Context Modeling

Extending the parametric object model with design context

by
FRED ANDREW MBURU

Bachelor of Architecture
University of Nairobi, 1996

Submitted to the Department of Architecture in partial
fulfillment of the requirements for
the Degree of

MASTER OF SCIENCE IN ARCHITECTURE STUDIES
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUNE 2001

Signature of Author: ____________________

Department of Architecture
May 24, 2001

Certified by: ____________________

William L. Porter
Professor of Architecture
Thesis Supervisor

Certified by: ____________________

Takehiko Nagakura
Associate Professor of Architecture
Thesis Supervisor

Accepted by: ____________________

Roy Strickland
Principal Research Scientist in Architecture

READER

Terry Knight
Associate Professor of Architecture

Context Modeling : Extending the parametric object model with design context

by
Fred A. Mburu

Submitted to the Department of Architecture on May 24, 2001
in partial fulfillment of the requirements for the Degree
of
Master of Science in Architecture Studies

ABSTRACT

Context can be described as the totality of ideas, situations and information that (a) related to, (b) provide the origins for, and (c) influence our response, perspective or judgement of a thing. Design always takes place in a context. However, current Computer Aided Architectural Design (CAAD) systems don't have a way to represent design knowledge associated with context. This thesis presents a computational model, called Context Modeling System (CMS), in which design context is modeled. Using this model, designers can define and prioritize design context. A prototype, based on CMS and rule-based systems in the field of Artificial Intelligence, is also presented.

Thesis Supervisor: Takehiko Nagakura
Title: Associate Professor of Architecture

Thesis Supervisor: William Porter
Title: Professor of Architecture

ACKNOWLEDGEMENTS

I am profoundly grateful to my advisor, Professor William Porter for his guidance, wisdom and insight throughout this thesis semester and my two years here at MIT. I am thankful to Professor Takehiko Nagakura whose guidance and advice was crucial to the completion of this thesis. I am also sincerely thankful to Professor Terry Knight for helping me formulate and shape this thesis.

I would also like to thank Peter Funk and Melrose Ross from Autodesk Inc. for the technical assistance.

To Mum, Dad, Njuguna and Nyoike, thank you for always being there for me. Special thanks to Sam, his wife Lydia and little Allan for their support and encouragement.

I am also grateful to Paul Njoroge and Timothy Murithi for their advice, generosity and assistance. I am also greatly indebted to Josphat Kibogo and his wife, Caroline, for their encouragement, kindness and hospitality while they were at MIT.

Lastly and not least, I would like to thank Miriam Mwangi for her support, assistance and generosity of spirit.

Contents

Chapter 1
Introduction

1.1 BACKGROUND

In the last thirty years, there has been a tremendous growth in the use of Computer Aided Architectural (CAAD) software in the design of buildings. CAAD software is based on the capability of a computer system to process, store and display large amounts of data representing a built environment. Despite the increased use of CAAD in architectural work, its use has not extended over the entire design process.

The design process can be considered to have four stages: Conceptual Design, Design Development, Documentation and Construction. Since most of the current CAAD systems are drawing-oriented, they are heavily used in the later stages of the design process especially in the documentation stage. These systems use geometric entities like lines, solids and surfaces to represent architectonic elements e.g. walls, columns, slabs, etc. Over the years, the concept of parametric objects in CAAD—based on Object Oriented Programming—has become widespread. Parametric objects in CAAD represent architectural objects in real life. These symbols generalize families of objects in the real world and encapsulate the functionality and behaviour of the architectural elements they represent. They make the task of drafting buildings very efficient because they have parameters that can be changed to obtain variations of a particular type. Drafting drawings using parametric objects, becomes a virtual construction process with the placement of windows, walls and doors etc. parallel to the work in the real world construction site.

Although CAAD applications have found utility—as production tools—in the later stages, they have found less acceptance in the conceptual stage where most designing takes place. During the conceptual phase, a designer tries to grasp the design problem by analyzing the functional and contextual factors within which designing will take place. This stage is very important because it determines the overall form of the building. In spite of the major advances in CAAD and computers in general, most architectural designers use computers only for visualization at the conceptual stage of the design process, leaving all the creative and analytical work to the designer.

One of the difficulties for architectural software is the representation of semantics and design emphasis in the digital drawings. There is an underlying assumption that given the structural information only, as contained in a CAAD drawing, all other information such as behaviour, function and context can be derived [Rosenman et al., 1994]. This is very misleading because there are contextual issues, design intentions and subtleties that may not be captured by the composition of drawing entities. There needs to be a shift from the drawing-oriented CAAD systems to integrated environments in which producing blueprints is only one part of the required functionality. CAAD systems should allow the explicit representation of declarative non-algorithmic design knowledge such as contextual issues, as used by designers. As part of the problem just mentioned, current systems are devoid of design knowledge related to the contextual factors considered in the design process. The absence of a way to embed this contextual information in the drawing hinders the designer from engaging the CAAD system during the conceptual part of the design process where design context plays a major role. Given that CAAD systems should have the capacity to support the design process in its entirety, this thesis proposes a model in which design context can be modeled and used to explore design alternatives. Among the problems it addresses is the difficulty of representation and use of context knowledge in the CAAD system. This thesis emphasizes the need to represent design knowledge within the CAAD framework so that intelligent inference making by the computer can be supported.

1.2 THESIS CONTRIBUTION

This thesis presents a computational model—Context Modeling System (CMS)—in which design context is modeled and used to develop a CAAD system in which design decisions can be made based on the designer's intuitions of context relevance. A prototype—ActiveContext—that is based on CMS and rule-based systems is also presented. CMS combines several techniques, which have been developed over a couple of decades in building knowledge based systems (in the field of Artificial Intelligence) to enhance the capabilities of modern CAAD software.

Chapter 2

Design

2.0 WHAT IS KNOWLEDGE?

According to the Oxford English Dictionary, Knowledge is familiarity, awareness or understanding gained through experience or study. Data, information and facts are words that are commonly used in place of knowledge[1].

Knowledge can be classified into three categories [Giarratano, 1998]:

1) Procedural knowledge
2) Declarative Knowledge
3) Tacit knowledge

Procedural Knowledge refers to the comprehension of how to do something. An algorithm that executes a certain task represents the procedural knowledge used to accomplish the task.

---

[1] The study of knowledge is called Epistemology and is the branch of philosophy that studies the nature of knowledge in regard to its presupportions, foundations, extent and validity. Some of the branches of epistemology are shown in Figure 2.1.
Priori knowledge refers to the knowledge that is independent of all particular experiences as opposed to posteriori knowledge, which is derived from experience only. "All cuboids have six sides" is an example of priori knowledge and the truth of that statement is universally true and cannot be denied without contradiction.
On the other hand the statement "Dining Rooms are next to the kitchen" is an example of posteriori knowledge. The truth of this statement can only be determined by an investigation. One may understand the sense and importance of the two rooms being together but to verify the statement, some kind of empirical investigation needs to be carried out.

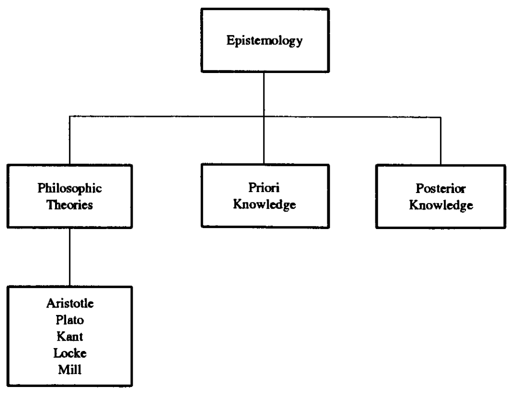


Fig. 2.1 Branches of Epistemology

Declarative Knowledge refers to the knowledge about the truth or falsehood of something. This kind of knowledge is manifested in statements like “Don’t place low windows in bathroom or toilet.” In this statement there exists a fact known to be true about a bathroom that makes placing a low windows in them a bad idea.

Tacit knowledge, also called the unconscious knowledge, cannot be expressed by language [Giarratano, 1998]. The design process involves a lot of tacit knowledge. If an architect was asked, “How do you design?” He could say that he gets inspired by some magic force and then sketches to develop ideas. However, if a lower level question was asked e.g. “How do you manipulate your brain cells to get inspired?” the architect is unlikely to give a reasonable answer even though he actually does it.

Knowledge complexity can be represented in a hierarchy structure as shown below. At the bottom level, there is noise, which refers to information that is of no interest to the subject under scrutiny. Above noise is data, which represents facts that are relevant to the subject matter. It is important to note that something could be data for one subject and noise in another. The relevance of the data to the subject matter determines its importance. On the next level of this hierarchy is information, which is processed data. Specialized information, referred to as Knowledge is the layer above information. Metaknowledge, the knowledge about knowledge, sits on top of the hierarchy and is the highest level of knowledge.

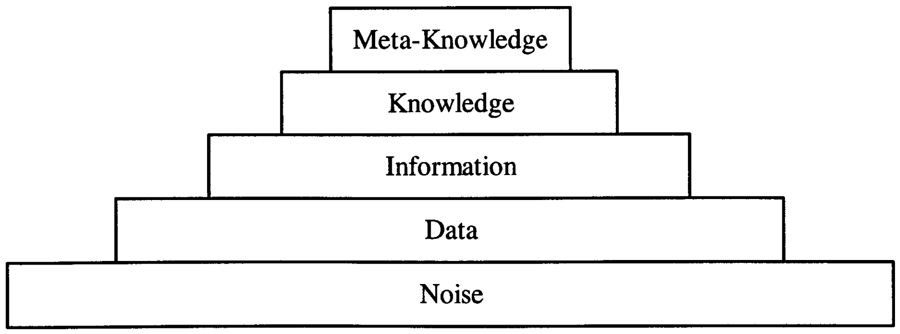


Fig. 2.2 Knowledge Hierarchy

2.1 DATA, INFORMATION AND KNOWLEDGE

The following series of ASCII characters may appear to be noise without any additional knowledge attached to it.

```
((-1 . <Entity name: 4007ddb0>) (0 . "LINE") (330 . <Entity name: 4007dcf8>) (5 . "A1") (100 . "AcDbEntity")
(67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 0.0 0.0 0.0) (11 1000.0 0.0 0.0) (210 0.0 0.0 1.0)) ((-
1 . <Entity name: 4007dda8>) (0 . "LINE") (330 . <Entity name: 4007dcf8>) (5 . "A2") (100 . "AcDbEntity")
(67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 1000.0 0.0 0.0) (11 1000.0 1000.0 0.0) (210 0.0 0.0
1.0)) ((-1 . <Entity name: 4007dda8>) (0 . "LINE") (330 . <Entity name: 4007dcf8>) (5 . "A3") (100 .
"AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 1000.0 1000.0 0.0) (11 0.0 1000.0
0.0) (210 0.0 0.0 1.0)) ((-1 . <Entity name: 4007dda8>) (0 . "LINE") (330 . <Entity name: 4007dcf8>) (5 .
"A4") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 0.0 1000.0 0.0) (11 0.0
0.0 0.0) (210 0.0 0.0 1.0)) ((-1 . <Entity name: 4007dda8>) (0 . "LINE") (330 . <Entity name: 4007dcf8>) (5
. "B1") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 1000.0 500.0 0.0) (11
1000.0 2500.0 0.0) (210 0.0 0.0 1.0)) ((-1 . <Entity name: 4007dda8>) (0 . "LINE") (330 . <Entity name:
4007dcf8>) (5 . "B2") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 1000.0
2500.0 0.0) (11 2000.0 2500.0 0.0) (210 0.0 0.0 1.0)) ((-1 . <Entity name: 4007dda8>) (0 . "LINE") (330 .
<Entity name: 4007dcf8>) (5 . "B3") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine")
(10 2000.0 2500.0 0.0) (11 1000.0 2500.0 0.0) (210 0.0 0.0 1.0)) ((-1 . <Entity name: 4007dda8>) (0 .
"LINE") (330 . <Entity name: 4007dcf8>) (5 . "B4") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0")
(100 . "AcDbLine") (10 1000.0 2500.0 0.0) (11 1000.0 500.0 0.0) (210 0.0 0.0 1.0))
```

Fig. 2.3 ASCII data

However, when this data is read by a CAAD program (capable of reading the DWG format), it translates into eight lines with the following coordinates.

```
Line A1 : start point (0,0) , end point (1000,0)
Line A2 : start point(1000,0) , end point(1000,1000)
Line A3 : start point(1000,1000) , end point (0,1000)
Line A4 : start point(0,1000) , end point (0,0)
Line B1 : start point(1000,500) , end point(1000,2500)
Line B2 : start point(1000,2500), end point (2000,2500)
Line B3 : start point(2000,2500), end point(1000,2500)
Line B4 : start point(1000,2500) , end point (1000,500)
```

An algorithm in the CAAD program transforms the above data into information regarding lines. Figure 2.4 shows the image that is displayed on the computer screen.

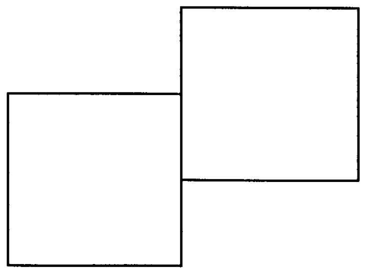


Fig. 2.4 Two squares image

The eight lines—A1 to B4—describe two squares. A viewer looking at the data in the digital drawing sees squares and not the 8 individual lines or the ASCII Data. Using the lines' information, it is possible to create new knowledge about the two squares (sqrA and sqrB). For instance, if the rule shown below was applied to the two squares, new knowledge about the relationship between two squares is generated.

```
If   Square (sqrA) and Square (sqrB) have a shared line
THEN
     (assertfact  (LinkedSquares sqrA sqrB) )
```

The knowledge stored as a fact object states that square1 and square2 (LinkedSquares sqrA sqrB) are linked. This fact is a knowledge representation of the adjacency relationship between sqrA and sqrB.

In the next section, some concepts in programming that are important in the representation of data and knowledge will be discussed.

2.2 DATA ABSTRACTION

Computer programs consist of code and data. They can be organized around either code or data. Two paradigms determine the way computer programs are written. The first paradigm is the procedural programming model in which programs are written as a series of operations. In this model, programs are written around what is happening and code acts on data. The second paradigm, is the object-oriented programming model in which programs are organized around data. In the object-oriented model, data controls access to code. The prototype developed in this thesis follows the object-oriented paradigm.

A key aspect to object-oriented programming is the concept of data abstraction. Human beings manage complexity through abstraction. They see a car, not as an assembly of brakes, engine and transmission, but as a well-defined object with its own unique behavior. Similarly, data abstraction allows programmers to abstract from the details of how data objects are implemented to how the objects behave [Liskov, 2000]. Through data abstraction, software programmers are able to extend a programming language by defining new data types.

One powerful way to manage abstraction is through hierarchical classification in which objects of semantic similarity are grouped together. A *type hierarchy* is used to define a family of a certain type. At the top of the hierarchy is a type whose behavior is common to all those that are derived from it. A type hierarchy corresponds to real world families like mammals and birds in the animal kingdom. Using hierarchical classifications, a complex object or system can be broken down into smaller parts that are easier to manage.

There are three concepts that embody the idea of object-oriented programming. They are encapsulation, polymorphism and inheritance. Encapsulation is the mechanism in which code and data related to a type of object are grouped together and protected from outside interference [Rambaugh, 1991]. Polymorphism means the ability to take many forms. In object-oriented programming, this refers to the ability of an entity to refer at run time to instances of various types [Meyer, 1988]. Inheritance is a mechanism in which an object type acquires the properties of another type. This promotes the idea of hierarchical classifications. Related objects are grouped together to form a type hierarchy. When an object B inherits properties from another object A, B is referred to as a subtype of A. By using the inheritance mechanism, subtypes need only define those characteristics that are unique to them. There is a wealth of literature about these concepts of object-oriented programming and further details can be obtained from sources like [Liskov, 2000], [Meyer, 1988] and [Rambaugh, 1991].

2.3 KNOWLEDGE REPRESENTATION

The field of Artificial Intelligence had grampled with the issue of knowledge represention for decades. The paper—What is a knowledge Representation? [Davis et al., 1993]—asserts that a knowledge representation is best understood in terms of the five distinct roles it plays. These are:

1) A surrogate
2) A set of ontological commitments
3) A fragment of a theory of intelligent reasoning
4) A medium for pragmatically efficient computation
5) A medium of expression and communication

First, a knowledge representation is a surrogate or substitute for something that exists in the real world. Operations on the representation replace direct manipulation of the corresponding real object. Using knowledge representations, abstract notions like space adjacency conditions can be represented just as well as doors and windows. Unfortunately, it is impossible to create surrogates that are prefect copies of objects in the real world. The complexity of the real world can be overwhelming and therefore a knowledge representation also describes a set of ontological commitments that focus on a certain aspect of the real world object.

A knowledge representation is also a computational medium within which intelligent reasoning can be accomplished. Therefore, it ought to be efficient so that decisions are made within reasonable time. In the last role, a knowledge representation is a medium of expression in which human beings can say things about the world [Davis et al., 1993].

2.3.1 Knowledge Representation Technologies

This section gives a brief overview of the most common knowledge representation technologies i.e Frames, Logic Programming, Semantic Networks and Rule-based systems.

Semantic Networks

Semantic networks consist of nodes and links. Nodes represent objects, concepts and events while links represent relationships that are relevant to connecting nodes. Using node-link structures, hierarchies are constructed to which deductions can be applied. Two types of link that are commonly used are IS-A (ISA) and A-KIND-OF (AKO) links. An ISA link refers to an instance of a type while AKO is used to connect a subtype to a supertype in the semantic network. A subtype inherits its characteristics from the supertype and adds only properties that differentiate it from its parent.

Logic programming

The most renown programming language that uses logic programming is Prolog. It uses Horn clauses of the form A1 & A2 & A3 & .... Az = > B to represent knowledge. A1 to Az represent conditions that must all be satisfied before and the predicate expression B can be said to be true [Giarratano, 1998].

Frames

Frames are data structures that represent stereotypical objects, activities or events.

They are made up of slots and have a structure analogous to a record structure in Pascal. Slots are filled with default values—referred to as fillers —which can be changed to create specific knowledge. The utility of frames lies in the hierarchical classifications and inheritance. Generic frames are placed on top of type hierarchies from which other objects inherit properties. By using inheritance as well as other frames as fillers, very powerful knowledge representations can be made.

Rule Based Systems

In rule-based systems, knowledge is represented in the form of rules and facts. Rules are the set of transformations that may be applied to a knowledge representations scheme. They can be broken down into two parts. The first—the antecedent—is the contextual part that expresses when a rule can be used and the second part—the consequent—is the transformational part that shows what changes take place in the knowledge base when the rule is applied. In the example shown below, statement 2 of the rule example is the antecedent of the rule and addDoorToThisWall is a consequent.

Fact Examples

```
(LinkedSpaces (Space A ) (Space B) (LinkedBy C))
(Wall    (StartPoint p) (EndPoint q) (Color D))
```

Rule Example

```
1(defrule  ADD_DOOR
2   (LinkedSpaces (Space ?one )(Space ?two)(LinkedBy?wall)
3   =>
4   (addDoorToThisWall ?wall)
5)
```

A rule-based system is made up of the following parts: user interface, explanation facility, working memory (where facts are stored), inference engine, agenda (list of rules created by the inference engine) and a knowledge acquisition facility [Giarratano, 1998]. In order to make inferences, the inference engine follows a cycle of these steps: First, it examines the conditional part of each rule to find any rules for which all the antecedents are satisfied. If no rule has all its antecedents satisfied, the cycle is terminated. Otherwise, it selects one of the rules (whose conditional part was satisfied) and performs the actions associated with the consequent part of the rule. If more than one rule had all the antecedents satisfied, it executes the one with the highest priority. After execution it goes back to the matching (first) step [Giarratano, 1998].

2.4 KNOWLEDGE BASED DESIGN SYSTEMS

In order to address the difficulty of representation and extraction of semantics from the digital drawings, systems—called Knowledge based design systems—that store design knowledge have been proposed. By definition, Knowledge based design systems are systems that use a knowledge base to perform—or assist in performing—a design task computationally. These systems allow the explicit representation of declarative, non-algorithmic design knowledge as used by architects, and provide various methods of symbolic reasoning [McCullough, 1991]. They can be viewed as decision support systems enabling designers to explore the structure of design problems and their solutions by combining human design expertise with domain and design knowledge represented in knowledge based technologies. There is a wealth of literature about these kinds of applications and further details can be obtained from sources like [McCullough, 1991], [Coyne 1990] and [Carrara and Kalay, 1994].

2.5 DESIGNING

The design activity is a creative process. It involves associating and arranging forms into new meanings [Burden, 1934]. Designing is a search—involving the invention and disposition of forms, elements and materials—for a solution that satisfies a myriad of functional, contextual and economic factors or constraints. Looking at design as a search or exploration, implies that, at the beginning, some goals are set and the search continues until a solution is found if any exists.

The design process involves a series of decisions taken one at a time. Decisions narrow choices, until there is one satisfactory, sometimes optimum, solution. Designers solve design problems by breaking them down into small sub-problems which when resolved contribute to overall solution. Simon Herbert sets forth the following view in his 1960s text *The Sciences of the Artificial* [Simon, 1969]: The basic idea is that several components in any complex systems will perform sub-functions that contribute to the overall function .... To design such a complex structure, one powerful technique is to discover viable ways of decomposing the complex structure into semi-independent components corresponding to its many functional parts. The design of each component can then be carried out with some degree of independence of the design of others, since each will affect the others largely through its function and independently of the details on the mechanisms that accomplish the function.

He suggests that one way of considering the decomposition, but acknowledging that the interrelations among cannot be ignored completely, is to think of the design process as involving, first, the generation of alternatives and, then, the testing of these alternatives against a whole array of requirements and constraints [Simon, 1969]. In this framework, design is guided by the testing process which checks whether a design solution satisfies all the constraints set at the beginning.

2.5.1 Design Context

As mentioned earlier, the architectural design has to take into account a multitude of factors namely functional, contextual or financial. In this section, contextual design factors are discussed.
In the Dictionary of Philosophy [Angeles, 1981], context is defined as follows:

> ***Context*** *(L. contexere, "to weave together." from con, "with," and texere, "to weave"): The sum total of meanings (associations, ideas, assumptions, preconceptions, etc.) that (a) are intimately related to a thing, (b) provide the origins for, and (c) influence our attitudes, perspectives, judgments, and knowledge of that thing.*

This definition captures the role of contextual factors in a design problem. In a good design, context plays a major role in "weaving" a building to its site. When a building is designed in context with its surrounding, there is a sense of harmony between the building and the site. The designer weaves the design solution according to the existing fabric of site conditions, problems and opportunities. This weaving activity involves three actors in the "consequence triangle". These are the users, context and the building [White, 1983].

Contextual factors can be classified into three broad categories:
1) Physical contexts
2) Social contexts
3) Cultural/Historical contexts.

Social and Cultural/historical contexts are related to the users of the buildings or the general human community around the site. Historical contexts can also be related to the architecture of the site's surroundings. Physical context pertain to site conditions.

Usually before any designing take place, there is a pre-design site investigation called context analysis—or site analysis—which focuses on existing and potentially significant conditions on or around the site. The purpose of this analysis is to enlighten the designer on the conditions of the site before the design process commences. This helps the designer incorporate meaningful responses to the site conditions. Data collected from contextual analysis can be either hard data or soft data [White, 1983]. Site conditions are hard data and they do not require any human judgement. Examples of physical context are location, orientation, and climate. Social contexts, on the other hand, are soft data and require some value judgement where evaluating a site. Examples of soft data are good and bad views, best site approach, focal points on or around the site.

2.5.2 Form Determinants

Context and function are major contributors to the form of a building. Contextual factors affect the disposition and form of a space according its relationship with the external conditions. Functions, on the other hand, influence the internal spatial relationship [White, 1983]. During the design process, these two factors push and pull the form of the building until a balance between the two is found.

To demonstrate how context may affects form, consider how the North-South orientation of a site affects the placement of openings in a building. If the site is in Nairobi, Kenya, to achieve optimal thermal and day lighting conditions, openings are placed on the northern side of a building since Nairobi is in the southern hemisphere. The next best orientation to place openings is south followed by east. Because of the hot afternoon sun, placing windows on the western walls results in unbearable hot conditions. On the contrary, in Boston, optimal day lighting and thermal conditions for spaces comes from the southern walls.

2.6 CONTEXT MODELING SYSTEM

The fundamental task of a Context Modeling System (CMS) is to provide the necessary framework to automate decision-making, based on context. As discussed in the last section, context is one of the most influential factors in design conceptualization. Most CAAD applications are fundamentally concerned with the graphical representation of architectural components—walls, doors, floors—in a drawing. These CAAD programs rarely attach to the drawing, any design knowledge that designers can use to relate the graphical representations to the contextual issues addressed in the design process.

A context modeling system, on the other hand, aims to provide the designer with a way to represent and store context-related design knowledge within a drawing. This knowledge can be used for analysis of design solutions or for synthesis by making decisions on behalf of the designer.

In order to examine how context can be modeled, this thesis explores the design of a context modeling system in a simplified design world in which a design variable is manipulated by varying the site context. A prototype called ActiveContext has been developed which supports context modeling within a CAAD program. In this prototype, the design variable is placement of windows on a building shell. The context contains three elements :

1) North-South orientation of the site
2) Views
3) Focal points

For simplicity, it is assumed that these are the only factors that affect designing. The term design context shall be used to mean any one of contextual factors, and design contexts to mean one or more of them. The three design contexts act independently of each other, though the placement of windows (the design variable) is determined by evaluating all three.

In this simplified model, the user begins by setting the design contexts on the site drawing. The user supplies an outline of the building shell onto which openings i.e. doors and windows are to be added. After the user has ranked each design context according to his/her interpretation of each context's importance, the system is ready to automate the task of placing openings. In the first run, and in any subsequent runs, one opening is added for each room. This opening is placed on the wall selected by the highest ranking context affecting that space. In the second run, another window is added based on the second highest-ranking context and so on.

2.6.1 System Requirements

In order to support design context, a context modeling system has to accommodate several key requirements. First, it must allow a designer to specify multiple design contexts. A site may have good views, bad views or both. Another site may have the sun's orientation as the only significant design context. It is up to the designers to identify and specify all the relevant design contexts. The system must also provide a way to explicitly represent each design context. Symbols, called context icons, are introduced in ActiveContext. It is highly unlikely that a CAAD system will have in its library, objects that represent context and therefore, the developer of the CMS has to provide these icons. Nonetheless, most CAAD systems will have a rich library of symbols that can be used by the context modeling system to automate task execution. This thesis views context modeling as an extension of the parametric object models used in CAAD systems.

A second requirement, for the system, is the ability to compute which objects are affected by a design context. Context icons exert their influence within a certain range or on particular types of objects. It is therefore important to have a mechanism through which a designer can define the scope of each design context. The process of selecting objects that are affected by a design context is referred to as scoping.

The third requirement is that there must be a way to specify how important each design context is. Contextual issues differ from site to site. Moreover, the relevance of a contextual issue in a site will differ from designer to designer and therefore a context modeling system must allow users to specify their personal interpretations of relevance.

The fourth requirement on the context modeling system, is that it must support a decision-making mechanism. To some extent, the context modeling system mimics the decision-making capability of a human designer. To avoid being overwhelmed by all the contextual factors that affect designing, ActiveContext makes decisions only on account of a limited number of design contexts defined in the simplified world.

The system must be extensible. This fifth requirement is necessary because it is highly likely that the system will need to grow. At any stage of the system's life , it must be possible to add new contexts without redesigning the entire system.

Finally, the context modeling system must provide a mechanism to subdivide a design space into smaller parts. By dividing the design space, the designer can respond to contextual issues differently in each of the parts.

In summary, the context modeling system must provide at minimum the following functionalities:

1) Context representation and Scoping
2) Ranking of importance
3) Decision making
4) Extensibility
5) Partitioning

2.6.2 Context Representation and Scoping

Design contexts are represented in a drawing by context icons. Each of the three design contexts supported by the ActiveContext application has a unique icon to distinguish it. The NorthScope icon is a directional indicator. Its orientation matches the North-South axis on a site. GoodView and UnsightlyView icons represent areas on a site that have good and bad views respectively. FocalPoint icons represent important points in a site. These icons enable users to locate design contexts on a site. They also allow users to state the significance of the design context in a design problem.

Context icons support the following functionalities :
1) They provide a graphical representation for design context
2) They compute and track objects that they affect
3) They allow users to set the level of importance of a design context
4) Assert facts in the inference engine linking objects with the design contexts that affect them

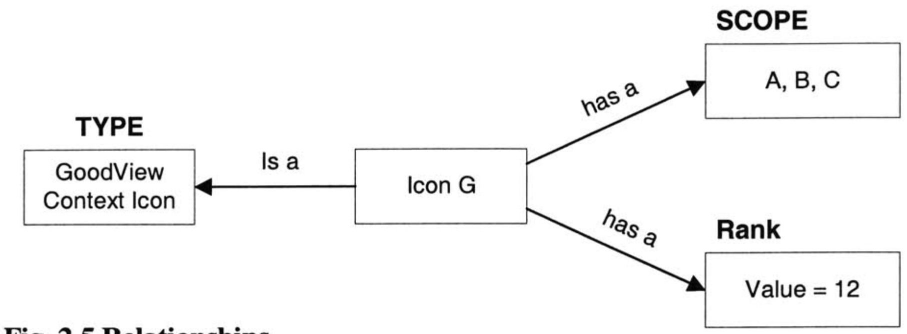


Fig. 2.5 Relationships

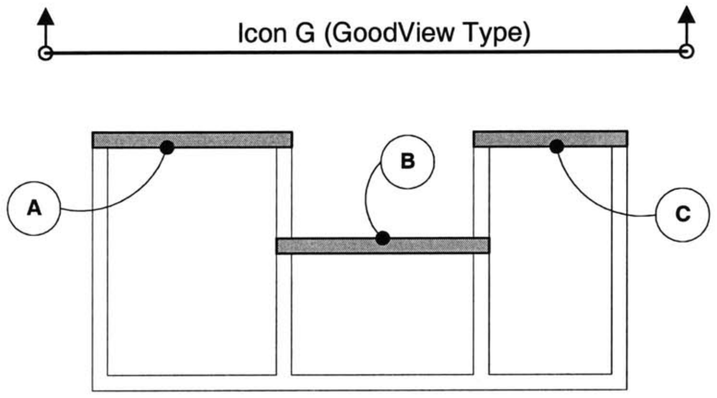


Fig. 2.6 Scope and Scope members

Each context icon has a scope. In Figure 2.5 above, Icon G has three members (Walls A, B, C) in its scope. Scopes contain all the drawing elements that are affected by the design context an icon represents. In the example above, all the walls facing the direction of the good view (shown in Fig. 2.6) by the arrows of Icon G) are added to the scope of Icon G. ActiveContext uses algorithms defined within each context icon to select objects that are affected by a design context. The GoodView context icon is a subtype of LineScope type. All LineScope objects (which include all GoodView objects) select the closest non-perpendicular objects in a drawing into their scope. Other general classes from which icons are derived are PointScope and AreaScope. A more complete discussion on types of context icons can be found in the next chapter.

During the decision-making stage, the context icon asserts facts, into the inference engine, associating each scope member to the design context it represents. A crucial part of the information sent to the inference engine are facts about how important the designer ranks each design context.

2.6.3 Context Ranking

In order to model the manner in which designers make decisions, given a set of design contexts, a ranking system has to be adapted. The significance of contextual issues, like views, varies from site to site. In some design problems, views are very crucial while in others views do not influence the design at all. In addition, the importance of one contextual issue in a single site can vary from one designer to another. It must therefore be possible to specify the significance of a design context in a design problem. One way to express these varying levels of importance is by using a range of integer values, to rank design contexts. If all design contexts are assessed along the same range of values, any two design contexts can be compared to find out which one the designer assigns a higher degree of importance. Figure 2.7 below shows an example of an interface where ranks of different context icons can be assigned and manipulated.

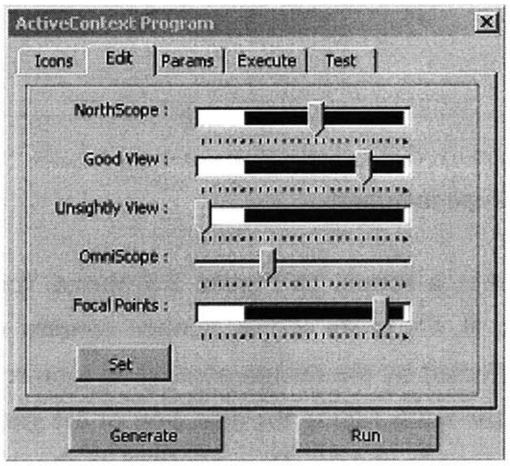


Fig. 2.7 Interface for setting rank values.

There are two types of ranking schemes:
1) Full ordering
2) Partial ordering

In full ordering, design contexts are assigned an absolute value. These values range from a predefined minimum to a predefined maximum (-5 to 20 for ActiveContext). When an icon A has a rank value higher than icon B, it implies that A's design context is more important than B's. In addition, if it is stated that A = 3 and B = 7, and if C is independently stated to have a value of 5, then it means that B is greater than C and C is greater than A.

In a partial ordering, if A = 2 and B = 4 is stated independently of C = 5 and D = 1, it does not follow that C is greater than B and A greater than D. In a partial ordering, relationships cannot be deduced by a simple comparison of independent premises.

When placing a window within a space, ActiveContext first checks the rank values of all contexts affecting that space and then places an opening on the wall selected by the highest ranking context. If contexts A, B, C, D have ranks –1, 4, 7, 3 respectively. The opening is placed on wall selected by context C since it has the highest ranking.

It is also important to define values that declare absolute certainty. Many times, it is important to say, “DO NOT put a wall on Wall A !” or “Wall A MUST have a wall!” These two conditions imply that no matter what ratings the other walls have, an opening must (or must not) be placed on Wall A. In ActiveContext, two values, the minimum and maximum numbers of the rank range, are reserved for declaration of absolute certainty. The maximum range number is referred to as the Affirmation Value while the minimum range number is called a Negation Value.When a context icon is assigned the Affirmation Value, the system is obligated to place an opening to the wall selected by that icon. On the other hand, if a context icon assigned a Negation Value, no opening can be added to any wall in the scope of that icon.

ActiveContext employs a full ordering scheme to compare different design contexts. All context icons have –5 as a minimum and 20 as a maximum. A graphical interface is used to assign values for each design context and since all context icons have the same range, by simple inspection, a user is able to determine the relative importance of design contexts.

2.6.4 Conflict Resolution

Designers frequently find themselves in situations where two or more contextual issues require conflicting responses. Fig. 2.8 shows a situation where a decision has to be made regarding placing an opening on the grayed wall. In this situation, one has to consider the effects of the North-South orientation of the site and the good view that extends the entire length of the wall. Designers consciously or unconsciously resolve such problems with their intuitions of relevance. However, since a computer does not have that kind of intelligence, a system to resolve such conflicts must be devised.

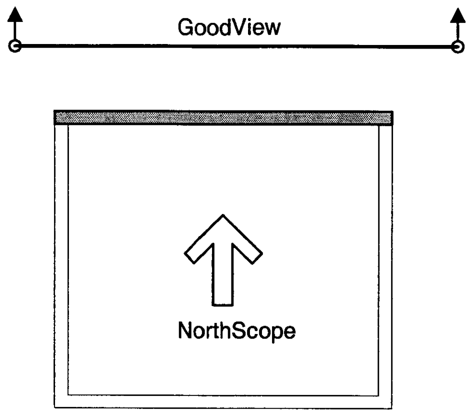


Fig. 2.8 A wall which is a scope member of two context icons.

Situation	NorthScope Rank	GoodView Rank
A	7	15
B	15	7
C	-1	10
D	-5(min)	12
E	-5(min)	20(max)

Fig. 2.9 Rank values

When two context icons exert their influence on one object (e.g placing an opening on the grayed wall in Fig. 2.8), one of four things may happen:

1) **Strengthening:** In this situation, one context icon reinforces another. This can result in an increment of the rank value or retention of the higher rank value. In situation A and B (Fig. 2.9), the NorthScope and the GoodView icons both support the placement on an opening on the wall. The rank of the wall can either be increased to 22 or left at the highest value of 15 depending on the implementation of the context modeler.
2) **Weakening:** In this situation, one context icon contradicts another and this results in the overall rank value of the object decreasing or retaining the higher rank value. Situation C (Fig. 2.9) illustrates this point. The wall can have a rank value of 9 (= 10 - 1) or can keep the higher rank of 10.
3) **Elimination:** Here, one context icon asserts a rank but another context icon has a Negation rank value thus canceling the effect of the other. Situation D is an example of elimination. The GoodView icon proposes that an opening be placed on the wall (rank value = 12) but the NorthScope prohibits this because of its Negation value (-5).
4) **Contradiction:** This is an undesirable situation where both the Affirmation Value and Negation value are assigned to one object. This may happen (Situation E) when the GoodView icon demands that an opening be placed (rank = Affirmation Value) while the NorthScope prohibits (rank = Negation Value) the placing of any opening. The system warns the user of this kind of contradiction.

2.6.5 Partitioning

Contextual factors considered important in one area of a project may not be as important to others. In a large project like a university, the design of a dormitory may require a different set of contextual considerations from a classroom or laboratory. In such a design problem, a single scheme for assigning ranks to design context cannot work well. In addition, as the architectural design process becomes more and more of a collaborative effort, it is important to accommodate each designer's interpretation of context within the drawing. One way to accomodate multiple schemes of context is through partitioning.

Partitioning is the subdivision of the design space into smaller units within which design context can be assessed autonomously. This allows the context modeling system to scale as the design problem gets larger. Partitioning enables the assignment of more than one value to a design context within a drawing. Using symbols called partition icons, a designer can define an area or volume where contextual issues is assessed independently. In the ActiveContext application, the AreaScope icon is a partition icon and it allows users to define a rectangular area within which design contexts can be ranked independently. Each partition icon has a collection of rank values assigned to each design context affecting the area. This collection is referred to as a context set. A partitioning example is discussed in the section 3.2.4.

Chapter 3

Examples

3.1 BACKGROUND

ActiveContext is a plug-in extension to Architectural Desktop Application. This section describes how users interacts with ActiveContext to model design context. Examples taken from a session using the system are also illustrated to demonstrate its functionality. As was mentioned in the last chapter, ActiveContext places openings on a "sketch" plan based on three design contexts:

1) North-South orientation of the site
2) Views
3) Focal points

ActiveContext also supports partitioning using a rectangular partition icon referred to as AreaScope.

3.2 USER INTERACTION

ActiveContext has one dialog box with multiple pages through which the users interact with the program. The first page contains a set of buttons used to create context icons (Fig. 3.2) and the second is a series of sliders used to change the rank values of the icons (Fig. 3.3). The following is a series of steps required to place openings in a plan using ActiveContext:

1) Loading or creation of a "sketch" drawing
2) Creation of walls
3) Placement of context icons
4) Ranking
5) Execution
6) Partitioning

3.2.1 Loading

Users start by loading or drawing a block model of rectangular spaces. Figure 3.1 shows an example of a block model. This block model is used to generate a graph that contains spatial relationships of the plan. The graph has rooms as nodes and room neighbors as edges.

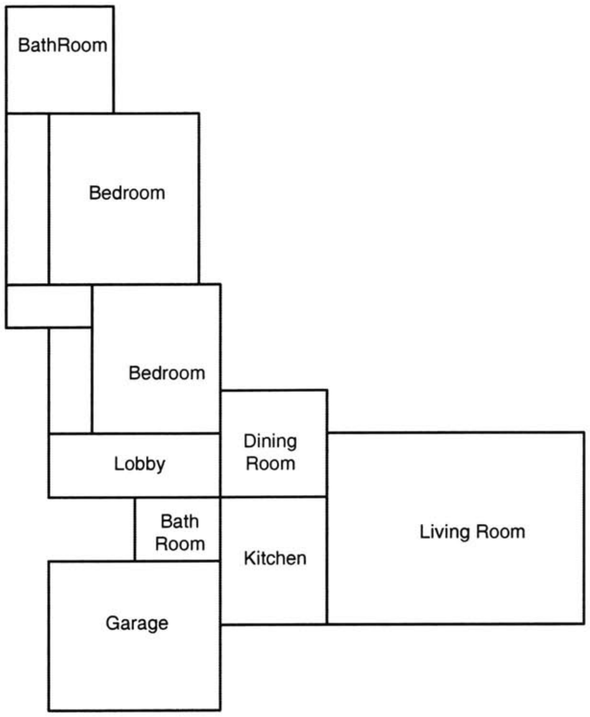
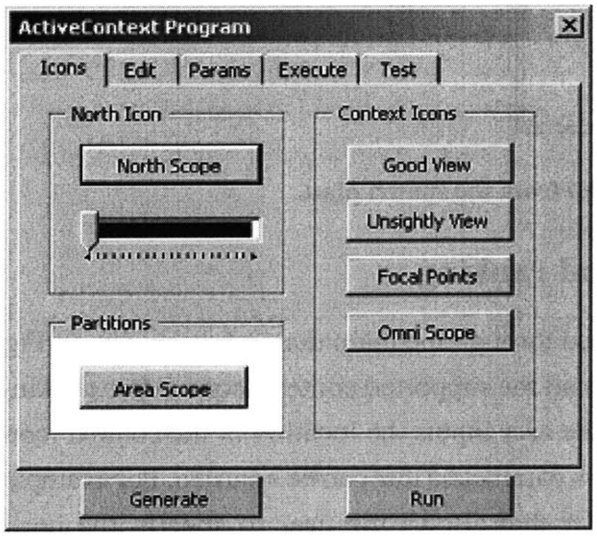


Fig. 3.1 Step One : Sketch Plan.



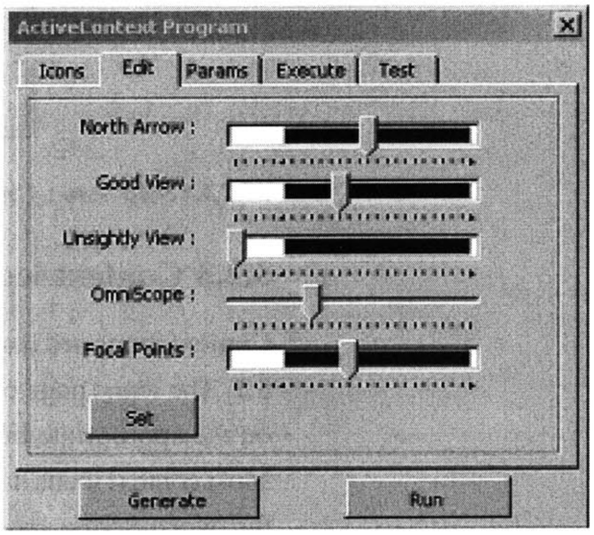


Fig. 3.2 Icons page

Fig. 3.3 Ranking page

3.2.2 Wall Creation

After loading or creating the sketch plan, the user creates walls by clicking on the generate button (Fig. 3.2) on the main dialog box. ActiveContext goes through all the spaces and creates the enclosing walls. This process not only reduces the amount of work required to create the building shell but also, asserts in the knowledge base facts that map walls to spaces. Figure 3.4 shows the building shell created from the drawing in Figure 3.1.

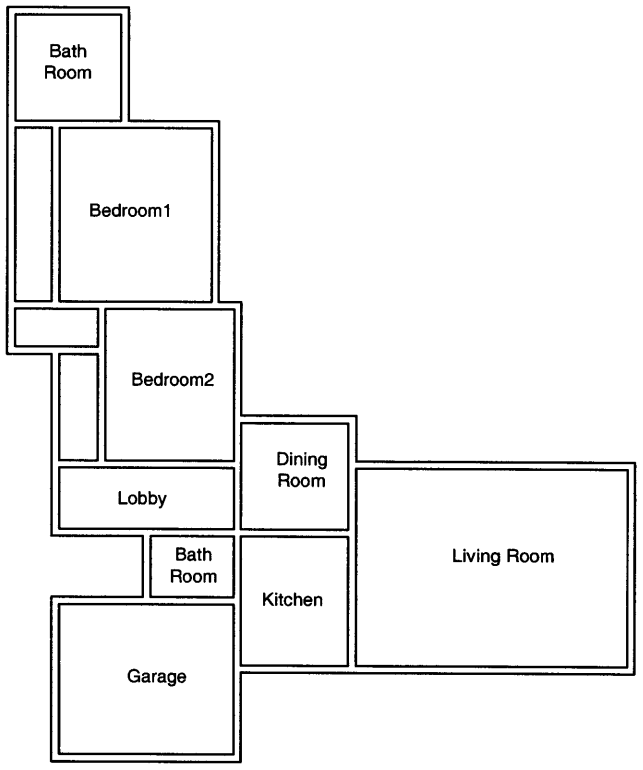


Fig. 3.4 Step Two : Create wall from the sketch plan.

3.2.3 Context icons and ranking

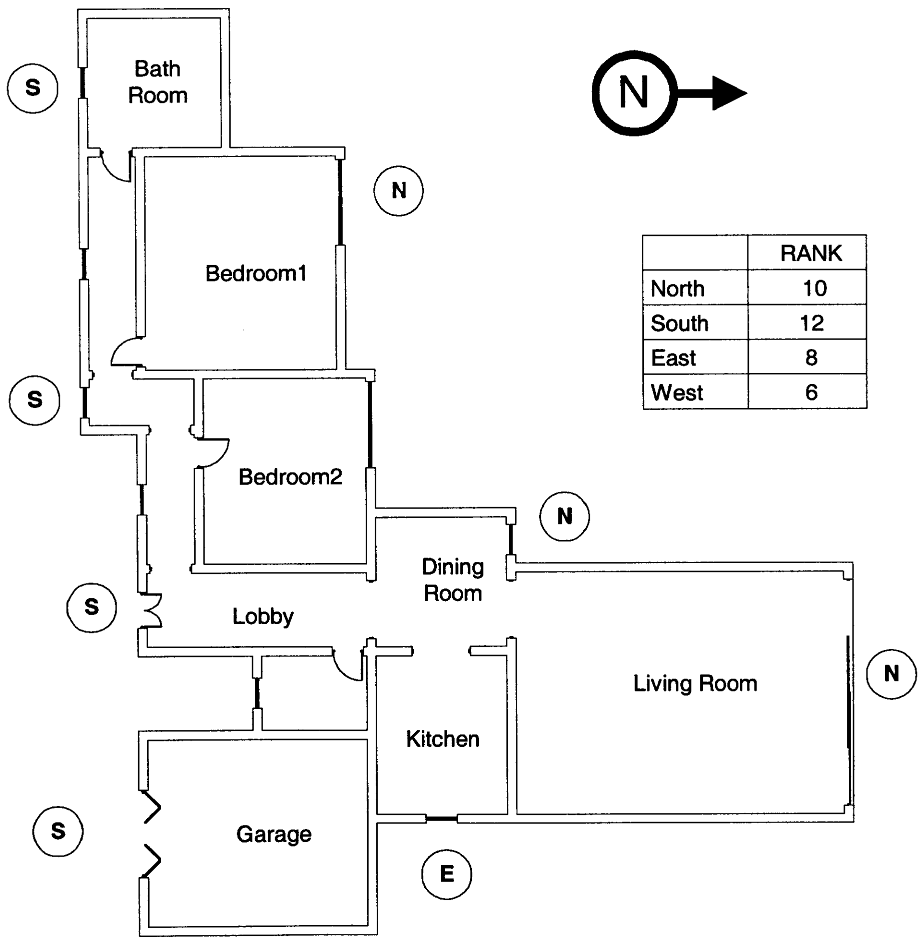
Context icons are instantiated through the main dialog box shown on Fig. 3.2. The icons page contains all the supported context icons. After clicking on the appropriate button, the user inputs the location of the context icon. Most context icons have extra parameters that can be adjusted. For example, the NorthScope has a slider that allows the user to specify the north direction. The AreaScope, GoodView and UnsightlyView scope have size parameters that users can adjust. After positioning the context icon, the user ranks the context icons according to the importance they attach to their respective design context. Fig. 3.3 shows the ranking interface.

3.2.4 Execution

After the user has located all context icons and assigned ranks, ActiveContext is ready for execution. This section demonstrates some drawing examples generated from running the ActiveContext application.

Example 1 :

The first example illustrates a scenario in which the only context icon defined is the NorthScope. Figure 3.5 shows the rank values of each context and the resulting plan. ActiveContext aims to put at least one window in every room. In this example, the NorthScope has 12, 10, 8, 6 as ranks for the south, north, east, and west orientations respectively. In bedroom1, the dining room and the kitchen, openings are inserted on the North-facing wall—the second highest ranked orientation—because these rooms do not have exterior south-facing walls.



	RANK
North	10
South	12
East	8
West	6

Fig. 3.5 Step three : Place context icons.

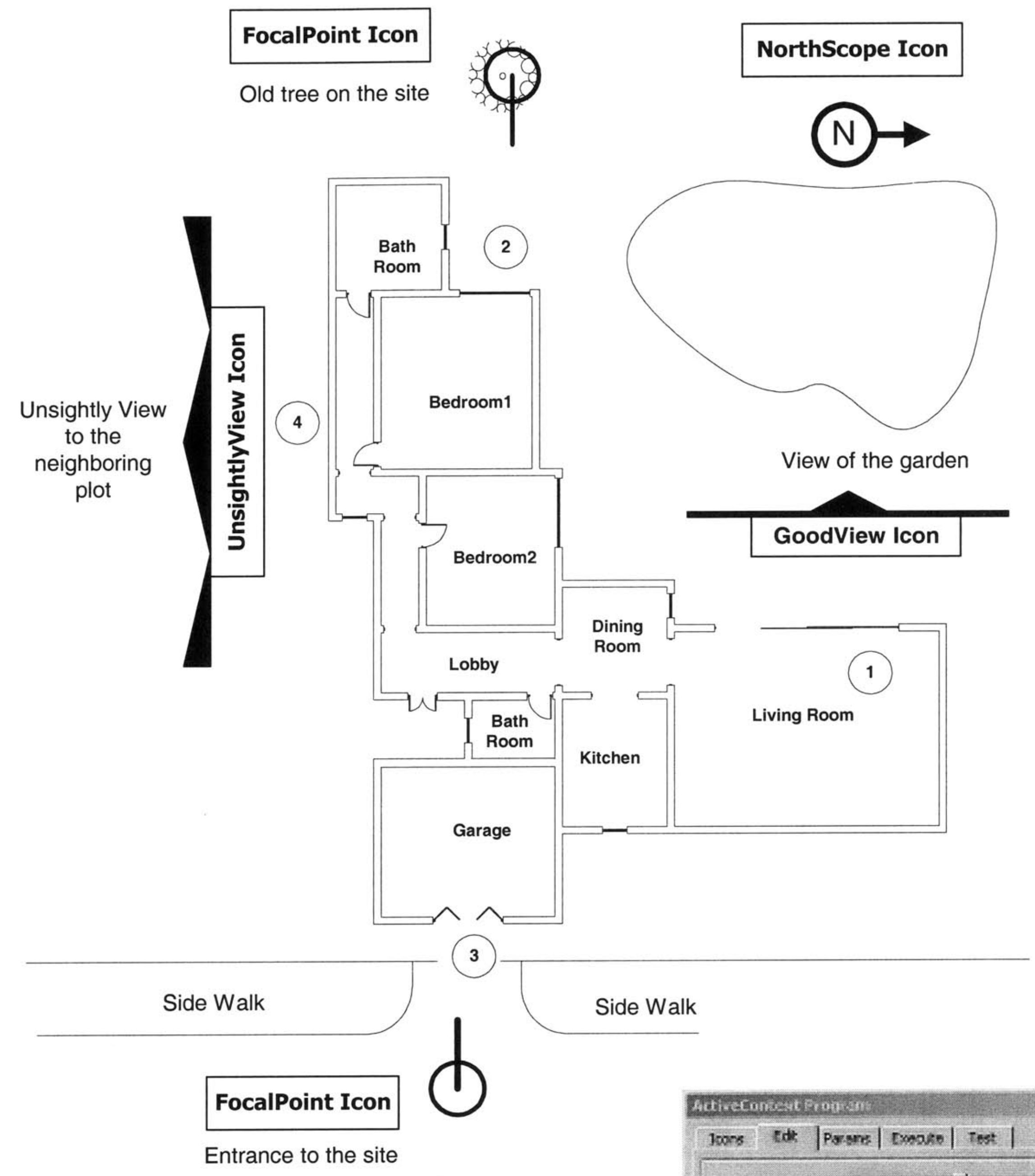


Fig. 3.6 Example 2 : Icons and Ranking

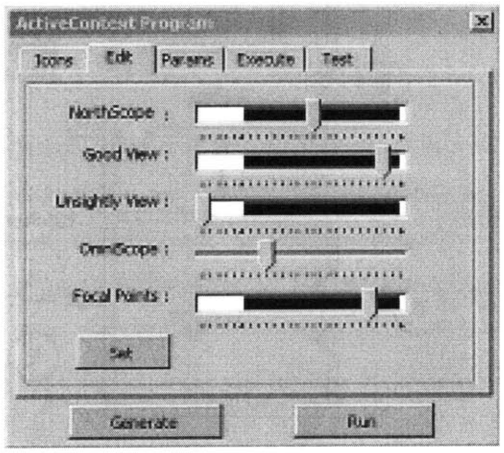


Fig. 3.7 Ranks for Example2

Example 2:

In example 2, context icons representing different kinds of contextual issues are added to the drawing in example1. By inspecting the Ranking page (Fig. 3.7), it can be seen that the user lays more emphasis on good views and focal points than North-South orientation. The NorthScope points in the same direction as in the previous example. A FocalPoint icon is added at the location of the old tree near Bedroom1. Because its ranking is higher than any of the NorthScope, the window (window 2) is repositioned to face the FocalPoint icon. The FocalPoint near the entrance has a similar effect on the garage door (Door 3).

The two view icons supported by ActiveContext are introduced to represent the good view towards the garden and the unsightly view towards the southern edge of the site. The UnsightlyView icon has a Negation Value (see section 2.6.3) and therefore no windows are added to any walls within its scope. The GoodView moves the window from the North-facing wall (in example1) to the wall facing the garden.

Example 3:

This example demonstrates partitioning. Two AreaScope icons are introduced to the drawing used in example 2. Partitioning allows a user to apply a unique set of context parameters to a section of the drawing. Figure 3.8 shows the Global rankings of all the context icons. These rank values are used to determine where opening are placed in all the areas outside the AreaScopes. A different criterion—based on the rank values of the AreaScope shown on Fig. 3.9—is used to place openings within the AreaScope. For example, within AreaScope

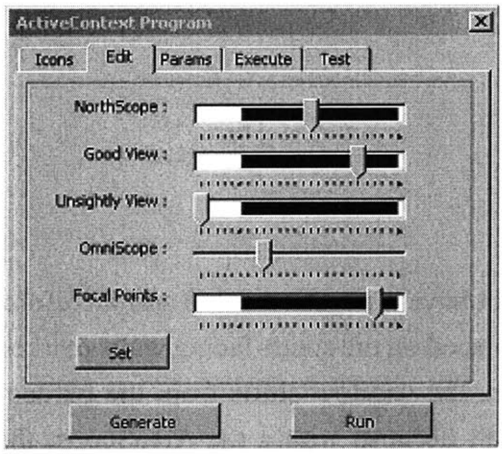


Fig. 3.8 Global Ranks for Example 3.

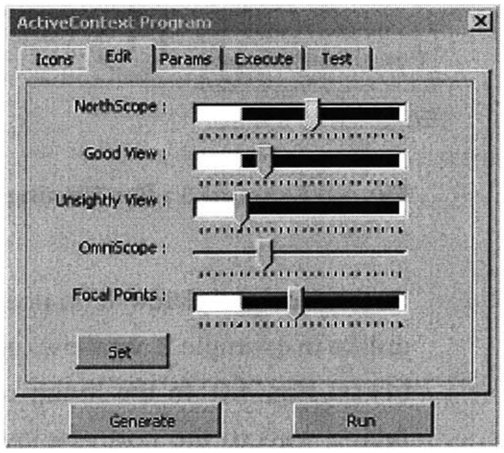


Fig. 3.9 Ranks for AreaScope A and B

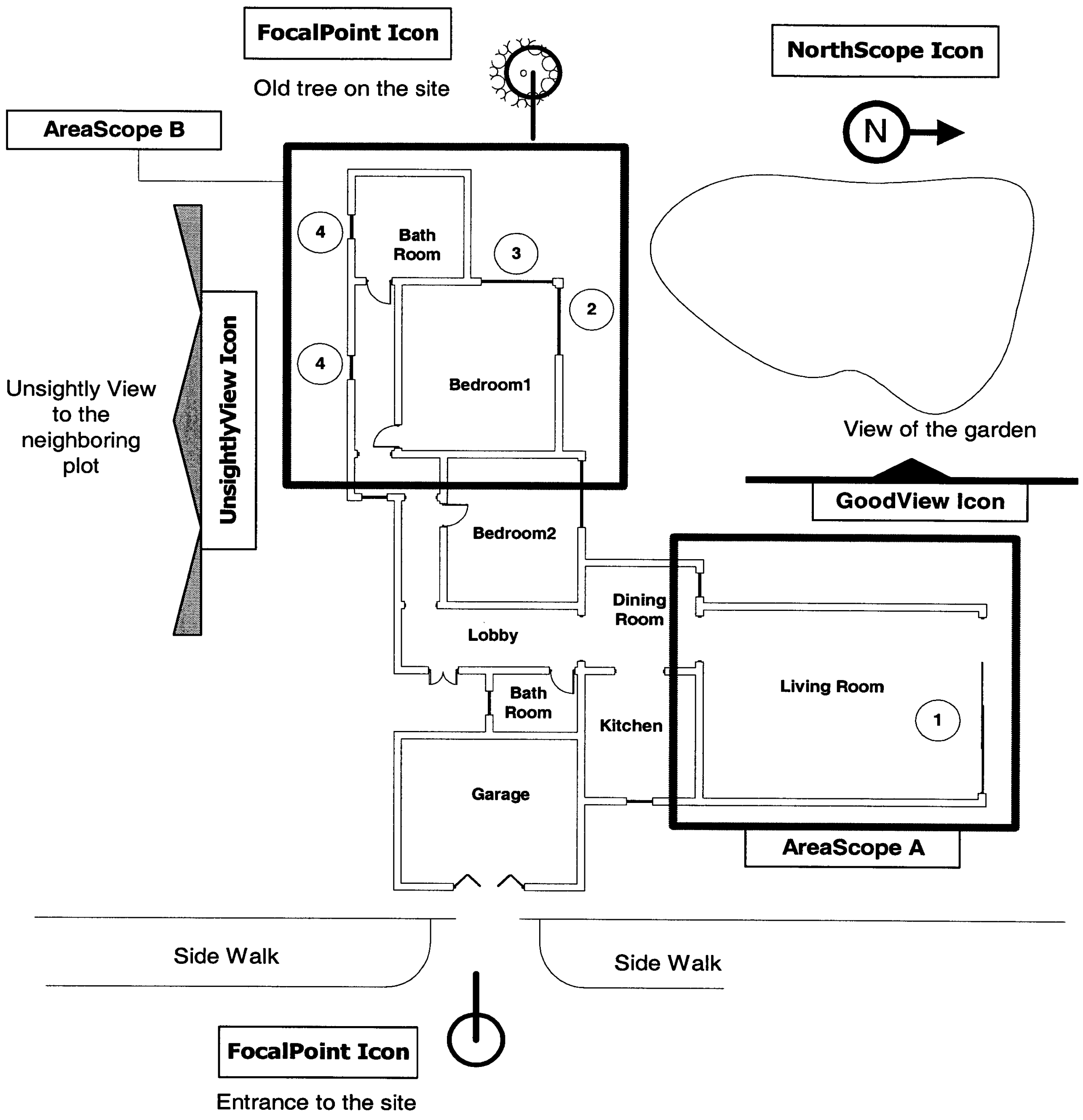


Fig. 4.0 Example 3 : Partitioning

B, the UnsightlyView Icon does not have a negation value and therefore, unlike in example 2, windows are placed on the south-facing walls marked (4) on Fig. 4.0. In the living room, the window shifts from the garden-facing wall to the north-facing wall because within the AreaScope, the North-South orientation is given a higher rank value than GoodView (see Fig. 4.0).

Chapter 4

Implementation

4.0 INTRODUCTION

This chapter describes the implementation design of the ActiveContext application, a context modeling system prototype. ActiveContext is a plug-in to the Architectural Desktop Computer Aided Design software and supports design context modeling using a rule-based system.

The context modeling architecture proposed uses the ObjectARX technology in the Architectural Desktop Software and C Language Integrated Production System (CLIPS). CLIPS is an expert system language that supports three types of programming paradigms: Rule-Based, Object-Oriented and Procedural Paradigms. ActiveContext uses both the ruled-based and object-oriented aspects of CLIPS. CLIPS and ObjectARX technology are briefly outlined in the following sections.

4.1 CLIPS

As mentioned earlier CLIPS, is a multi-paradigm language that provides support for rule-based, object-oriented and procedural programming which only supports forward chaining. Back chaining is only possible through a workaround. CLIPS is highly portable and can be installed in a wide range of computers ranging from PCs to Cray Supercomputers. It was designed using C programming at NASA/Johnson space center with the intention of providing a system that was low cost, highly portable and easy to integrated into external systems.

Using an ActiveX component—CLIPSActiveX—it is easy to embed CLIPS within the AutoCAD VBA framework and this is one of the reasons why the CLIPS shell is used in the ActiveContext application. Embedding the ActiveX component within AutoCAD VBA seamlessly integrates a rule-based system and a CAAD program making it easy to build knowledge representations upon which inferences can be made.

4.2 ObjectARX TECHNOLOGY

ObjectARX™ is an object-based development environment for AutoCAD-based applications. This application programming interface (API) has full access to the AutoCAD kernel, giving applications build on top of it speed, integration, and compactness. This framework allows software developers to extend AutoCAD's functionality using C++ and ActiveX Automation. This framework includes the basic drawing entities like lines, splines, and arcs that are used by all AUTOCAD applications. AutoCAD stores objects, (e.g. Lines, polylines and arcs) created in drawing in an object-oriented database. These objects are displayed on the computer screen by the Graphics System (GS).

AEC Object Modeling framework (OMF) is an extension of ObjectARX. It enables the development of Architectural, Engineering and Construction applications using parametric objects that correspond to real-world building elements. It allows programmers to leverage common development tools not available in the ObjectARX framework and provides direct access to a big library of existing AEC objects. Some of the objects supported by the OMF are AecDbDoor, AecDbWall, AecDbWindow, which correspond to a door, wall and window respectively.

The AEC OMF objects mimic the form, function and behavior of their real-world counterparts. Objects created using OMF are parametric. Each object has a number of variables that alter its physical shape as well as its behavioral properties. These objects interact with each other according to their real world properties. For example, when a window is placed on a wall, it automatically creates a hole in the wall and places itself within the wall within intervention from the user.

Figure 4.1 shows the class hierarchy of the Object Modeling Framework. As mentioned earlier, this framework is an extension of ObjectArx and most of its classes are subclasses of ObjectArx classes.

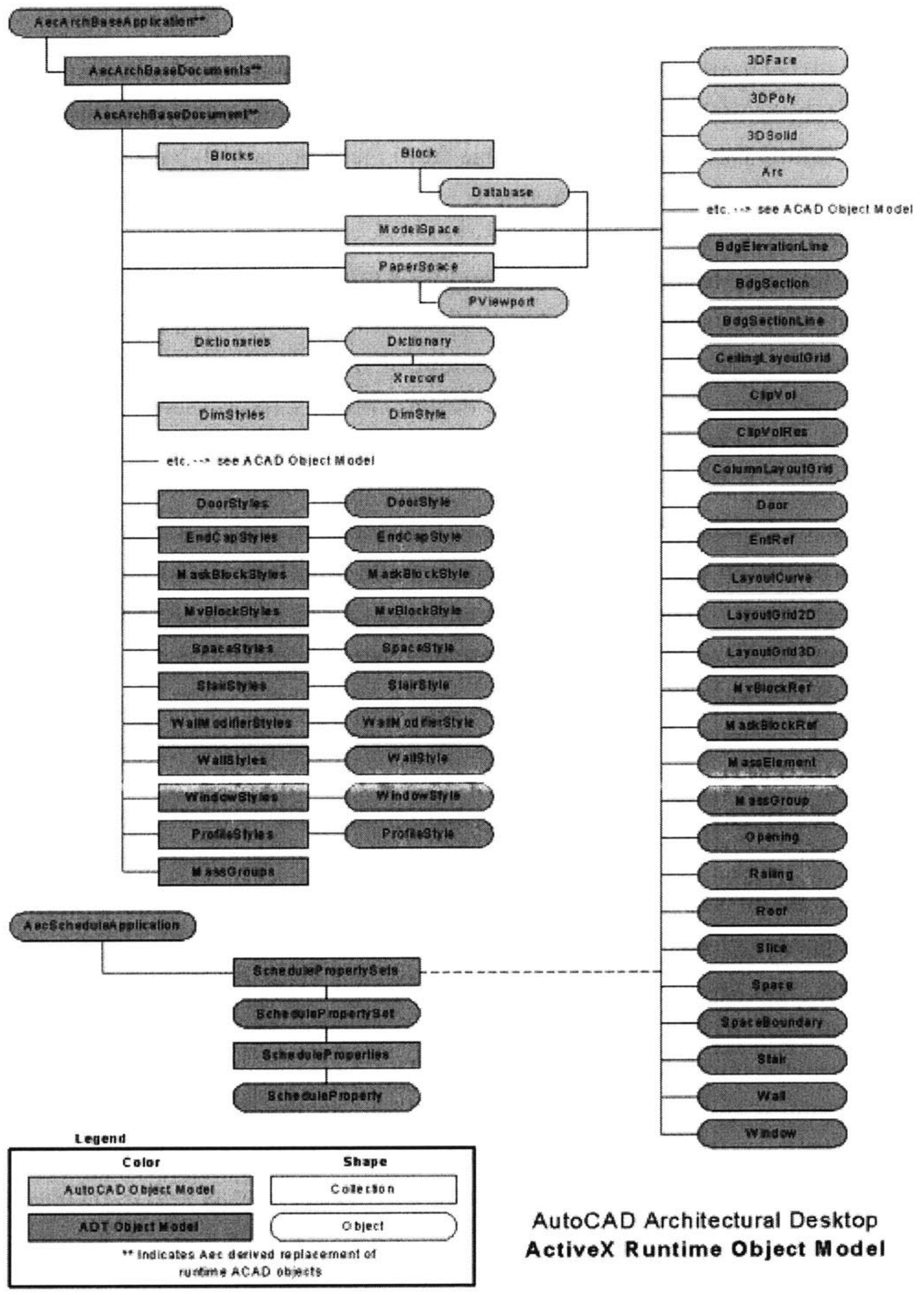


Fig. 4.1 Object Model of the Object Modeling Framework (OMF) . [from adn.autodesk.com]

4.3 ActiveContext

ActiveContext is an add-on application embedded in AutoDesk's Architectural DeskTop (ADT). It extends the functionality of Architectural Desktop by extracting knowledge from the Autocad database and making inferences using CLIPSActiveX.

The ability to map design context into a digital drawing is only useful if it can assist in solving some design problem. As mentioned in the last chapter, ActiveContext automates placement of openings based on the level of importance placed on its three design contexts (NorthScope, Good/Bad views and Focal Points) and predetermined adjacency conditions.

ActiveContext takes advantage of the elaborate parametric object model provided in the OMF framework to build up relationships and automate task execution based on rules fired in the CLIPS knowledge based tool. In summary, the following components make up ActiveContext :

1) Context icons
2) Knowledge Processor
3) Command Parser
4) CLIPSActiveX (a knowledge building tool)

As discussed in the last chapter, context icons are the graphical representations of design context in the drawing. They allow users to specify the type, the scope and the relative importance of a design context. They are used by the knowledge processor to mine architectural knowledge from the information in the database. The current version of ActiveContext contains only a few rules, all of which are involved in placement of openings. It is important to mention that the knowledge base can be extended by adding more rules to the knowledge base as well as increasing the number of context icons.

The fourth component, the CLIPSActiveX, is a software tool used for construction of an empty knowledge based system. CLIPSActiveX provides the ruled-based functionality that ActiveContext uses to represent knowledge, define rules and make inferences. This component is responsible for the "reasoning" capability of the system. Other KBS building tools that can be used are high-level KBS programming languages like Smalltalk, Jess and Prolog.

The last component is the command parser, which executes the output from the CLIPSActiveX component. It parses commands and calls procedures responsible for creating and manipulation of objects in the drawing. ActiveContext uses the large library of parametric objects in the OMF framework to add openings to walls. In the subsequent sections, the design and implementation of each of the components shall be discussed in greater detail.

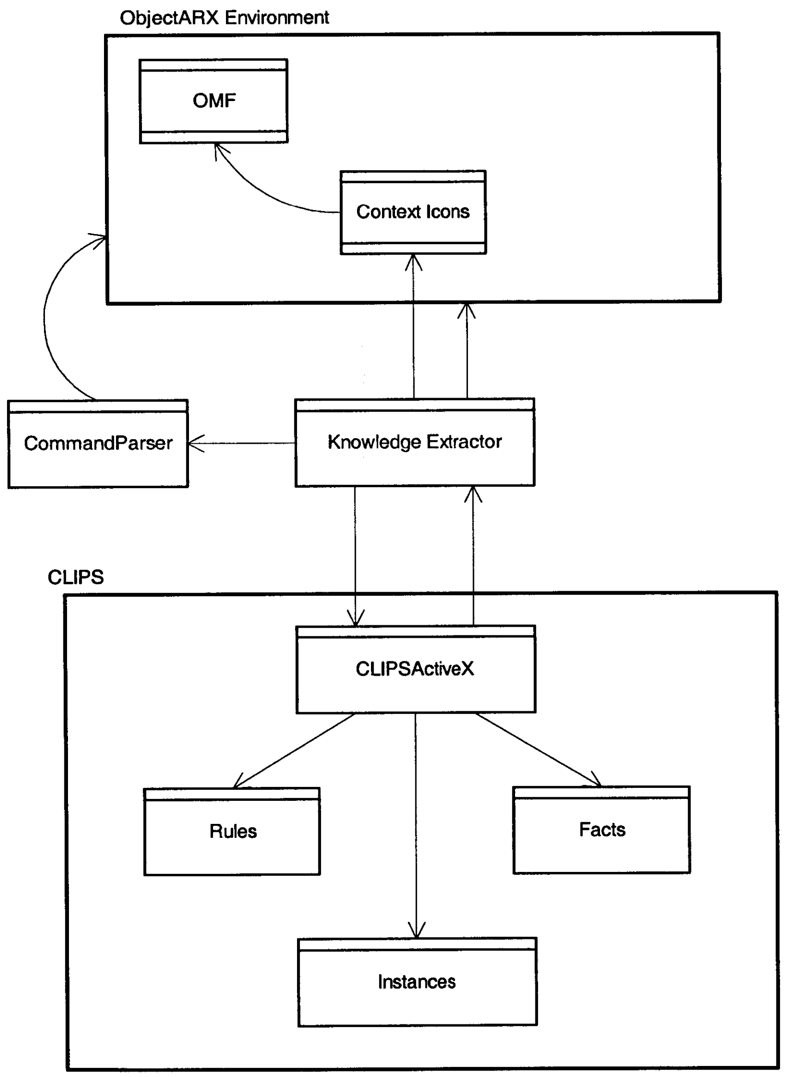


Fig. 4.2 ActiveContext Component Architecture

4.4 CONTEXT HIERARCHY

All context icons are members of a context type hierarchy. At the top of this hierarchy is the AbstractScope type from which all the other context icons are derived. This hierarchy defines the following abstract classes — PointScope, CurveScope, AreaScope and VolumeScope—which allow the definition of context icons that related to points, curves, surfaces or volumes respectively. New context icons can be introduced by inheriting from types already on the tree. The figure below shows the the context icons object model.

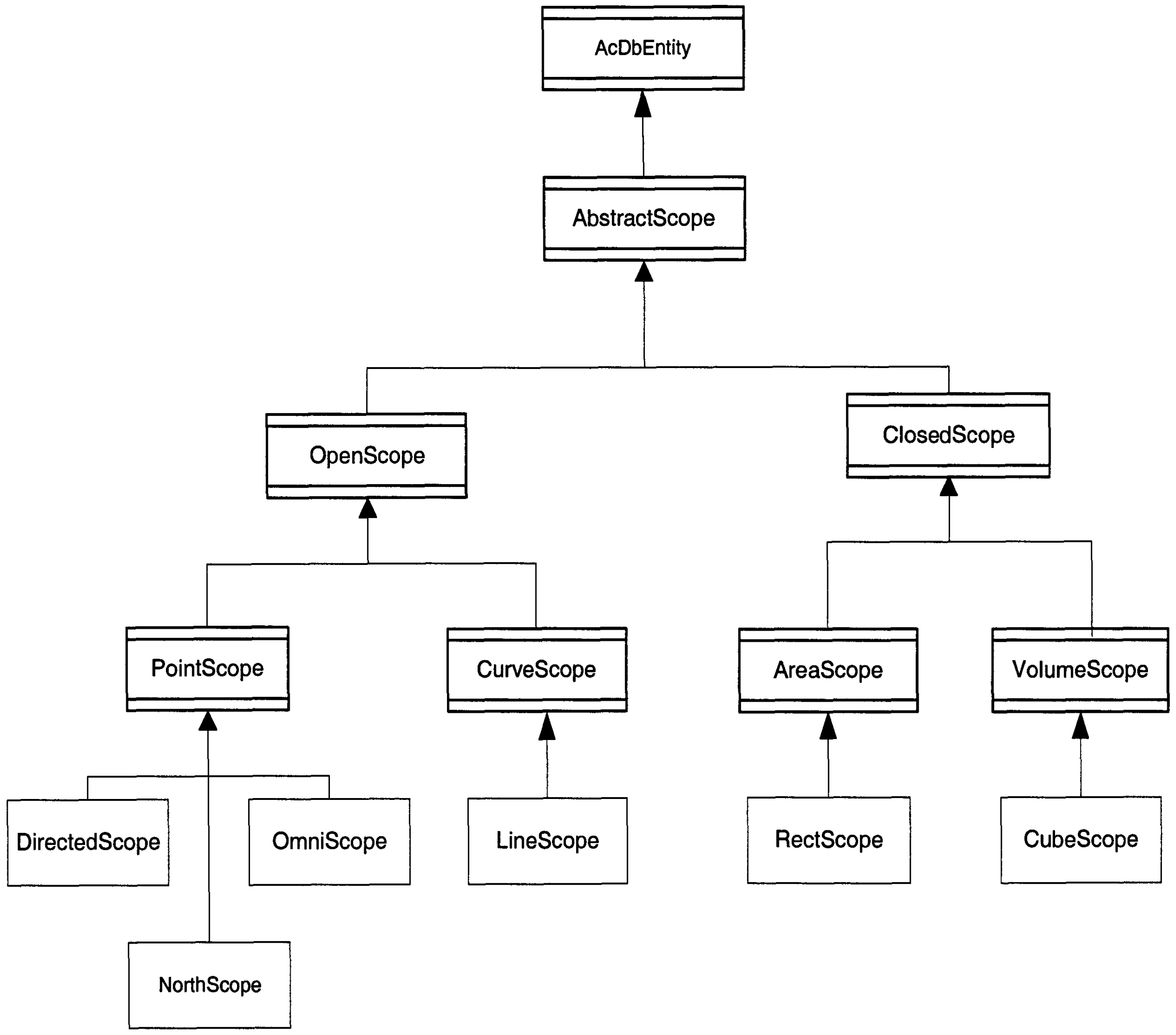


Fig. 4.3 Context Icons Type Hierarchy

In this hierarchy, there are several abstract classes that provide partial implementation for inheriting classes. In the current version of the ActiveContext, all the abstract classes defined are AbstractScope, OpenScope, ClosedScope PointScope, CurveScope, AreaScope and VolumeScope.

4.4.1 AbstractScope object

AbstractScope is the root of context icon hierarchy. It is derived from the AcDbEntity class which is the base class for all database objects that have a graphical representation. AbstractScope implements all the methods common to all context icons. Below is a partial specification of the AbstractScope class.

```
class AbstractScope : AcDbEntity {

    /* fields */
    Collection scopeObjectsList ;
    Point3d Location ;    // Location of the scope
    int Rank ;            // the rating of this scope
    CString scopeName ;   // name of this scope.

   //Effects: From the input, it searches the list for
            all objects that are within the scope.
   //Param: components, a list of objects to filter from
   //Return: Returns a list of objects from the input
            List
    List getScopeObject(List components) = 0 ;

   //Effects: Given a list of objects this method
            filters the list and obtains adds objects
            that are within this scope to the
            scopeObjectsList list
    void AppendObjectsToScope(List handles) ;

   //Effects: Returns all the objects that are part of
            this scope.
    Collection GetAllScopeObjects() ;

   //Effects: When called on any scope object, it
            removes all the objects stored in the
            scopeObjectsList list.
    Bool ClearList() ;

    ... continued next page
};
```

```
class AbstractScope {
    .. continued

    /* getters and setters */
   //Effects: Set the base point of the scope icon.
    boolean SetLocation(Point3d basePoint) ;

   //Effects: Set the rank of the scope icon.
    boolean SetRank(const int vRank) ;

   //Effects: Set the name of the scope icon.
    boolean SetName(CString vName) ;

   //Effects: Set the base point of the scope icon.
    Point3d GetLocation(Point3d basePoint) ;

   //Effects: get the rank of the scope icon.
    int GetRank(const int vRank) ;

   //Effects: Set the name of the scope icon.
    CString GetName(String vName) ;

} ;
```

All the methods of the AbstractScope class are full implementions except for getScopeObjects(). All concrete classes must provide full implementations of this abstract method. These specific implementations differentiate how context icons select scope members. Through polymorphism, all the different context icons are called with the same method but the set of drawing entities selected as scope members depends on the object type. This simplifies coding especially in the Knowledge Processor where all the different types are used.

Selection of scope members is triggered by a call to an icon's AppendScopeObjects procedure from the Knowledge Processor. This method then calls the corresponding GetScopeObjects() method and adds the result to the scopeObjectsList field of the class. To retrieve all the objects with an icon's scope, the GetAllObjects() method is called. Since this action of appending objects to the scopeObjectList is common to all classes, this method, unlike GetScopeObjects(), is implemented in the AbstractScope class. Finally, the set and get methods change and retrieve the values of rank, name and location from the scope object.

4.4.2 Open and Closed Scopes

ActiveContext makes a distinction between context icons that select objects through enclosure and those that use other criteria. Context icons that enclose their members are ClosedScopes and those that don't, belong to the OpenScope family. ClosedScopes are those that define a 2D or 3D space within which their effect can be felt. For example, a RectScope context icon selects objects that are enclosed by a rectangular shape. OpenScopes on the other hand, search for the objects that are nearest to them. The point and line family of scopes are derived from the OpenScope type while AreaScopes and VolumeScopes are subtypes of ClosedScopes.

```
class OpenScope : AbstractScope {

      .. all AbstractScope properties

      /* fields */
      Vector3D direction.

      /* getters and setters */
     //Effects: Set the direction to which the effect of
              this scope should be felt.
      boolean SetDirection(Vector3D dir) ;

   } ;
```

OpenScopes have a direction member field that defines the direction towards which they base the selection of scope members. OpenScopes and ClosedScopes are subtypes of the AbstractScope class. As such they inherit all the properties of the superclass but because they do not implement the getScopeObjects() method, they too are abstract classes and therefore cannot be instantiated.

4.4.3 PointScopes

A PointScope is a context icon that uses a reference point to determine scope members. It is an OpenScope because it does not use enclosure to determine scope membership. PointScopes can be used to represent design factors like noise, directed light that emanate from a point. They may have one or more direction vectors, along which scope members are selected. ActiveContext incorporates three kinds of PointScopes: DirectionalScope, OmniScope and UniversalScope. The PointScope class is abstract while DirectionalScope, OmniScope and UniversalScope are concrete classes.

4.4.3.1 DirectionalScope

A DirectionalScope has a location and a single directional vector. It selects the closest object that intersects a ray from the scope's location in the direction pointed to by the vector. A directional Scope can be used with or without the space object. Figure 4.4 illustrates a Directional Scope.

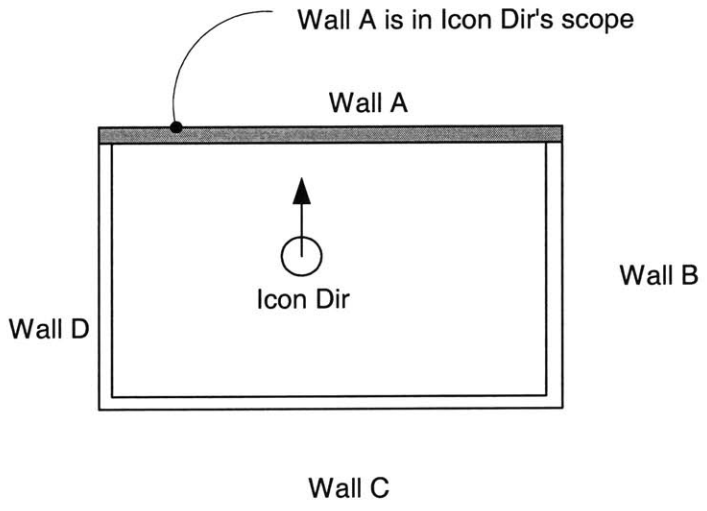


Fig. 4.4 A DirectionalScope.

The DirectionalScope class inherits the GetLocation() and SetLocation() methods from the AbstractScope class and places the ray's origin at the position returned by GetLocation(). It provides a full implementation of the GetScopeObjects() method inherited from the AbstractScope class and can therefore be instantiated. When passed a set of elements, this method returns the drawing entity closest to the origin in the direction of the ray. The FocalPoint context is a subtype of DirectionalScope.

4.4.3.2 OmniScope

An OmniScope icon selects scope members from within a range that can be described by two direction vectors (see Fig. 4.5). It is similar to the DirectionScope but instead checks for objects that are within a certain range as opposed to a particular direction.

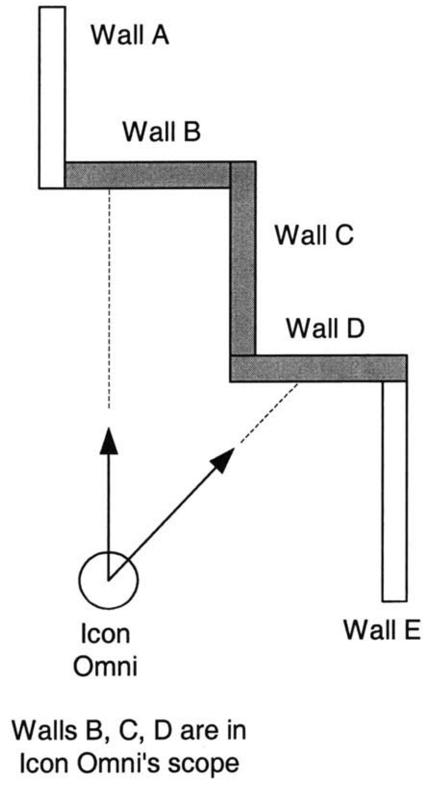


Fig. 4.5 An OmniScope.

4.4.3.3 NorthScope

This is a specialized PointScope that determines an entity's orientation within a space (see Fig. 4.6). The icon gets its name from the fact that, like a compass, it always points to the north direction. It returns the position of all the elements passed into the GetScopeObjects() method in terms of compass directions e.g. East, Northwest, South, Southwest, etc. The NorthScope scope is dependent on a space object from which the orientation of the argument entities can be computed. The orientation position, returned by the NorthScope, can be used to rank entities in a drawing based on their position within a space. For example, if it is most appropriate to place a window on the "south most" side in a room, a

NorthScope context icon can be used to determine the correct wall from the set of walls in that space. Each direction that the NorthScope returns is assigned a rank. After querying the NorthScope icon, these rankings are asserted as facts in the knowledge base.

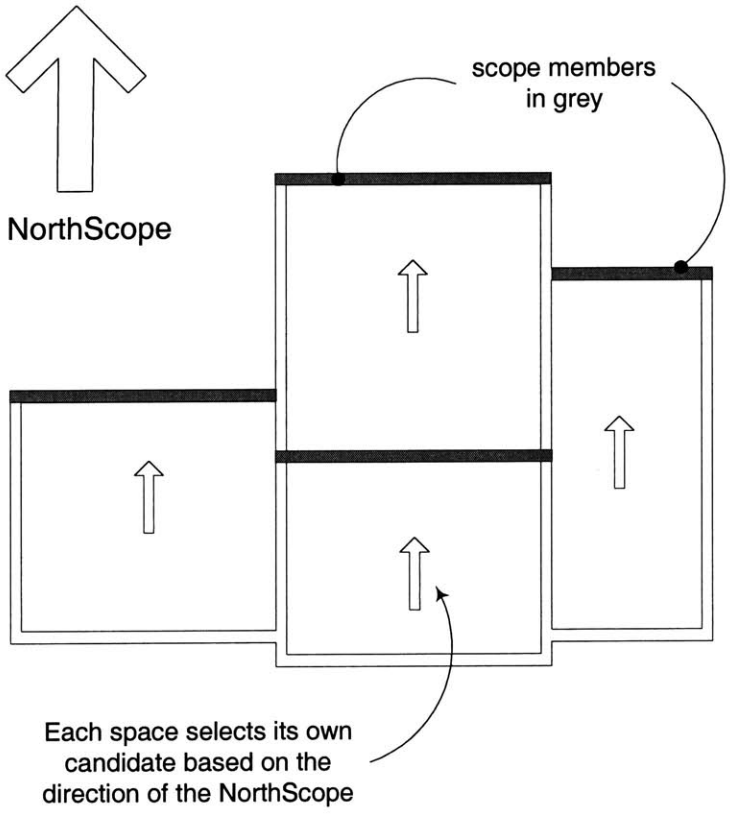

scope members in grey
NorthScope
Each space selects its own candidate based on the direction of the NorthScope

Fig. 4.6 LineScope.

4.4.4 CurveScope

CurveScopes are OpenScopes that use a curve to select scope members. A curve is a 2D entity like a line, arc, circle or spline. In ActiveContext, CurveScopes are the most useful context icons. They can be used to simulate design factors like site boundaries, plot line, paths etc. A CurveScope has a start point and end point. They can form loops and still be considered true CurveScopes as long as scope members are selected based on their relationship to the path followed by the curve as opposed to being enclosed in the loop. Examples of subtypes of this class are LineScope and SplineScope.

4.4.4.1 LineScope

As the name suggests, a LineScope is based on a line segment. Scope members are selected based on their proximity to a line segment starting at the start point to the end point of the LineScope. A LineScope can be used to group together entities that are located in different spaces e.g. walls of a façade. The current implementation of LineScope in ActiveContext performs a scan along the length of the LineScope to select its scope members. The closest entity touching a perpendicular vector to the line is added to the scopeObjectsList. Although this technique functions well, it is not the most elegant algorithm to use. LineScopes can be used to represent site lines, plot boundary, view lines, etc. In fact, GoodView and UnsightlyView context icons in ActiveContext are LineScopes.

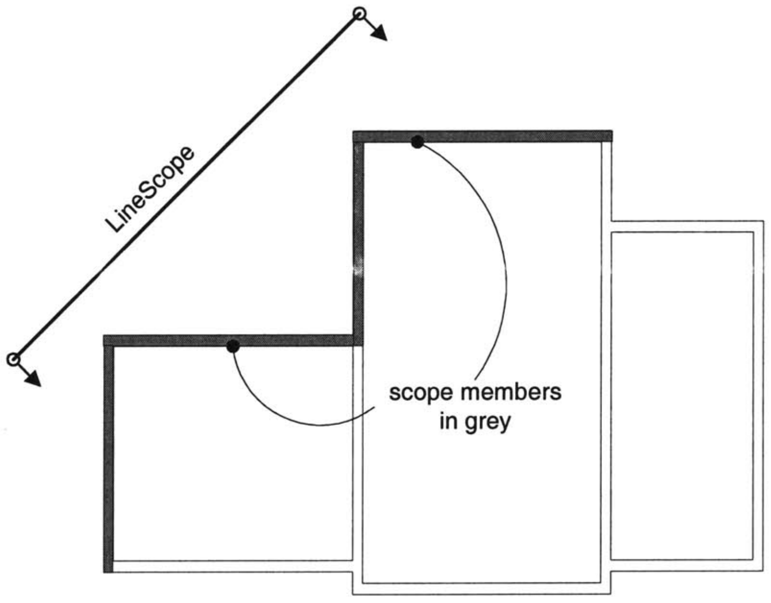


Fig. 4.7 LineScope.

4.4.5 AreaScope and VolumeScopes

These two classes are direct subtypes of the ClosedScope family. Both types select scope members by determining whether entities are within the space they enclose. A VolumeScope can be used to group together entities that are within a 3D space. An AreaScope is used to relate entities enclosed within a 2D space. In ActiveContext, the only concrete subtype of ClosedScope is RectScope. The RectScope class is a subclass of the AreaScope class, which provides methods to determine whether an object is within or just touching a polygon. The ClosedScope class contains methods common to both AreaScopes and VolumeScopes, like GetExtents which returns the geometric boundary of a context icon.

4.4.5.1 RectScope

A RectScope has a rectangular shape and scope members are either partially or totally enclosed within the bounding box of the rectangular outline. It can be set to either one of two modes: crossing or window. If a RectScope is in the crossing mode, any object that touches, or is within, the icon becomes a scope member. In window mode, only those entities that are totally enclosed within the icon are selected. ActiveContext uses this type of AreaScope for partitioning.

4.5 KNOWLEDGE PROCESSOR

The Knowledge Processor is the component responsible for mining data and information from ADT's database. When ActiveContext is initialized, the Knowledge Processor scans the drawing and asserts facts in the CLIPS ActiveX component. The output from the CLIPActiveX component is sent to the CAD system as commands. Figure 4.9 shows a partial specification for the Knowledge Processor module.

One of the tasks performed by the Knowledge Processor is building an adjacency graph of spaces and their neighbors. The procedure GenerateSpaces(), creates the graph, locates each AecDbSpace object in the database and obtains all its neighbors. Once the graph is built, other procedures use it to assert facts in the CLIPSActiveX component. For example, InstanceSpaceFacts() gets all the nodes in the graph and creates CLS_SPACE instances in the CLIPSActiveX component. The piece code below shows a CLS_SPACE instance that is generated from the InstanceSpaceFacts function.

```
([HC1C] of CLS_SPACE (HANDLE "C1C")
   (COLOR white) (LENGTH 0) (ROOM_NAME "Kitchen")
   (SPACE_WALLS [H12CC] [H12CB] [H12CA] [H12C9])
   (WALL_LOCATION "WEST" "NORTH" "EAST" "SOUTH")
   (WALL_COUNT 4) (WALL_RANKS 1 8 3 7)
   (COERCE_ABS NO NO NO NO NO)
   (EXCLUDE_ABS NO NO NO NO NO)
)
```

Fig. 4.8 Sample CLS_SPACE Instance

class KnowledgeProcessor {

```
    // fields
    CLIPSActiveX clipsComponent ;

   //Effects: Creates a Knowledge Processor object
            and sets clipsComponent to component
    KnowledgeProcessor(CLIPSActiveX component) ;

   //Effects: Returns a collection of spaces that
            are adjacent to the space argument
   //Param: handle, The unique String identifier for
            the space
    Collection GetAllAdjacentSpaces(String handle) ;

   //Effects: Creates an adjacency graph of spaces
            and their neighbors
    Graph GenerateSpaces(String handle) ;

   //Effects: Returns a collection of all spaces in
            the drawings
    Collection GetAllSpaces() ;

   //Effects: Instantiates CLS_SPACE objects in
            component
   //Param: component, The CLIPSActiveX component
   //Return: Returns True if no errors occur, false
            otherwise.
    boolean InstanceSpaceFacts(CLIPSActiveX
component) ;

   //Effects: Asserts LinkedSpaces facts in
            clipsComponent
    boolean AssertLinkedSpacesFacts() ;

};
```

Fig. 4.9 Specification for Knowledge Processor module.

```
(MakeOpening
  (FIRST "Kitchen") (SECOND "Lobby") (WALL "B23")
)
```

Fig. 4.10 Sample LinkedSpace Fact

Figure 4.10 shows an example of fact asserted by AssertLinkedSpacesFact() procedure of the Knowledge Processor. Once in the working memory of the CLIPSActiveX component, this fact causes a rule regarding placement of openings on walls to be inserted in the agenda.

When the knowledge processor has asserted all the facts into the CLIPS component, the component is activated causing rules to be fired by the inference engine. The output from the rules fired is then sent to the command parser, which parses the command, and manipulates the drawing according to the inferences made.

4.6 COMMAND PARSER

The output from the inference engine is in the form of character string commands. These commands dictate the type, size and location of the openings to be placed within the drawing. The CommandParser calls subroutines in Architectural Desktop Application that instantiate doors and windows from the parametric object library. The CommandParser also handles errors that occur when the CLIPSActiveX component is making inferences.

Chapter 5
Conclusion

This chapter concludes this thesis with a summary of its contributions, possible applications and a discussion of future work.

5.1 SUMMARY

This thesis contributes a system—called Context Modeling System (CMS), which allows the representation and extraction of design context from a CAAD model. The context modeling system proposed enables a designer to explicitly represent and prioritize contextual issues in a CAAD drawing. This thesis highlights several issues that have to be addressed by any implementation of a context modeling system. These issues include: explicit graphical representation of design context, ranking of design contexts, conflict resolution when design contexts clash, partitioning and extensibility. A model is meaningless if it cannot be implemented; hence an implementation of the Context Modeling System—called ActiveContext—is presented which, is based on a simplified design world with three design contexts.

One of the primary goals of this work was to find ways in which design contexts could be incorporated into CAAD systems. To achieve this, symbols called context icons are introduced into the CMS model. Using context icons, design contexts are explicitly represented in the drawing so that designers can define and prioritize them. In the prototype ActiveContext, a context icon is a parametric object with a rank variable that a designer can change to reflect the importance he/she attaches to the represented design context. A framework—the context icon hierarchy—was developed from which more context icons can be derived to increase the types of icons available.

In order to allow a designer to prioritize design contexts, a CMS requires a ranking mechanism. Two ranking schemes are proposed. In the first, full ordering, design contexts are assigned absolute values and—unlike in the second (partial ordering)—relationships can be deduced by comparing independent contexts. This thesis also emphasizes the need to support declaration of absolute certainty using the ranking mechanism.

Designers frequently find themselves in situations where two or more contextual issues interfere with each other. One design context may require a response that contradicts another. This thesis proposes a conflict resolution mechanism that context modeling systems can use to mediate such conflicts. The proposed intermediation strategy uses strengthening, weakening, contradiction and elimination to resolve conflicts.

This thesis also asserts that a context modeling system ought to provide a way to partition the design space into smaller parts that can be treated autonomously. Partitioning makes the system more scalable. Using partitioning, a large design problem is broken down into smaller parts that can be given their own contextual considerations. This allows a designer to heterogeneously prioritize design contexts in a project. Hence, a designer has the freedom to rank context, based on other design parameters like space type or location within the site. Partitioning is important because in real life designing, different spaces or building types within the same project might require different contextual considerations.

5.2 APPLICATIONS

The practice of architecture is increasingly becoming a collaborative effort. This is evident as design projects become larger and more complex, and design tasks become more specialized. In such collaborative environments, CAAD systems must take into account the different contributions each designer may make. As was discussed before, contextual considerations are very important in the design process. Consequently, collaborative environments need a way for designers to communicate with one another—even in remote locations—their interpretation and/or response to context. CMS offers a means to do so.

As the ActiveContext prototype demonstrates, CMS can be used to develop context-sensitive design tools as plug-ins or extensions to existing CAAD applications. These tools can offer suggestions to designers about how to respond to contextual issues as well as assist designers perform design tasks automatically. Since CMS stores design knowledge, designers can customize applications to respond to contextual issues in a user-defined fashion.

Lastly, architectural pedagogy is another area that stands to benefit from context modeling applications. Educators can use context-sensitive learning tools to teach students how to respond to contextual design issues. Since CMS is integrated within a CAAD framework, students can experiment with context in a hands-on manner while modeling or drawing with the CAAD system.

5.3 FUTURE WORK

The ActiveContext prototype was based on a simplified design world with only three design contexts. In that model, there was only one design variable i.e. the placement of openings. Such a simple model was necessary in the investigation of a complex activity like decision-making using design context. In order to explore context modeling in a more sophisticated design environment (like the real world), the functionality of this ActiveContext can be extended in various ways.

The first would be to increase the number and complexity of design variables that the program handles. Since ActiveContext already incorporates a knowledge building tool, the program could be extended to perform more design tasks computationally. One possibility is to integrate ActiveContext into a space-planning program for the purpose of optimizing on existing site conditions. Another extension would be to make the program compute optimal sizes of openings—based on orientation and/or other factors—instead of using predefined sizes. This kind of enhancements give designers ideas about how to better respond to contextual issues.

Another way to extend the program is to incorporate more contextual factors. Except for the user interface, the current implementation can be extended to handle more design contexts without redesigning the entire system. Since the graphical interface used to prioritize design contexts lists all contexts in one window, if the number of context icons supported becomes too high, the program could become too cumbersome to use. This opens up interesting possibilities; one way to tackle this problem, at the interface design level, would be incorporating a reframing mechanism in which, by focusing on an icon or button, more and more details related to a design context could be exposed to a designer. Another approach, is to incorporate a learning mechanism which acts on behalf of the user according to the choices made by the user in the past.

An important extension to the CMS is, to provide a framework through which designers (not just programmers) can create or modify design variables and context representations. The current prototype requires a user to have a reasonable amount of programming experience to augment the system because it does not include a simple "programming interface" that designers can use to define their own variables and context representations. By providing such an interface, designers can create new custom tools which they can use to explore design alternatives.

Lastly, future work should explore new design worlds or environments in which design variables and context can be manipulated. ActiveContext uses 2D icons and rule-based technology to model context and this is by no means the only way context can be modeled. Through the invention of new context-sensitive design worlds and the elaboration of existing ones, designers stand to gain new ways to exploit CAAD in the creative part of the design process where CAAD is rarely used.

Bibliography

[1] Addis T. R. (1985) Designing knowledge-based systems. Kogan Page Ltd.

[2] Akman, Varol (1995) Book Review — Vladimir Lifschitz, ed., Formalizing Common Sense: Papers by John McCarthy. Artificial Intelligence 77(2): 359-369.

[3] Akman, Varol and Surav, Mehmet (1995) Contexts, Oracles, and Relevance. In Buvac, Sasa, Eds. Proceedings AAAI-95 Fall Symposium on Formalizing Context, pages 23-30, Cambridge, Massachusetts.

[4] Akman, Varol and Surav, Mehmet (1997) The Use of Situation Theory in Context Modeling. Computational Intelligence: An International Journal 13(3): 427-438.

[5] Burbery, P. (1983) Environment and Services. B T Batsford Ltd, London.

[6] Burden, E. E. (1934) Elements of architectural design. John Wiley & Sons.

[7] Coyne R.D., Rosenman M.A., Radford A.D., Balanchandran M., Gero J.S. (1990) Knowledge-Based Design Systems, Addison-Wesley Publishing Company, Reading Massachusetts.

[8] Carrara G., and Kalay Y. E., (1994) Knowledge-based computer-aided architectural design, Elsevier science B.V. Amsterdam.

[9] Davis R., Shrobe H., Szolovits P. (1993) What is a Knowledge Representation? AI Magazine, 14(1): 17-33

[10] Giarratano J. C., Riley G. (1998) Expert Systems: principles and programming. PWS Publishing Company, Massachusetts.

[11] J. McCarthy (1993) Notes on formalizing context. In Proceedings of the Thirteenth International Joint Conference in Artificial Intelligence, Chambery.

[12] Kokinov, B. (1995) A dynamic approach to context modeling. In P. Brezillon & S. Abu-Hakima (Eds.), Proceedings of the IJCAI-95 Workshop on Modeling Context in Knowledge Representation and Reasoning. LAFORIA 95/11.

[13] Liskov B., Guttag J. (2000) Program development in Java: abstraction, specification, and object-oriented design. Addison-Wesley Publishing Company.

[14] McCarter, R. (1997) Frank Lloyd Wright. Phaidon Press Limited.

[15] McCullough, J. (1991) Knowledge based systems in architecture, Stockholm: Swedish Council for Building Research.

[16] Meyer, B. (1988) Object-Oriented Software Construction. Prentice Hall.

[17] Angeles, P.A. (1981) Dictionary of Philosophy. Harper and Row Publishers, New York.

[18] Rambaugh, J. (1991) Object-Oriented Modeling and Design. Addison-Wesley.

[19] Reffat, R. and Gero, J. S. (2000) Computational situated learning in design, in Gero, J. S. (ed.), Artificial Intelligence in Design'00, Kluwer, Dordrecht , pp. 589-610.

[20] Rosenman M.A., Maher L.M., Gero J.S. (1994) Knowledge based-research at the key centre of design computing, Knowledge-based computer-aided architectural design, Elsevier science B.V. Amsterdam, pp. 354.

[21] Simon, H. A. (1969) The sciences of the Artificial. The MIT Press, Cambridge Massachusetts.

NOTE: Unless otherwise noted, all illustrations are by the author.