecified
window: unspecified

Then he fixes the value of a property and specifies a component.

```plaintext
==>
ASSIGN-PROPERTY  room height 9

### height: 9
```

Towards meaningful computational descriptions of architectural form.
Suppose now that the architect has to create another architectural primitive, which is called "bathroom", and it is going to be a subclass of "room":

```plaintext
==> MAKE-ELEMENT room bathroom
##> identifier: bathroom
element-type: room
list-of: bathroom
list-of-properties
area: unassigned
height: 9 inherited-default-setting
list-of-components
wall: unspecified
wall: unspecified
wall: unspecified
wall: unspecified
door: door-type-1 inherited-default-setting
window: unspecified
```

We see that "bathroom" has inherited the settings from its parent "room". The flag 'inherited-default-setting' tells us (and the machine) that such values can be replaced at any time by other default settings or by specific settings that the designer wants to use. Let us elaborate on this point. First, the
information about the element "bathroom" has to be increased:

```plaintext
==>
ASSIGN-PROPERTY  bathroom  area  45
##>
area:  45

==>
SPECIFY-COMPONENTS  bathroom  window-14
##>
list-of-components
    wall:  unspecified
    wall:  unspecified
    wall:  unspecified
    wall:  unspecified
    door:  door-type-1  inherited-default-setting
    window:  window-14

Then an instance of a "bathroom" is created:

==>
MAKE-ELEMENT  bathroom  main-bathroom
##>
identifier:  main-bathroom
element-type:  bathroom
list-of:  main-bathroom
list-of-properties
    area:  45  inherited-default-setting
    height:  9  inherited-default-setting
list-of-components
    wall:  unspecified
    wall:  unspecified
    wall:  unspecified
    wall:  unspecified
    door:  door-type-1  inherited-default-setting
    window:  window-14  inherited-default-setting
```
We can see that "main-bathroom" has inherited settings from both ancestors, "room" and "bathroom". Let us suppose, however that we need in this particular instance different settings for 'height' and "door":

```latex
==>
ASSIGN-PROPERTY
main-bathroom
height
11

SPECIFY-COMPONENTS
main-bathroom
door-type-3

list-of-components
  wall: unspecified
  wall: unspecified
  wall: unspecified
  wall: unspecified
  door: door-type-3
  window: window-14 inherited-default-setting
```

As we see, user's settings have priority over default settings. The flag 'inherited-default-setting' disappears, so both the designer and the machine know about the procedence of the settings. It is useful that the machine 'knows' which settings are default and which are not. To prove this point suppose that during the design process we decide to alter some settings in the class-prototype "room". As a consequence, knowing where the settings come from, all the instances that had previously inherited default settings will change to the new settings:

```latex
==>
ASSIGN-PROPERTY
room
height
13
```

towards meaningful computational descriptions of architectural form
As a result of this instruction the machine checks the instances of "room" that need to be changed. So, it will check "bedroom", and "bathroom". For clarity purposes let us check what changes are performed by the machine only in relation to "bathroom":

---

**SPECIFY-COMPONENTS**  
room  
round-window

### list-of-components
- wall: unspecified
- wall: unspecified
- wall: unspecified
- wall: unspecified
- door: door-type-1
- window: round-window

---

Bobrow and Winograd [Bobrow and Winograd 1977] call this 'procedural attachment'. This concept states that procedures may be associated with the internal structure of a conceptual object, and that they may be activated under certain conditions.
Notice that the 'height' has changed to 13 which is the new default setting; however, the value of the variable 'window' is still "window-14" because it was specified by the user - and therefore has a higher priority -, and the machine knows it because of the absence of the flag 'inherited-default-setting'. The machine also knows that "main-bathroom" is an instance of "bathroom", so it will check what changes need to be made there:

```plaintext
=> STATE?
#> identifier: main-bathroom
    element-type: bathroom
    list-of: main-bathroom
    list-of-properties
      area: 45  inherited-default-setting
      height: 11
    list-of-components
      wall: unspecified
      wall: unspecified
      wall: unspecified
      wall: unspecified
      door: door-type-3
      window: window-14  inherited-default-setting
```

As we can see, nothing has changed, "main-bathroom" keeps its own settings plus the ones inherited from "bathroom". In this case no setting was inherited from "room" because the component "window-14" and the property 'height' had been specified by the architect; therefore, no changes need to be made. This capability can be used for design explorations through the inheritance of default settings, as it has been explained in Chapter 3. "Defaults are useful when we want to begin with a standard description for all instances..."
and (possibly) return later to specify or change descriptions of particular instances" [Gross 1985, pg 71]. The architect has not to be committed to his decisions, but he can try tentative moves and understand its consequences. At any time he can inquiry the machine about all the primitives and all the elements he has already created. 'ALL-ELEMENTS' is just a 'super-class' that includes every primitive or element that has been created or instantiated. It has the same structure that any other representation, but the list of properties is always empty. Moreover, it cannot be instantiated. 'ALL-ELEMENTS' is created automatically by the machine when it is turned on, unless the architect wants to work with a previous file.

```plaintext
==>
STATE? ALL-ELEMENTS
##>
| identifier: all-elements
| element-type: all-class-prototype
| list-of-properties: none
| list-of-components
    | class-prototype: wall
    | class-prototype: room
        | bedroom
        | master-bedroom
        | master-bedroom: master-bedroom-1
        | bathroom
        | main-bathroom
```

But more flexibility is still needed: what happens if we want to add properties or components to elements? What if we do not need anymore a property or a component?. By using the architectural primitive "room" again this point can be illustrated. Suppose the architect needs an extra property in
every "room" called 'volume'. Before adding any property, the architect verifies what properties he already has in "room".

```plaintext
===> LIST-OF-PROPERTIES  room

##> list-of-properties

area: unassigned
height: 13

===> INCLUDE-PROPERTIES  room  volume

##> area: unassigned
height: 13
volume: unassigned

Similarly, the operation to delete a property that is not needed anymore (e.g., 'area') follows the same pattern.

===> DELETE-PROPERTY  room  area

##> property-deleted: area
height: 13
volume: unassigned

The property 'area' is not part of the class of architectural primitive "room" anymore. Using the same fashion we can include or delete components. For example, suppose that the architect needs a second 'window' in "room":
```
As expected, the new property 'volume' and the new component "window" appear in the last place of their respective lists. If, for whatever reason, the architect does not want four "walls" anymore in his "room", but just three then it is necessary to:

```plaintext
===> DELETE-COMPONENT  room  wall
```
While, if we ask:

```plaintext
==> MAKE-ELEMENT  room  new-bathroom
##> identifier: new-bathroom
element-type: room
class-of: new-bathroom
list-of-properties
   height: 13
   volume: unassigned
list-of-components
   wall: unspecified
   wall: unspecified
   wall: unspecified
   door: door-type-1
   window: round-window
   window: unspecified
```

This last example shows that "bathroom" keeps the previous structure of "room". This way, the architect can keep previous stages of the design and perform the exploration of the alternatives necessary to evaluate different design
situations. However, he may as well be interested in retropropagating the effect of these changes. One very simple reason for that is to keep the consistency between descriptions of architectonic elements. That is made possible by two new procedures, as the following example demonstrates.

```plaintext
==> INCLUDE-PROPERTY-RETROPROPAGATION  room  volume
    height: 13
    retropropagated-property: volume unassigned

==> INCLUDE-COMPONENT-RETROPROPAGATION  room
    list-of-components
    wall: unspecified
    wall: unspecified
    wall: unspecified
    door: door-type-1
    window: round-window
    window: unspecified
```

Now, let us check bathroom:
In the same way that the changes have been propagated to "bathroom", they will also propagate to "main-bathroom". The new structure retropropagates all the way down the lattice, but it does not go up. Suppose that the architect had also created an architectonic element called "main-bathroom-1" whose parent is "main-bathroom". Hence, if the architect says:

```plaintext
===> INCLUDE-PROPERTY-RETROPROPAGATION main-bathroom wall-surface
===> INCLUDE-COMPONENT-RETROPROPAGATION main-bathroom bathtub
```

the new property 'wall-surface' and the new component "bathtub" will be part of "main-bathroom" and of "main-bathroom-1", but not of "bathroom". The architect can make it sure by asking:
As we have seen, changes propagate down in the hierarchy from the point where they were introduced. The diagram in figure 4.10 express this concept graphically.
towards meaningful computational descriptions of architectural form
We have introduced a representation scheme in which the architectural knowledge is organized and stored hierarchically inside a computer in terms of classes, subclasses and individuals. As a desired consequence, lower level architectonic elements inherit their properties and components from the higher level ones. We have also seen how to create and manipulate representations of architectonic elements, and how the architect can defer and alter the information about the object being designed. In the following chapter, we propose some alternatives to further develop the ideas introduced so far. We will present ways to manage the information the architect designs with, and ways to associate the representation of architectonic elements with graphic descriptions.
Chapter 5
Discussion

"If you give it a sense, it make sense"
- Ludwig Wittgenstein

5.1 Considerations

One of the uniquenesses of architectural design is its dynamic character by which information about architectonic elements and their relationships is defined, modified, increased or suppressed during the design process. As the design progresses, the architect specifies more and more the design object. In such a dynamic process, a great number and variety of relations between architectonic elements must be controlled. In a computational design environment, as in any other design environment, information and specifications are constantly changing, so ways to formally express rules and maintain consistency are necessary.

In the previous chapter, we have presented a computational representation framework to represent architectural knowledge. As we have already said, knowledge representation is an essential step to succeed in making computers help architects design. In the proposed knowledge representation framework, information about architectonic elements is stored in sets of variables representing properties and relationships between elements. However, in order to convey design ideas through manipulation of the representations, we
need means to manage them. Independently of the method the architect uses to represent his architectural knowledge and to store his design expertise, a tool to manage the information about architectonic elements and their relationships, and to reconfigure the database at every stage as the design progresses is required. As such a feasible and readily available tool, we will describe our use of the 'Constraint Manager', which has been developed at the School of Architecture and Planning at the Massachusetts Institute of Technology.1.

5.2 A Tool for Managing Architectural Knowledge

During the architectural design process, it is an arduous task to keep track of all the information related to different representations and abstractions involved in the alternatives being explored. The 'constraint model' of designing [Gross 1985][Fleisher and Gross 1984] is appropriate to manage this information, providing the architect with more flexibility and confidence to shift back and forth between the alternatives.

1 "The constraint manager is a set of routines to manage, solve and enforce relations among variables. The constraint manager provides a declarative programming environment in which the programmer models a complex system (the design) by describing its variables and relations. The constraint manager maintains a consistency, keep track of dependencies, solves and satisfies constraints." [Ervin and Gross 1987a].
The Constraint Manager provides a facility for declaring, checking consistency among and enforcing user-defined relations in variables. "Its built-in routines propagate and retract fixes, algebraically solve relations and systems of simultaneous equations, find multiple solutions, optimize objective functions, and handle inequality relations using interval values." [Ervin and Gross 1987a, pg 7]. Among the several capabilities offered by the Constraint Manager, we are fundamentally interested in two: the 'algebraic solver' and the 'logic solver' [Gross et al 1986].

The 'algebraic solver' is useful whenever the architect needs to express relationships between quantifiable variables, as, for instance, dimensions and positions of elements. Any piece of design expertise that can be codified in algebraic expressions can be stored in the computer. Thermal, structural, cost and other technical analyses can be performed this way. The machine can calculate, for example, the insulation factor, the stress and the price of an external brick wall, one foot thick, under certain conditions of temperature, load or construction prices. Moreover, since the expressions refer to the variables that represent the properties of architectonic elements, the machine can be used to make inferences about appropriate dimensions, positions or materials. For instance, it can calculate the thickness of a brick wall to satisfy certain insulation requirements, or it could offer the architect a list of materials that fulfill the structural requirements of a rectangular column of dimensions 'a' and 'b', and

\[2\text{In order to acquire familiarity with the constraint manager and fully understand its capabilities the reader should refer to [Fleisher and Gross 1984], [Gross 1984], [Gross 1985], [Gross et al 1986] and [Ervin and Gross 1987a].} \]
height 'h' under a certain load 'l'. If all but one of the values for the variable is known in an algebraic expression, the remaining value can be established by the algebraic solver. The algebraic solver can also solve sets of simultaneous linear equations and perform interval calculations for upper and lower limits for a variable, thus being able to deal with inequalities as well as equations. Geometric relationships can be translated to algebraic expressions via analytic geometry and, hence, stored in the computer. Exploration on parametrical variations are made available through the 'algebraic solver'.

The Constraint Manager deals not only with bidirectional expressions but also with one-way assignments. The examples in the previous paragraph can be classified as bidirectional, since any of the variables can be isolated and calculated after the others are known. But we may be interested in one-way assignments for specific purposes. For instance, we may want to have the ground floor of a house three feet over the street level, what does not mean that the street level has to move down when the ground floor does; or the external walls of the house must be separated ten feet from the plot boundaries. This hierarchical behavior of some variables reflects the hierarchies between architectonic elements transmitted also to its properties.

---

3 A wall can be located parallel or perpendicular to another wall, or the height of a room can be specified as half its shortest side, for instance

4 In an expression like 'the area of a room equals its length times its width', there are three variables: area, length and width. Fixing any two of them causes the third one to be fixed.
The 'logic solver' takes care of logical implications of the type 'if-then' (production) rules. This kind of propositions allows for assumptions and inferences. The 'logic solver' is, in fact, a pattern-matcher inference engine, similar to the ones used in expert systems. A critical part of the 'intelligence' of an inference engine relies on its capabilities to identify patterns and match them with the propositions. The more 'intelligent' the pattern-matcher, the more powerful the engine. The inference engine could possibly identify certain spatial patterns: 'if there is a porch in the south side of the house, then cover it with a pergola', or 'if there are two windows separated by less than the width of the smaller one, then replace them for a stripe window from the left-end to the right-end', and so on. This spatial pattern-recognition capability would allow for replacement, modification, substitution and other manipulations of spatial configurations. The architect can use it as well to develop shape grammars describing the vocabulary, the configurations the machine must recognize and the production rules to generate new configurations.

The Constraint Manager dialogues with the representations of architectonic elements provided in the proposed environment through the already presented procedures 'GET-PROPERTY' and 'ASSIGN-PROPERTY'. The former is used to get the values of the variables needed, the latter to set the calculated values for the properties. All the variables (properties) involved in a calculation will be asked and set by the constraint manager, since there is where the consistency is checked.

5 Example of its use would be: 'if the column is I-shaped then its material must be steel' or 'if this room is a bathroom then place a toilette bowl'.

towards meaningful computational descriptions of architectural form
The procedure 'SPECIFY-COMPONENTS' together with 'INCLUDE-PROPERTIES' and 'INCLUDE-COMPONENTS' may be also useful when working with the Constraint Manager, specifically to modify the description of an architectonic element after relationships or rules have been defined and applied. For instance, suppose we have two-columns, "column-1" and "column-2". "Column-1" is 'red'; and "column-2" does not have 'color' in the list of its properties:

```plaintext
==>
STATE? column-1
#>
identifier: column-1
element-type: column
list-of:
list-of-properties
  material: unassigned
color: red
list-of-components
  base: unspecified
  shaft: unspecified

==>
STATE? column-2
#>
identifier: column-2
element-type: column
list-of:
list-of-properties
  material: unassigned
list-of-components
  base: unspecified
  shaft: unspecified
```
Then, if we say:

```plaintext
===> ADD-CONSTRAINT [color (column-1)] = [color (column-2)]
###> constraint added
```

Now, if we check column-2:

```plaintext
===> STATE? column-2
###> identifier: column-2
    element-type: column
    list-of: column-2
    list-of-properties
        material: unassigned
        color: red constrained by [color(column-1)]
    list-of-components
        base: unspecified
        shaft: unspecified
```

As we can see, the machine creates a new slot for the property 'color' using the procedure 'INCLUDE-PROPERTIES', and fills it in with the value 'red'. Now, again manipulating the same column-1, suppose we say:

```plaintext
===> ADD-CONSTRAINT ['if element is a column
    then it must have a capital']
###> constraint added
```

If we check again "column-1":

```plaintext
```
The Constraint Manager has used the procedure 'INCLUDE-COMPONENTS' to represent the fact that the column needs a capital. The capability that the Constraint Manager offers to express and keep track of relationships between architectonic elements is fundamental in the association of graphic representations to the descriptions of elements. The next section elaborates on this question, providing, as well, a more comprehensive illustration of the use of the Constrain Manager in the proposed computational design environment.

5.3 Association of Graphic Description with Architectural Primitives

Descriptions of architectonic elements may include graphic representations or not, depending on the particular needs of every design situation. This is particularly true for spatial elements, that can be associated with specific graphic representations (e.g., a room represented as a rectangle, or
as a circle), or can be represented through their physical components (e.g., a room represented through the representations of its walls, windows, doors, etc.). In any case, the architect decides how to represent each architectonic element and defines the manipulations that can be applied to the representations.

Suppose that the architect wants to define an architectural primitive to generate representations of architectonic elements of the class "wall". He has in mind a particular way to express graphically such a "wall" that suits his particular needs for a specific design situation. Suppose also that the architect is not concerned, at this point, with properties of "wall" other than graphic representation. He has available a graphic computational environment where entities can be identified and manipulated. Despite obvious limitations, for clarity and simplicity purposes, we will assume that such environment deals with two-dimensional representations, and that the graphic entities available to the user are points, lines and rectangles. A graphic editor allows for complex manipulation of such entities and also provides a comfortable machine-user interface, via a pointing device (mouse, digitizer, light-pen), on-screen dynamic menus and text window(s). Appendix 2 elaborates in more depth on the characteristics of the graphic environment and illustrates the kind of supported operations.

The architect wants to associate his "wall" to a graphic representation of the class "rectangle". Such graphic representation must 'understand' the "origin" and the "end" of the "wall", its "axis" and its "thickness", so every
"wall" of this class can be expressed and drawn consistently according to these principles. So, he types:

```lisp
===> MAKE-ELEMENT  rectangle  rectilinear-wall
###> identifier:  rectilinear-wall
element-type:  rectangle
list-of:  rectilinear-wall
list-of-properties
  area:  unassigned
  width:  unassigned
  length:  unassigned
list-of-components:  none
```

Rectangle is a computational entity previously defined by the architect or a built-in graphic primitive. Now, the architect expresses the properties of "rectilinear-wall" that will differentiate it from "rectangle".

```lisp
===> INCLUDE-PROPERTIES  rectilinear-wall  starting-point
     ending-point
     length-wall
     thickness
###> starting-point:  unassigned
     ending-point:  unassigned
     length-wall:  unassigned
     thickness:  unassigned
```

Now, using the Constraint Manager, the architect specifies the relationships between the variables:

towards meaningful computational descriptions of architectural form
==>
ADD-CONSTRAINT
[(starting-point rectilinear-wall) =
  (midpoint-left-side rectilinear-wall)]
##>  constraint added

==>
ADD-CONSTRAINT
[(ending-point rectilinear-wall) =
  (midpoint-right-side rectilinear-wall)]
##>  constraint added

==>
ADD-CONSTRAINT
[(length-wall rectilinear-wall) = (length rectilinear-wall)]
##>  constraint added

==>
ADD-CONSTRAINT
[(thickness rectilinear-wall) = (width rectilinear-wall)]
##>  constraint added

Now, the machine knows that every "rectilinear-wall" is associated to a rectangle that will be its graphic representation. It also knows how certain properties (variables) of the "rectilinear-wall" are associated to certain properties (variables) of the rectangle. "Rectilinear-wall" can be manipulated and modified both textually and graphically, and the Constraint Manager will propagate the necessary changes in both directions, keeping the consistency between the representations as seen in the following diagram.
wall <=> rectangle
length-wall <=> length
thickness <=> width
starting-point <=> midpoint-left-side
ending-point <=> midpoint-right-side

Now, the architect can instantiate "rectilinear-wall":

```plaintext
==> MAKE-ELEMENT rectilinear-wall wall-1
##> identifier: wall-1
element-type: rectilinear-wall
list-of: wall-1
list-of-properties
  area: unassigned
  width: unassigned
  length: unassigned
  starting-point: unassigned
  ending-point: unassigned
  length-wall: unassigned
  thickness: unassigned
list-of-components: none
```

Then, the architect can add information about "wall-1":

```plaintext
==> ASSIGN-PROPERTY wall-1 thickness .3
##> thickness: .3
==> ASSIGN-PROPERTY wall-1 starting-point (20 40)
##> starting-point: (20 40)
```
And the result is:

```plaintext
==>
STATE?  wall-1
```  
```plaintext
#>
identifier:  wall-1
element-type:  rectilinear-wall
list-of:  wall-1
list-of-properties
area:  15
width:  .3
length:  50
starting-point:  (20 40)
ending-point:  (60 70)
length-wall:  50
thickness:  .3
list-of-components:  none
```

As we can see, the information provided to the computer has been translated by the Constraint Manager into values for the related variables. Now, "wall-1" can be drawn, and the result can be seen in Figure 5.1.
The architect can manipulate "wall-1", for instance:

```lang
=> ASSIGN-PROPERTY wall-1 thickness .4
```

Once more, all the variables affected by the change in the value of the property 'thickness' have been set to their new values. In order to check if that is so:
When defining "rectilinear-wall", the architect specified the variables describing it, besides the way its graphic representation - in this case, a rectangle - was going to be manipulated. However, the rectangle as a graphic primitive allows for other manipulations. Suppose the architect wants to define a new architectural primitive, called "column", with properties different from those describing "rectilinear-wall".
And the properties specific to "column" to be added to the ones describing "rectangle" will be:

```plaintext
==>
INCLUDE-PROPERTIES column side-a side-b
center-column

```#

```
side-a: unassigned
side-b: unassigned
center-column: unassigned
```#

The relationships between the variables are described as:

```plaintext
==>
ADD-CONSTRAINT
[(side-a column) = (width column)]
```#

```
constraint added
```#

```plaintext
==>
ADD-CONSTRAINT
[(side-b column) = (length column)]
```#

```
constraint added
```#

```plaintext
==>
ADD-CONSTRAINT
[(center-column column) = (center column)]
```#

```
constraint added
```#

As a result, we have the following diagram expressing the association:

towards meaningful computational descriptions of architectural form
column $\leftrightarrow$ rectangle
side-a $\leftrightarrow$ width
side-b $\leftrightarrow$ length
center-column $\leftrightarrow$ center

And, when the "column" is instantiated:

```plaintext
>>> MAKE-ELEMENT column column-1
>>> identifier: column-1
element-type: column
list-of: column-1
list-of-properties
  area: unassigned
  width: unassigned
  length: unassigned
  center: unassigned
  side-a: unassigned
  side-b: unassigned
  center-column: unassigned
list-of-components: none
```

Now the architect makes the position and the size of the element "column-1" definite:

```plaintext
>>> ASSIGN-PROPERTY column-1 center-column (20 10)
>>> center-column: (20 10)

>>> ASSIGN-PROPERTY column-1 side-a 10
>>> side-a: 10
```

towards meaningful computational descriptions of architectural form
And the result is:

```plaintext
==>
ASSIGN-PROPERTY
column-1
side-b 8
```
towards meaningful computational descriptions of architectural form
which can be diagrammed as:

```
  window <=> rectangle
  axis <=> longitudinal-axis
  position-axis <=> transversal-axis
  width-window <=> length
  thickness <=> width
```

And now to make sure the "window" belongs to a "rectilinear-wall":

```
===> INCLUDE-COMPONENTS  rectilinear-wall  window
### list-of-components
        window: unspecified

===> ADD-CONSTRAINT
    [(axis window) superpose (longitudinal-axis rectilinear-wall)]
### constraint added

===> ADD-CONSTRAINT
    [(position-axis window) perpendicular (longitudinal-axis rectilinear-wall)]
### constraint added

===> ADD-CONSTRAINT
    [(thickness window) = (thickness rectilinear-wall)]
### constraint added
```
Now the architect can instantiate "window":

```plaintext
===> MAKE-ELEMENT window window-1
### identifier: window-1
### element-type: window
list-of: window-1
### list-of-properties
  area: unassigned
  width: unassigned
  length: unassigned
  center: unassigned
  axis: unassigned
  position-axis: unassigned
  width-window: unassigned
  thickness: unassigned
### list-of-components: none
```

The architect places the "window-1" on a reference axis:

```plaintext
===> ASSIGN-PROPERTY window-1 position-axis
### position-axis: [(30 80) (60 40)]
```

and assigns a width to "window-1":

```plaintext
===> ASSIGN-PROPERTY window-1 width-window
### width-window: 3
```

If the architect checks:
The properties whose values are affected through propagation have been assigned new values by the Constraint Manager. Suppose now the architect wants to place "window-1" in "wall-1". So, he checks the state of "wall-1":

`===> STATE?  wall-1
##> identifier:  wall-1
element-type:  rectilinear-wall
list-of:  wall-1
list-of-properties
  area:  20
  width:  .4
  length:  50
  starting-point:  (20 40)
  ending-point:  (60 70)
  length-wall:  50
  thickness:  .4
list-of-components
  window:  unspecified
`
The architect places "window-1" as a component of "wall-1":

```plaintext
=> SPECIFY-COMPONENTS    wall-1    window-1
### list-of-components
window:    window-1
```

But if the architect checks "window-1":

```plaintext
=> STATE?    window-1
### identifier:    window-1
element-type:    window
list-of:    window-1
list-of-properties
  area: .9
  width: .4
  length: 3
  center: (45 60)
  axis: [(20 40) (60 70)]
  position-axis: [(30 80) (60 40)]
  width-window: .3
  thickness: .4
list-of-components: none
```

As we can see, by placing the variables of "window-1" in relation to those in "wall-1", the Constraint Manager can infer by calculation the related values in "window-1" from the ones already provided for "wall-1". Now, the architect can draw, and the can be seen in Figure 5.3.
towards meaningful computational descriptions of architectural form
5.4 Multiple Descriptions

With very simple examples we have seen how to use the Constraint Manager to keep track of relationships between representations of architectonic elements, specifically in the case of association of descriptions with graphic entities. However, a problem arises, the one of consistent taxonomy. As we have explained in the previous chapter, representations of architectonic elements are organized according to a classification of classes and subclasses that is reflected in their descriptions. But if representations that are built up from graphic representations by adding properties to the existing descriptions have to follow the same taxonomy, we run into a conflict. If "rectilinear-wall" is a "rectangle", that means that "rectangle" is a class-prototype to "rectilinear-wall". This implies that other graphic representations could not be used for "rectilinear-wall", and that the top of the pyramidal organization is occupied by the graphic description, as shown in Figure 5.4.

If such is the case, a "column" cannot be either a "rectangle" or a "circle" while being at the same time a "column". To do this, the previous diagram in Figure 5.4 should look like the one in Figure 5.5.
towards meaningful computational descriptions of architectural form
But then, a "rectangle" has to be a "column", a "rectilinear-wall", or a "window", and the pyramidal lattice is upside-down. In order to solve this contradiction, we plan to work with multiple descriptions, within which a representation can inherit descriptions from more than one ancestor. This way, a "circular-column" can be a "column" and a "circle", while a "rectangular-column" can be, at the same time, a "column" and a "rectangle" (Figure 5.6).

![Diagram of multiple descriptions](image)

Figure 5.6.

Multiple descriptions of architectonic elements may solve the problem of conflicting inheritance. While a concept denotes at most one object at a time, the same object can be denoted by several descriptions (Figure 5.7). Some of these descriptions express qualities, others express facts, while others express additional propositions about the design object. *Multiple descriptions containing redundant information are used in the human representation system towards meaningful computational descriptions of architectural form*
to trade off memory space for computation depth; computer systems can take advantage of the same techniques." [Bobrow and Winograd 1977, pg4]. The use of multiple inheritance mechanisms provides as well a more efficient way to organize information, avoiding unnecessary redundancies.

**Figure 5.7.**

Descriptions can combine different modes of description [Bobrow and Winograd 1977], such as:
- assigning an object to membership in a category
  ("element 'A' is a wall");
- stating its role in a complex object or event
  (the structural role of a wall);
- providing a unique identifier
  (for example, this includes using a name like: "wall-1");
- stating a relationship in which the object is a participant
  ("wall-1" is parallel to the row of columns);
- asserting a complex logical formula which is true of the object
  ("either "wall-1" is a structural wall made of reinforced concrete,
  or it is not parallel to the row of columns");
- describing an object in terms of membership in a set, or a set in terms
  of the object it contains
  ("one of the four possible types of wall");
- combining these other descriptors into time-dependent or contingent
  descriptions
  (the location of an element at a particular moment of the design).

Multiple descriptions seem potentially contributive for the progress of
our embryonic computational design environment. If they prove to be so, the
architect will be able to express complex relationships in a structured frame (fig 5.8). Multiple descriptions can also provide the means to accommodate different representations (graphic or not) for a same object, fulfilling one of the intended goals of the environment we have introduced.
5.5 Summary

Rather than being a conclusion, this chapter has introduced proposals for further research. These proposals have not been tested and they should be. We believe they are worth trying. Even if it proves our proposals to be wrong, we believe such effort will provide a better understanding of how to carry on architectural design processes in computational environments.
The proposed environment wants to be a tool to design better, not directly a tool to produce better designs. We would like it to be a kind of ‘form processor’, analogous to already well known word processors. A word processor is not designed to produce better writing, but to help write better. When human beings have better tools, however, they usually produce better results. Whether we want to teach computers how to design is a different issue. Nevertheless, if we are concerned about design quality, we may as well ask if it is possible (and desirable) to teach a computer to write Shakespeareanlike masterpieces. Whether Sheakeapeare would have used a word processor is an unsolvable conjecture.

Our representation of reality has been based on the assumption that the world can be described in terms of discrete elements and the relationships between them. The limitations of any attempt to constrain the complexity of our universe to the restrictions of human brains have been proved evident before in many occasions by distinguished authors, and we are not going to try to emulate those arguments here. However, we would like to point out that ours is not an atomistic view of the world. On the contrary, our hierarchical organization gives way to a holistic approach, where complexity can be expressed and tackled at any level, as we believe architects do when designing. Whether it is a city or a dining-table, architects have in mind a whole network of relationships where to position their design objects. Such network is, at least, as important as the elements located in it, thus our efforts to reflect it in our computational design environment.
In this work, we have shown with very simple examples how the computer can provide the means for the architect to create his own architectural primitives according to his specific needs in front of a particular design situation. Then he can use these primitives to generate representations of architectonic elements which he will be able to define, manipulate and transform according to his will. The environment also provides the capability of making explicit relations between entities, so these relations can also be defined, manipulated and transformed. The tool has been presented, even if in an embryonic state, and we believe it can help the architect design. It is not a finished product. However, it is important to notice here not the limitations of the examples that have been presented, or the incompleteness of the proposed system, but rather the understanding of what can be generalized regarding the issues raised throughout this work.
Appendix 1

Scheme Code

(1) General selectors

(define first car)
(define second cadr)
(define third caddr)
(define fourth cadddr)
(define rest cdr)
(define description cddr)

(2) General selectors to retrieve information about an architectural primitive or an architectonic element.

(define (header-element element) (first element))
(define (list-of-offsprings element) (second element))
(define (list-of-properties element) (third element))
(define (list-of-components element) (fourth element))

(define (identifier-element element)
  (first (header-element element)))

(define (get-element-name element)
  (cdr (identifier-element element)))

(define (get-element-type element)
  (cdr (second (header-element element))))
(3) General procedures

The following procedure returns the last element of a list.

```
(define (last list)
  (cond ((null? (cdr list))
          (car list))
        (else (last (cdr list))))
```

The following procedure builds up a new list. It associates each element of a list called 'list-of-things' with a specific argument.

```
(define (cons-list list-of-things argument)
  (cond (null? list-of-things) '( )
        (else (cons (cons (car list-of-things) argument)
                   (cons-list (cdr list-of-things) argument))))
```
(4) The following procedure produces the representation of an architectural primitive whose name will be the same as the argument 'class-prototype', and whose number of properties is variable. It returns a pretty print of the architectural primitive.

(define (create-element class-prototype properties)
  (begin (eval `(define ,class-prototype
                (cond ((null? ,properties)
                       (list (list (cons 'identifier ,class-prototype)
                                (cons 'element-type
                                      'class-prototype))
                       (cons (cons 'list-of ',class-prototype) 'none)
                       (cons (list 'list-of-properties) 'none)
                       (cons (list 'list-of-components) 'none)))
                       (else (list (list (cons 'identifier '
                                      ,class-prototype
                                      (cons 'element-type
                                      'class-prototype))
                                      (cons (cons 'list-of '
                                      ,class-prototype) 'none)
                                      (cons (list 'list-of-properties) 'none)
                                      (cons-list ,properties 'unassigned))
                                      (cons (list 'list-of-components) 'none))))))
  (specify-components-2! ALL-ELEMENTS (eval class-prototype)))
  (state? (eval class-prototype))))
(5) The following procedure includes the name of an architectonic element in the list of instances of its superior, which may be an architectural primitive or a higher level architectonic element.

(define (insert-element! class-prototype element)
  (cond ((eq? 'none (get-offsprings class-prototype))
            (set-cdr! (list-of-offsprings class-prototype)
                (list element)))
       (else (set-cdr! (list-of-offsprings class-prototype)
             (append (get-offsprings class-prototype)
                  (list element))))))

(6) The following procedure instantiates an architectural primitive generating architectonic elements called 'element-name'. It returns a pretty print of the architectonic element.

(define (make-element class-prototype element-name)
  (begin (eval ,class-prototype)
    (eval `(define ,element-name
      (list (list (cons 'identifier ,element-name)
        (cons 'element-type
          (get-element-name',
            class-prototype))
      (cons (cons 'list-of ',element-name) 'none)
    (cons (list 'list-of-properties)
      (assign-default-property
        (get-properties ',class-prototype))
    (cons (list 'list-of-components)
      (specify-default-component
        (get-components ',class-prototype))))
  (specify-component-2! ALL-ELEMENTS (eval element-name))
  (insert-element I class-prototype element-name))
  (state? (eval element-name)))

towards meaningful computational descriptions of architectural form
(7) The following procedure adds property(ies), not its value or description, to the representation of an architectural primitive or of an architectonic element. It returns a pretty print of the list of properties of the primitive or of the element.

```
(define (include-properties! element properties)
  (begin (cond ((eq? 'none (get-properties element))
      (set-cdr! (list-of-properties element)
        (cons-list properties 'unassigned))
    (else (set-cdr! (list-of-properties element)
      (append (get-properties element)
        (cons-list properties 'unassigned))))
  (state? (list-of-properties element))))
```

(8) The following procedure adds component(s), not its specification, either to a representation of an architectural primitive or of an architectonic element. It returns a pretty print of the list of components of the primitive or of the element, after the component(s) has been included.

```
(define (include-components! element components)
  (cond ((eq? 'none (get-components element))
      (set-cdr! (list-of-components element)
        (cons-list components 'unspecified))
    (else (set-cdr! (list-of-components element)
      (append (get-components element)
        (cons-list components 'unspecified))))
  (state? (list-of-components element))))
```
(9) The following procedure assigns the value (e.g., height = 10) or the description (e.g., color: blue) of a property either to an architectonic primitive or to an architectonic element.

\[
\text{(define (assign-property! element property value)}
\]

\[
\text{\hspace{1cm} (set-cdri (get-property element property) value))}
\]

(10) The following procedure assigns a default value(s) to an inherited property(ies) of a new architectonic element. It associates the flag 'inherited-default-setting' to the value(s).

\[
\text{(define (assign-default-property list-of-properties)}
\]

\[
\text{\hspace{1cm} (define (assign-default-property-aux first-list-of-properties)}
\]

\[
\text{\hspace{2cm} (cond ((null? first-list-of-properties) '( ))}
\]

\[
\text{\hspace{2cm} (else}
\]

\[
\text{\hspace{3cm} (cond ((or (eq? 'unassigned (cdr first-list-of-properties))}
\]

\[
\text{\hspace{4cm} (and (not (atom? (cdr first-list-of-properties)))}
\]

\[
\text{\hspace{5cm} (eq? 'inherited-default-setting}
\]

\[
\text{\hspace{6cm} (last first-list-of-properties)))}
\]

\[
\text{\hspace{5cm} first-list-of-properties)}
\]

\[
\text{\hspace{4cm} (else (list (car first-list-of-properties)}
\]

\[
\text{\hspace{5cm} (cdr first-list-of-properties)
\]

\[
\text{\hspace{6cm} 'inherited-default-setting))}))}
\]

\[
\text{\hspace{2cm} (cond ((eq? 'none list-of-properties) 'none)
}\]

\[
\text{\hspace{2cm} (else (mapcar assign-default-property-aux
}\]

\[
\text{\hspace{3cm} list-of-properties)))
}\]

\[
\text{\hspace{1cm})})})})}
\]

\[
\text{towards meaningful computational descriptions of architectural form}
\]
(11) The following procedure specifies one or more components of a representation of an architectural primitive, or of an architectonic element. The specified component must be a previously defined architectonic element, otherwise a filter will prevent a misassignment. It returns a pretty print of the list of components of the primitive or of the element.
(define (specify-components element . components)
  (define (specify-components-aux1 element components-list)
    (define (specify-components-aux2 components-in-element component)
      (cond ((null? components-in-element)
             (set-cdr! (list-of-components element)
                        (append (get-components element)
                                 (list (cons
                                       (get-element-type component)
                                       (get-element-name
                                        component))))))
            ((eq? 'none (get-components element))
             (set-cdr! (list-of-components element)
                        (list (cons
                               (get-element-type component)
                               (get-element-name
                                component))))))
            ((and (eq? (get-element-type component)
                    (car (first components-in-element)))
                  (eq? 'unspecified
                    (cdr (first components-in-element)))
                  (set-cdr! (first components-in-element)
                            (get-element-name
                             component)))))
          (else (specify-components-aux2
                 (cdr components-in-element) component))))
    (cond ((null? components-list)
            (state? (list-of-components element)))
          (else (specify-components-aux2
                 (get-components element)
                 (first components-list))
                 (specify-components-aux1
                  (rest components-list))))))
  (specify-components-aux1 element components))
towards meaningful computational descriptions of architectural form
(12) The following is an auxiliary procedure to the previous one. It is also used by 'CREATE-ELEMENT' to include the new architectural primitive, and by MAKE-ELEMENT to include the new architectonic element in the list 'ALL-ELEMENTS'.

(define (specify-components-2! element . components)
  (define (specify-components-aux1 element components-list)
    (define (specify-components-aux2 components-in-element
      component)
      (cond ((null? components-in-element)
        (set-cdr! (list-of-components element)
          (append (get-components element)
            (list (cons
              (get-element-type component)
              (get-element-name
                component)))))))
      ((eq? 'none (get-components element))
        (set-cdr! (list-of-components element)
          (list (cons (get-element-type component)
            (get-element-name component)))))
      ((and (eq? (get-element-type component)
        (car (first components-in-element)))
        (eq? 'unspecified (cadr (first
          components-in-element))))
        (set-cdr! (first components-in-element)
          (get-element-name
            component)))
      (else (specify-components-aux2
        (cadr components-in-element) component))))
    (cond ((null? components-list) newline)
      (else (specify-components-aux2
        (get-components element)
        (first components-list))
        (specify-components-aux1
          (rest components-list))))
    (specify-components-aux1 element components))

towards meaningful computational descriptions of architectural form
The following procedure performs a default specification to a component(s) of a new architectonic element, and associates to it the flag 'inherited-default-setting'.

```
(define (specify-default-component list-of-components)
  (define (specify-default-component-aux first-list-of-components)
    (cond ((null? first-list-of-components) '( ))
      (else
        (cond ((or (eq? 'unspecified (cdr first-list-of-components))
                        (and (not (atom? (cdr first-list-of-components)))
                         (eq? 'inherited-default-setting (last first-list-of-components)))
              first-list-of-components)
              (else (list (car first-list-of-components)
                          (cdr first-list-of-components)
                          'inherited-default-setting))))))
    (cond ((eq? 'none list-of-components) 'none)
      (else (mapcar specify-default-component-aux list-of-components))))
```
(14) The following procedures return a list of all instances generated from an architectural primitive, the properties of an architectural primitive or an architectonic element, and the components of an primitive or an element, respectively.

\[
\text{(define \ (get-offsprings \ element) \ (cdr \ (list-of-offsprings \ element)))} \\
\text{(define \ (get-properties \ element) \ (cdr \ (list-of-properties \ element)))} \\
\text{(define \ (get-components \ element) \ (cdr \ (list-of-components \ element)))}
\]

(15) The following procedure returns an specific value or description of a property of an architectural primitive or of an architectonic element.

\[
\text{(define \ (get-property \ element \ property) \ (assq \ property \ (get-properties \ element)))}
\]

(16) The following procedure returns a pretty print of an element. It may be an architectural primitive, an architectonic element, the list of properties, etc.

\[
\text{(define \ (state? \ element) \ (pp \ element))}
\]
(17) The following procedure creates the 'super-class' ALL-ELEMENTS, where all primitives and instances will be included.

(define (create ALL-ELEMENTS)
  (eval `(define ALL-ELEMENTS
    (list (cons 'identifier 'ALL-ELEMENTS)
      (cons 'element-type 'all-class-prototype))
    (cons (cons 'list-of 'ALL-ELEMENTS) 'none)
    (cons (list 'list-of-properties) 'none)
    (cons (list 'list-of-components) 'none))))

(create ALL-ELEMENTS)
Appendix 2

The Graphic System

Architects explore design ideas by creating representations using various different media, then transforming these representations. Design media are not neutral; each medium has different properties, strengths and weaknesses that make it suitable to represent some things better than others, that make some kinds of transformation easy while others hard, that encourage some kinds of explorations and discourage others, and that lead the imagination in different directions.

While the properties, advantages and limitations of traditional media are sufficiently known, computational media can be defined to suit different design needs, offering new ways for architectural understanding and speculation, and potentially leading to unknown architectural worlds.

A computational environment for architectural design should have more than one graphic environment. The architect may want to use a three-dimensional solid modelling system, or a very structured two-dimensional drafting package, or an on-screen bit-map capability where operations can be performed on unstructured pixel representations of drawings produced in other graphic systems or even scanned images from diverse origins. Each environment allows for specific operations, while different kinds of images can
be projected one on top of each other on the same screen to help the architect understand the consequences of his design decisions.

In this appendix, we will use a very simple graphic system to illustrate the graphic capabilities that we devise for the proposed computational environment for architectural design. Any of the computer graphic systems should have built-in graphics primitives that can represent entities of variable complexity. These primitives must be easily accessible to the user, as well as the procedures to operate manipulations and transformations (e.g., translation, rotation, scaling, etc.).

The graphic entities in the proposed environment can be used simply as such, with the help of the typical editing tools available in current computer-aided drafting systems, and the architect does not need to be concerned about their ulterior capabilities. The system is then being used as a plain drafting tool. However, it can do better than that. Graphic entities are something else than just a list of parameters that tell the machine how to draw, transform and manipulate them. The representation is 'intelligent' enough to identify specific entities and their geometrical properties, so manipulations and transformations can be made in many different ways.
towards meaningful computational descriptions of architectural form
To illustrate this point, suppose one of the built-in graphics environment available is a two-dimensional package. Suppose also that this particular two-dimensional system is based on a hierarchical organization of points, lines and polygons. The point is the simplest entity in this environment and it is represented in cartesian or polar coordinates (Figure A.2.1). Lines go from point to point, so each of the extremes points can be operated independently as a point (Figure A.2.2).

Besides, other significant points can be identified on the line: the midpoint, the point that is placed at a certain ratio from one of the extreme points, etc. For example (Figure A.2.3), there are two lines on the screen, L1 and L2, and suppose that the architect wants to intersect L1 and L2 so the point located at 1/3 of the length of L1, measuring from P1 to P2, is translated to the midpoint of L2.
[translate-line L1 (vector-translation
  (get-point-line L1 (1/3 (distance L1P1 L2P2)))
  (get-point-line L2 (1/2 (distance L2P2 L2P1))))]

Figure A.2.3.
Following the same pattern, a polygonal shape would be constituted of lines that go from point to point, being the vertices of the polygonal shapes the points of two consecutive lines that are constrained, with the help of the Constraint Manager, to have the same spatial coordinates (figure A.2.4)

![Diagram](image.png)

Figure A.2.4.

Of course all these operations can be performed more easily with a pointing device than with the keyboard, but the issue here is to explain the concepts of representation and manipulation of graphics primitives rather than elaborating on user interface systems. In the present context, text is more appropriate to show and understand the communication between the architect and the computer.
With the operations presented so far it is possible to build up more complex entities. For instance, we can define a rectangle as a set of four lines constrained to have common spatial coordinates in the four vertices, as well as constrained to have perpendicular lines (Figure A.2.5).

Then we can define also:

\[
\begin{align*}
\text{length-rectangle} &= \text{length } L1 = \text{length } L3 \\
\text{width-rectangle} &= \text{length } L2 = \text{length } L4 \\
\text{top-left-corner-rectangle} &= L1P2 = L2P1 \\
\text{top-right-corner-rectangle} &= L2P2 = L3P1 \\
\text{bottom-left-corner-rectangle} &= L1P1 = L4P2 \\
\text{bottom-right-corner-rectangle} &= L4P1 = L3P2 \\
\text{top-side-rectangle} &= L2 \\
\text{bottom-side-rectangle} &= L4 \\
\text{left-side-rectangle} &= L1 \\
\text{right-side-rectangle} &= L3 \\
\text{area-rectangle} &= (* \text{length-rectangle} \text{width-rectangle})
\end{align*}
\]

Furthermore, it is possible to define more complex things:

\[
\text{center-rectangle} = \text{intersection} \left( \text{line } (\text{top-left-corner-rectangle} \text{bottom-right-corner-rectangle}) \right) \\
\left( \text{line } (\text{bottom-left-corner-rectangle} \text{top-right-corner-rectangle}) \right)
\]

---

towards meaningful computational descriptions of architectural form
or

\[
[horizontal-axis-rectangle] = [line (get-point-line left-side rectangle (1/2 (distance top-left-corner rectangle bottom-left-corner rectangle))) (get-point-line left-side rectangle (1/2 (distance top-right-corner rectangle bottom-right-corner rectangle)))]
\]

\[
[vertical-axis-rectangle] = [line (get-point-line top-side rectangle (1/2 (distance top-left-corner rectangle top-right-corner rectangle))) (get-point-line bottom-side rectangle (1/2 (distance bottom-left-corner rectangle bottom-right-corner rectangle)))]
\]
\[ \text{longitudinal-axis-rectangle} = \max (\text{length horizontal-axis-rectangle}) \]
\[ \text{transversal-axis-rectangle} = \min (\text{length horizontal-axis-rectangle}) \]
\[ \text{reference-point-rectangle A B} = \]
\[ \text{intersection (line (get-point-line top-side-rectangle (A (top-side-rectangle P1 top-side-rectangle P2))) (get-point-line bottom-side-rectangle (A (bottom-side-rectangle P2 bottom-side-rectangle P1)))) (line (get-point-line left-side-rectangle (B (left-side-rectangle P2 left-side-rectangle P1))) (get-point-line right-side-rectangle (B (right-side-rectangle P1 right-side-rectangle P2)))))} \]
That locates a reference point placed in ratios $A$ and $B$ to the length and the width, respectively, of a given rectangle (Figure A.2.6).

Following this structure it is possible to create, manipulate and transform shapes in many different ways. The essential feature of this approach is that every significant entity can be given a name, and, hence, be manipulated or combined with others to create more complex entities that can also be given names. As we said before, one of the most crucial points in the proposed environment is the capability of combining simple ideas to form more complex ones by using primitive expressions, means of combination and means of abstraction [Abelson and Sussman, 1985]. This same idea is then used to
representations of the same entity can be given different names, manipulated in different ways, and associated to other entities according to the architect's requirements.
Bibliography:


towards meaningful computational descriptions of architectural form

Bazjanac, V. "Architectural design theory: models of the design process", in "Basic questions of design theory", [Spillers 1974] (editor), pg 3-19, 1974


towards meaningful computational descriptions of architectural form


towards meaningful computational descriptions of architectural form

Jurgensen, P. "A method of architectural knowledge representation", in Proceedings of ARECDAO 87, Barcelona, 1987


towards meaningful computational descriptions of architectural form
<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title and Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Salvadori 1974</td>
<td>Salvadori, M. &quot;Is automated architectural design possible?&quot;, in &quot;Basic questions of design theory&quot; [Spillers 1974] (editor), pg 95-102, 1974</td>
<td></td>
</tr>
<tr>
<td>Smith 1978</td>
<td>Smith, B. &quot;How is knowledge representation system like a piano?&quot;, Working Paper 174, Artificial Intelligence Laboratory, MIT, Cambridge, 1978</td>
<td></td>
</tr>
</tbody>
</table>

towards meaningful computational descriptions of architectural form
Vanier, D. and Grabinsky, M. "Integrated computer-aided building design systems", in Proceedings of ARECDAO 87, pg 34-37, Barcelona, 1987


towards meaningful computational descriptions of architectural form
List of Illustrations

page 8: (top) Project for a monument for the International Socialist by Tatlin, Moscow (1920).
(bbottom) Robbie House by Frank L. Wright, Chicago (1906)

page 9: (top) Plotted image of a project by John Burgee and Philip Johnson.
(bbottom) Computer generated two point perspective.


page 11: Wire-frame model on a CAD CRT display.

(bbottom) Computer generated one point perspective.

page 13: (top) Typical CAD display with on-screen menu.
(bbottom) CAD design application: stair design.

page 14: (top) Project for a glass skyscraper by Mies van der Rohe, (1921).
(bottom) Project for a brick country house by Mies van der Rohe, (1923).

page 16: Vuoksenniska Church by Alvar Aalto, (1956).

page 17: (top) Different views (projections) of a same object.
(bbottom) Space typology study by Rob Krier.


towards meaningful computational descriptions of architectural form
list of illustrations

(bottom) Casa Andreis by Paolo Portoghesi and Vittorio Gigliotti, Scandringlia (1965).

(bottom) Galleries Lafayette by Eric Mendelsohn, Berlin (1928).


page 38: Shape grammar; subset of rules and description functions, G. Stiny.

page 39: 'Writing Form' exercise, by Chistina Gryboyanni (1986).

page 58: Barcelona Pavillon by Mies van der Rohe, Barcelona (1929).

page 93: (top) Proportions of columns determined in relation to differing imposed loads.
(bottom) Parametrical study of the facade of San Sebastiano.


towards meaningful computational descriptions of architectural form