DOMAIN AND CONSTRAINT VISUALIZATION IN COMPUTER-AIDED DESIGN

by

Enikő Ilona Rózsa

M.Sc. Technical University of Budapest, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in the School
of
Engineering Science

ISBN 0-612-01108-9

APPROVAL

Name:	Enikő Ilona Rózsa
Degree:	Master of Applied Science
Title of thesis:	Domain and Constraint Visualization in Computer-Aided Design

Examining Committee: Dr. W. Havens
Associate Professor of Computing and Engineering Sciences, Chairman

Dr. J.C. Dill
Professor of Engineering Science
Senior Supervisor

Dr. J.D. Jones
Associate Professor of Engineering Science
Supervisor

Dr. T. Calvert
Professor of Engineering Science
Examiner

Date Approved: 21 March 94

Abstract

The objective of this research is to explore interactive intelligent design with visual aids to the designer. Straightforward CAD/graphics methods work at too low a level; they cannot interpret drawn objects. Through an appropriately structured knowledge base, an expert system can supply the needed interpretation, can keep track of design relationships; it is also desirable if the expert system can play a more active role, suggesting design steps. Further, since automated design seems unlikely and since design is an iterative interactive process, an approach whereby both system and designer contribute to the solution is desired. Further, since designers work visually, visualization tools are needed to supply graphic feedback.

This thesis describes FLOWER, *Floor LayOuts With Expert Recommendations*, a system for assisting with the layout of the floor plan of a house. A model-based reasoner with constraint propagation was combined with computer graphics visualization techniques to achieve this goal. The reasoner generated information related to the domains of design variables and constraints between them. This information is made available to the graphics side of the system for visualization, helping the designer see and understand the design space at each step of the interactive, iterative design process.

FLOWER and the user work together in a mixed-initiative style: the system gives "hints" to the designer about the outcome of certain design choices. For example, when the designer chooses to place a room, the system shows the acceptable areas of placement for that particular room. The system also provides feedback about the choices, approving acceptable ones while indicating and explaining errors when they occur.

Dedication

To *Jennifer* who was happily playing in Kahpoo while mommy was finishing her thesis and to *Shahram* for all his encouragement and support.

Acknowledgments

My sincere thanks to my supervisor Dr. John Dill for guiding and encouraging me to work in this very interesting research topic and also supporting me throughout the course of this research. I am very grateful to my committee members, Dr. John Jones, Dr. William Havens and Dr. Tom Calvert, for their assistance and valuable comments.

I would like to express my appreciation to my colleague Cheryl Petreman, whom originally I started to work on this area with. I would like to thank the members of the Computer Graphics Research Lab at SFU for the lively environment, especially Teo, Albert, Frank, Phil, Lyn, Sumo and Sang. Thanks is also given to Miron, Sue, Mike and Russ from the Expert System Laboratory.

This work was also supported in part by the Natural Sciences and Engineering Research Council and the EBCO Epic Chair for Expert Systems.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation of Thesis

The objective of this thesis is to implement an interactive intelligent design application with visual aids to the designer. Visualization helps a designer explore data and information in order to gain greater understanding and insight into the design process. Several existing techniques taken from object-oriented expert systems, computer graphics and user interface methodology were combined in order to achieve this goal.

A system was created that assists with the design of a floor plan of a house. Using current computer graphics technology alone for that purpose is limited in that it lacks the ability to interpret the drawn objects. Lines are simply lines without a specific meaning such as "these lines represent a room", with all the properties of a room, such as walls, neighbors and other relations to other rooms. These constraints can tie rooms together or separate them. The interpretation of the drawing is completely left to the imagination of the designer. An expert system can supply the needed interpretation through an appropriately structured knowledge base, and can easily keep track of many design relationships. However, an expert system on its own, i.e.

without visualization tools and support, cannot supply needed visual feedback to the designer. For instance, without appropriate graphic indications from the system, consequences of poor or incorrect design choices are difficult for the designer to see.

Even those previous systems with a graphics environment and an expert system suffered from a certain lack of interactiveness: i.e. their approach was to request a design goal from the designer and then work toward a solution with no further input. If not satisfied with the solution, the designer would have to restart from scratch, completely discarding the solution, even if some elements of it were acceptable. Thus we see the need to support an interactive, mixed-initiative approach.

This thesis presents a novel approach by creating a system that provides the user with expert aid with FLOWER and the user working *together* in a *mixed-initiative style.* First, the system provides feedback about designer choices, "approving" acceptable choices and indicating errors when they occur. For some error types, the designer may proceed but the error is marked. Other types of errors are unacceptable, so the system would not accept such requests from the designer. Second, the system gives suggestions to the designer about the outcome of certain design choices. For example, when the designer chooses to place a room, the system shows the acceptable areas of placement for that particular room, taking the current layout and constraint set into consideration.

1.2 The Design Process

The area of intelligent CAD has long been of interest to researchers. Before applying intelligence to CAD, it is helpful to understand how human perform this task and much has been written (e.g. [1], [2], [37]) on *design* itself, as a process. The design process can be divided into subtasks. Artificial intelligence can be applied to any of these. However, AI has so far only been successful in some of these ([2]).

Rosenmann et.al. ([41]) describe design as a "goal-oriented decision-making activity in which, given a set of goals, a designer prescribes the form of some artifact so as to satisfy a set of goals" and suggest that different levels of design can be achieved. At the "easiest" level, when given a problem, one simply selects a solution from an existing set of solutions. The goal is to make some modifications, if required. At the next level, the designer knows the general form of the artifact, but the parts and parameters of it have to be chosen. At the top level, the designer does not know even the general form of the artifact. The solution is generated creatively, with trial and error. The authors see expert systems as a tool that can be applied to the simplest level of the design and in lesser ways at more advanced levels.

Pylyshyn ([37]) views problem-solving as a process of search through a *Problem Space*. The problem statement starts with a specification of a set of states that constitute the problem space, a set of operators that can be used to move about in that space, a state (or set of states), that constitutes the starting state(s), and a state (or set of states), that is a goal state. Problems which can be characterized in this way are called *Well Structured Problems*. Problems which lack one or more of these features are called *Ill-Structured Problems*. The problem with the design process is, that it is clearly Ill-Structured. Design problems do not have a fixed problem space to search through: finding one is part of the problem. Generally ill-structured problems are solved by formulating one or more well-structured subproblems and attempting to solve them. Pylyshyn sees expert systems as capable of automating the design process, but they need to satisfy certain requirements. As an example, such systems must have the capability to express mutual constraints in such a way that consistency and progressive constraint restriction can be monitored automatically.

In [1] and [2], Akin attempts to give a descriptive model of the design behavior of architects. In his view, before attempting to make an expert system to model the process, we must be familiar with the practice of design. Akin is skeptical about existing expert systems in CAD, especially regarding their participation in the design process. The problem he sees is that there is very little known about the practical expertise of the architects. He also observes that there may be debate and disagreement about the correctness or goodness of design even among experts. The existing expert systems are no more than helping tools in the various stages of design; they do not automate the whole design process. However, he suggests that certain requirements must be considered in the development of expert system for design. One of these was in fact the same as one of Pylyshyn's realizations: it should be possible to organize problem constraints into a hierarchy, distinguishing between local and global constraints. Also special representations of design elements are needed so that dependencies between the hierarchically organized constraints and design elements can be automatically propagated. Such a tool would allow the designer to predict the consequences of modifications made at one level to elements on another.

Architectural design is generally a very complex process and is usually considered as an area for human endeavor and not for computers. The contribution of computers and CAD to automated design has mostly been limited to drawing creation. Intelligence still needs to be introduced.

Currently expert systems are introduced only in low-level architectural design involving routine, simple tasks. Such expert systems are able to perform classification and decision-making, when there is a known number of decision options. When the knowledge is described, the expert systems can be created to perform certain "design" tasks. The existing expert systems are able to generate "correct" answers as a human would, but they are not able to "explain" the relationships that make those particular choices right. At present, it is unrealistic to construct models of architectural design, without oversimplification of the design task. I believe that in the future this would be the aim of the researchers of this area.

Through the architectural design process, there are certain *Design Codes*, which must be satisfied. Design codes include physical laws, heuristic rules and experimental knowledge. It seemed a natural step to implement expert systems for checking whether the design meets the code requirement. Such expert systems would not do design, but would be a useful aid through the process. One of the significant difficulties of the implementation of such systems lies in acquiring the knowledge, even though the codes are already structured. Several authors ([14] and [40]) proposed expert systems to make use of such codes. The usefulness of such systems is significant in the architectural design, even though they are not part of the design process, only assistants to it. These systems can free the designers from routine tasks so they can concentrate on the design itself.

Other expert systems (e.g. *Fixer* ([15]), *HI-RISE* ([27], [28] and [29]), etc.) were developed to assist the designer in specific architectural tasks. These systems will be described in the next chapter.

As we can see from the above there has long been a need for a system that can support design in a more intelligent way. It is not enough any more to simply rely on passive participation in the design process. Of course there are routine tasks to be performed throughout a design; for example existing expert systems are capable of checking certain results. However, if we want a system that can be a real help when designing, we wish the system to play a more active, supportive role. We want our system to *suggest* design steps.

We also expect to be able to try out new ideas while taking design steps. We would like to make a step, see the outcome and then proceed or discard the step based on some feedback from the system. In this way we could carry on with the trial and error inherent in the design process. None of the expert systems mentioned above have this ability of providing this kind of expert aid. As a consequence, FLOWER was created to narrow the gap between systems that only passively participate in the design process and those that take over the design.

1.3 Contribution of Thesis

The previous section showed that although expert systems have been successfully used in the design process, there is a problem in that they are not capable of working interactively *with* the designer. They are passive problem solvers and give no or just minimal help to explain their actions. CAD systems are interactive, however, the designer is left alone to make all design decisions.

Therefore, the aim of our research was to create a system capable of

- visualization of domain of design variables and design constraints: users actually want to *see* how their design space changes as a result of their actions
- letting the user design in a "human" way by means of trial and error and not being fully automated
- giving visual suggestions or guidelines on how to proceed with the design
- checking design decisions and making corrective actions or marking problem steps

FLOWER was created as a first step toward fulfilling the above requirements. As noted in Chapter 2, we were unable to find a description of an existing system that addressed all of these goals, though the need for such system is recognized by many. The system closest to meeting our goals simply produced a long textual explanation for the user about the system's decision ([3]).

1.4 Thesis Organization

A literature review is presented in Chapter 2. First, we briefly describe a few expert systems in architectural design, then we give a short overview on visualization. Finally, we present design systems using constraints. The main problem is that we were unable to find papers which directly dealt with constraint visualization. It seems that the lack of readily available constraint processing systems resulted in works where the main focus was on how to *implement* the constraints as opposed to *using* them for different purposes.

The scope of our system is presented in Chapter 3. The overall problem that **FLOWER** addresses is defined first. The solution methodology is then explained, followed by the user's view of the system. Finally, the details of our visualization methods are described.

The fourth chapter describes the architecture of the system and the fifth chapter gives an overview of the user interface of the program.

The final chapter presents an evaluation of the system, outlines direction for future work and concludes with a brief summary.

A User's Manual is included in the Appendix.

CHAPTER 2

Literature Review

In this chapter we review relevant literature on our area of research. First we describe some of the existing expert system in architectural design in Section 2.1. These systems were automating design and were not attempting to involve the user in the design process. Next, in Section 2.2 we look at the use of constraints as a way to express design goals. We describe some of the early systems first. The common finding of these systems is that maintaining constraints should be left for the underlying system and should not be the responsibility of the user. Several ways of dealing with constraints were proposed by various authors. However, we want to do more than just use constraints. Our goal is to create a design environment where the system can help the designer by suggestion and explanation as opposed to just automating the entire design process. We want to *visualize* constraints and domains of variables, as a pictorial explanation is almost always more beneficial to the user than a long textual one.

Thus next we look at visualization in general and in particular visualizing relationships. In order to design our user interface such that it facilitates design and shows relations and design decision at the same time we reviewed general guidelines first. We found that the key in designing successful interface was in understanding the users and their tasks and then matching the interface to these requirements. Next, we decided to visualize relations using colours, thus we reviewed papers on colour usage in user interfaces.

In the last section we review in more detail those papers most related to our work. They can be categorized in two areas. First are those describing work on automated design systems, all of which use constraints as a way of expressing design goals. However, the user is left out of the design process. Only the final solution is presented to him/her. Papers in the second group describe systems where users took part in the design. In the one case, (s)he could browse through partial results choosing the most suitable with which to continue. This system was still almost fully automated, as the user could only choose from the system-generated solutions, and was not able to give independent input. In another case, the user was an integral part of the system communicating through a well-established link. We think that our system is capable of more than that, since FLOWER can suggest design steps, explain failures and suggest corrections.

2.1 Expert Systems in Architectural Design

In [40], the authors propose an expert system to make use of design codes. It is shown to be useful in conjunction with a comprehensive computer-aided design system. The nature and capabilities of the code checking expert system are described.

In [14], the authors propose another expert system which uses the results of the above research. It is also intended to function together with an architectural design system. After a building design has been developed, the expert system assists the designer in making sure that the plan is consistent with certain codes and regulations.

The usefulness of such systems is significant in the architectural design, even though they are not part of the design process, only assistants to it. These systems can free the designers from routine tasks so they can concentrate on the design itself.

An interactive expert system - *Fixer* ([15]) - was developed to help in the determination of fastener specifications. It is a knowledge-based system and no attempt was made to deduce specification from the underlying physics or chemistry. The interaction with the user is performed through dialogues. It will ask the user to provide data about a fastener problem and it will offer alternatives from which the user can make a selection. Following the dialogue session, the system will provide a final specification.

Gerő et.al. ([17], [35]) proposed the concept of a *prototype* as a conceptual schema for representation of generalized design knowledge. The design experience is generalized in a way that allows representation at the concept level in the form of a class from which instances may be instantiated to meet the specific design situation. Class and instance take their meaning from object-oriented programming. The development of knowledge-based systems to aid or automate the design process requires the identification of a representation schema for this design knowledge. A prototype is a generalization of grouping elements in a design domain which provides the basis for the design. Designers are capable of using prototypes and of generating new ones.

Maher ([27], [28] and [29]) developed a system called HI-RISE. It is a knowledge-based expert system that performs preliminary structural design of high-rise buildings. In the preliminary design process, the key terms are *selection* and *constraints*, in Maher's opinion. The selection of a structural configuration implies that there is a set of potential configurations from which to choose. The constraints may be grouped into several categories, ranging from subjective constraints imposed by the architect to functional constraints imposed by laws of nature.

The user takes part in the design process through the selection of a functional system to be pursued further. The design knowledge is represented in the form of schemas and rules. The schemas contain the description of the design subsystems and components, and the rules represent design strategy and heuristic constraints.

As we can see from these examples there is a need to have intelligent aid to the design process. Existing systems however, leave the user out of the design process.

2.2 Visualization

This section reviews the area of visualization and especially visualizing relationships. We were unable to find an article or book, that dealt explicitly with the problem of constraint visualization. Relatively few books exist on constraints and those that do are about constraint management systems, i.e. about their implementation. There is much work on the use of constraints or how to draw them in a graph-like manner ([4], [8], [12], [22], [23], [36] and [47]) but these are generally about mathematical/physical constraints and not logical ones.

One of the best known such constraint-based systems is Thinglab ([10]) and its follow-up: ThinglabII ([30]). Thinglab provides a set of so called things, which the user can add to a graphics work place. It lets the user attach complex graphical constraints to graphical objects: e.g. one can constrain a line to be horizontal.

A constraint is a relation that must be maintained. Using such relations proved to be helpful in constructing user interfaces. Maintaining these relations should be left for the underlying system and shouldn't be the responsibility of the user. In Thinglab constraints are used for the following purposes:

- to maintain consistency between underlying data and a graphical depiction of that data on the screen
- to maintain consistency among multiple views of data
- to specify how information is to be formatted on the screen
- to specify animation events that are to occur when a given event occurs in the underlying system
- to specify attributes of objects in animation, such as speed and trajectory

Constraints allow a declarative description of the user interface. With them the user can specify *what* relations are to hold and the system will decide *how* to keep the relations. Maintenance of consistency between data and displayed information or among multiple views of the same data is a common problem in user-interface design. The usage of constraints is one of several techniques to handle it. The advantage is that a constraint relation can satisfy itself *bidirectionally*.

As we can see, Thinglab allows the user to constrain "things" together but it does not show anything about the "outcome" of those constraints. The user can create thing1, then create thing2, then specify a constraint between them. Now, it is the responsibility of the system to make sure that the constraint is held.

The designer takes a different approach in our system. First (s)he creates room1. Then (s)he can specify a relationship between the existing room1 and a not yet created room2. When the user indicates (s)he will create room2, FLOWER shows all possible spaces where that room can be placed, based on the specified constraint. If the user places room2 on a suggested, suitable area then the constraint will be satisfied.

A second major system is Peridot ([32]) which infers graphical constraints as the user adds objects to a work area. To help Peridot infer dependency relationships between a new object and others, the user may select a particular object for Peridot's attention. The constraints function in one direction only. The system confirms inferences with the user as it infers constraints. It displays a textual explanation of the constraint it thinks the user intended. Then the user is asked to accept or reject the inference.

Several other interactive graphics systems permit the specification of constraints, either directly or by demonstration. The original one was Sketchpad ([44]). In Sketchpad when the user merges two objects of the same type, constraints on either of them are applied to the new merged part. It also lets the user display a graphical representation of constraints: they are shown as a circle containing a symbol representing the type of constraint.

Another such system that permits specification of constraints by the user is Juno ([33]). It also supports the direct application of graphical constraints. The user can select icons representing constraints which the system applies to points previously selected.

One of the newer systems is Grace ([3]). It is a graphical editor that lets users define graphical or geometric constraints. It provides mechanisms for constraint specification via simple means. These include simple direct-manipulation methods and a constraint-by-demonstration facility that incorporates both novel heuristics for inferring user-demonstrated relations and natural-language explanation tools that help the user understand the inferencing behavior of the system. This research was primarily focused on two ideas:

- investigating user-interaction mechanisms for conveniently specifying and obtaining information about relationships
- moving towards guidelines for inferring constraints from user-demonstrated examples like in Peridot

An explanation facility was built in for cases when the visual feedback might not be sufficiently informative. Users might not always understand the "behind-the-scenes" activity and exactly why certain constraints were inferred and others not. Grace pops up a separate window containing a natural language description of all the constraints it inferred or chose not to infer along with a justification for each decision.

As can be seen, the goal of the above research was primarily to construct a platform for the user to specify constraints in some "smart" way. A secondary goal was to maintain them. In some cases an explanation facility was provided. The problem with natural language explanation is that it can be very ambiguous if the problem is significantly big. Users do not want to read through pages of information describing actions that lead to a problem. (Even if they did, it is a very painful procedure to go through a long list of constraints that are affected by a single action). This was an important part of our motivation to construct a system where the main idea is visualizing constraints and domains. The user wants to actually *see* the consequences of his/her action. (S)he may also need guidelines for the next steps in the design. Letting the user experiment while designing and to have a system - provided guidelines was one of our aims.

Visualization itself is a broad area and can be described as the graphical description of a physical phenomena where the data itself need not be visual ([48]). In many applications the difficulty in dealing with large volumes of data led researchers to try known techniques in different contexts. Visualization also has drawbacks: it is very easy to get false impression by plotting data in a "pleasing" manner ([46]).

[49] also discusses some new and not-so new techniques for presenting data in some sort of visual manner, e.g. using image processing of non-image data. Colour spectrograms or pseudo-colour animation of selected parameters can help review inspection of large amounts of data in a very short time. Wolff states that visualization should not be viewed as the end result of a process of some scientific analysis, but rather as the process itself. It should be more than an application of a technique for displaying data. It can be viewed as

"a paradigm for exploring regions of untapped reservoirs of knowledge".

FLOWER helps the designer to take the next step in the design by showing the domains of design variables. The aim is not just simply show that domain but to suggest visually the next step to be taken.

In computer graphics systems there are two basic forms of visual design: drawing systems and modeled systems. With the drawing systems, one can sketch ideas into the computer using it as a sophisticated drawing board. The other alternative is to give the machine a mathematical model of objects the designer wishes to create and have the computer make images from that. The main idea of our new proposed system (FLOWER) goes beyond this division. Creating a system that can be used as a drawing board if the designer wishes to draw only, *and* to have a system that gives recommendations and supervises design decisions at the *same* time was our goal.

Since our approach is to visualize relations using colours, we reviewed papers on colour usage in user interfaces. Colours used in computer graphics are often selected in an ad-hoc fashion, without considering their physical and psychological effects. Problems arise from the fact that there are no established algorithms that can be applied to choosing colours, only heuristics. Many of the existing guidelines suggest using or not using a particular colour for a specific use, i.e. they are not general strategies or design guidelines for selecting colours. One approach to select appropriate, effective and tasteful colours for user interfaces is to use an expert system ([31]).

In the area of the use of colour in visualization, Rheingans and Tebbs in [39] visualize data by mapping the value of a variable to a colour value. Levkowitz and Hermann also used colour scales to display image data in [26]. This suggested to us that we should represent different types of constraints with a colour scale. The spatial constraints are colour coded: the closer a room should be, the darker the gray representing the constraint.

Frome in [16] contains some suggestions to consider when designing with colours: colour aftereffects should be avoided if possible, colour differences can be increased to make objects more visible, standard conventions should be followed. There may also be cultural differences in interpreting colours ([9]).

2.3 Intelligence Aids to the Design Process

In this section we review several papers more specifically relevant to portions of our work. The papers are sorted into two groups. First we describe systems that were fully automated. Then we turn to those systems where the user plays a role in the design.

2.3.1 Automated Design Systems

The work of Seligmann and Feiner on the use of expert system in designing illustrations ([42]) is discussed first. It shows that design is a goal-driven process within a system of constraints. When analyzing a partially completed design, their system backtracks for generating a better solution so previous mistakes or off-track solutions can be avoided.

Next we describe the work of Henry and Hudson ([18]) on using constraints in User Interface Management Systems. Their work on designing screen layouts is very similar to our work on placing rooms in a floor plan. Thus their ideas provided suggestions to our work.

The research of Hudson and Yeatts in [20] is presented next. They described a technique for integrating rule-based inference methods into a direct manipulation interface builder. Though they refer to the desirability of the designer control of the process, their system followed an automated approach.

Finally we present the work of Baykan and Fox in [5], [6] and [7]. They investigated constraint-directed heuristic search as means of performing design. Their application was very similar to ours: they were designing layouts of kitchens. They were also emphasizing on using constraints throughout the design process. They did not deal with feedback from the user: their designing system is also fully automated.

Expert Systems in Illustration Design. An *illustration* is a picture that is designed to portray meaning, i.e. meet some communicative *intent.* Seligmann and Feiner describe IBIS ([42]), a system for automated design of intent-based illustrations. Their design is a goal-driven process within a system of constraints, where the goal is to achieve the purpose and the constraints are the illustrative techniques an illustrator can apply.

The idea behind the system is to generate presentations, each designed to satisfy the same communicative intent for a particular audience such that the illustration has the *exact same meaning* to many different people. IBIS designs illustrations to fulfill a high-level description of the intent. The work described in this paper represented a new method for generating illustrations utilizing multi-level backtracking. Evaluators analyze partially completed designs. Based on the evaluations the system backtracks for generating a better solution. Illustration objects are generated based on both the representation of the physical object and the communicative intent. This way, the system also takes into consideration the physical properties of the object, not just its intent. The multi-level backtracking idea seems to be very useful, since it works like a developing design, which learns from previous mistakes or off-track solutions. However, the precoded evaluators do their work with no user input taken into consideration while the system is working.

Using Constraints in UIMS Design. If we think of placing windows on a screen as a task similar to placing rooms in a floor plan, certain aspects of UIMS work are applicable to our work. Henry and Hudson for example describe the *Apogee UIMS* in [18] which uses a unified data model (from [38]) to support a range of tasks. This active data model not only stores data, but also acts when changes occur in them and is based on incremental attribute evaluation concepts. In this UIMS, interfaces are treated as editors and browsers of data. Both the application and the user are given access to the data. Under this paradigm, the primary task of the user interface is one of translation, i.e. the user actions have to be translated into internal data and into actions within the application. Also, when the application changes data, these changes have to be translated into new graphical images, presented to the user. An active data model can be used to automate these translations.

At the lexical and syntactic levels, graphical presentations are defined in terms of attributes. Graphical images are updated automatically whenever the attributes which define them change value. This allows simple specification of dynamically changing layouts that can automatically adapt to make good use of available screen space.

At the semantic level or application interface level, the system allows important

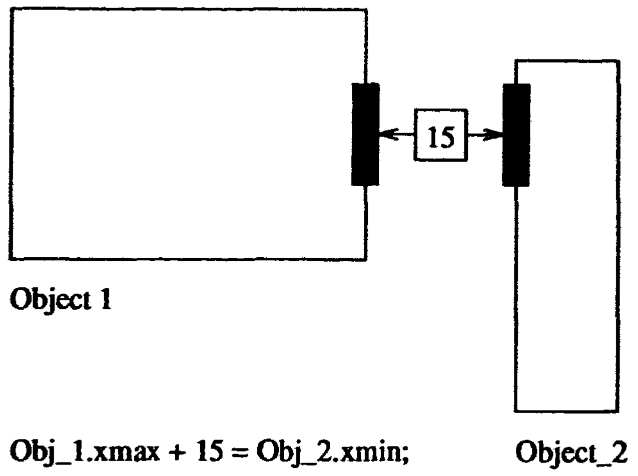


Figure 2.1: An Example of Apogee's Data Model

application entities to be equationally related to the overall system of attributes. This creates an automatic connection between changes in application entities and graphical representations on the screen (shown in Figure 2.1).

An object-oriented data model is used in the Apogee UIMS, which supports multiple inheritance for defining objects. Objects respond to a set of messages by invoking methods, but their internal structure and implementation are completely hidden.

This work was similar to the work of Zanden et.al. ([50]), in the sense that both authors were using constraints describing a set of dynamically changing relationships. Apogee allows constraints to directly reference objects but does not allow indirect references, as does Zanden's work.

Both methods deal primarily with implementation of constraints: their application is secondary. In FLOWER we want to use constraints without worrying how they were implemented. Many researchers have pointed out the importance of being able to use constraints in a design system (starting from [10] and [32] etc.).

As we suggested above, there are some analogies between an UIMS and an intelligent design system. Placing windows in the screen and placing rooms into a floor plan can be very similar. Thus some ideas in this article provide suggestions for FLOWER:

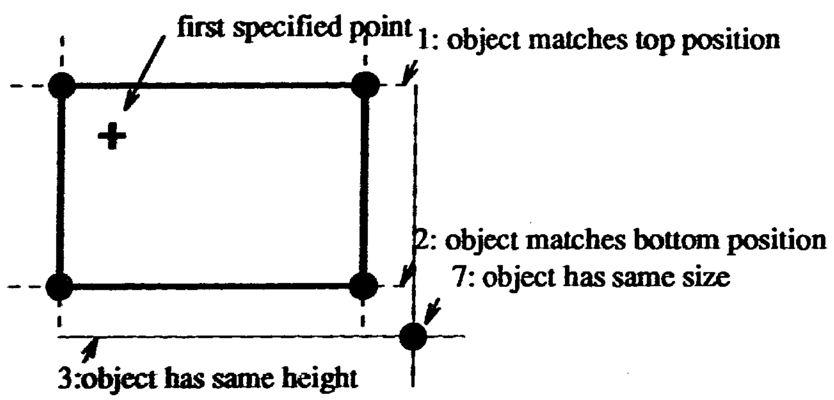


Figure 2.2: An Example of Inference Rules Expressed as Snap Sites

how to show "beside" constraints, how to show a fixed position according to some reference point, and how to show a variable size object.

Rule-Based Systems in Interface Design. Research in building user interfaces is going in two main directions. One set of systems (*interface builders*) provides environments or editors that allow an interface to be specified with direct manipulation. Others are highly automatic, constructing an interface with minimal (initial) user effort. Both directions have their own advantages and their drawbacks as well.

Hudson and Yeatts in [20] attempt to find a way to integrate these two approaches. They describe a technique for integrating rule-based inference methods into a direct manipulation interface builder. The results and effects of the rules are presented to the user. A direct feedback and control over the application rules are provided by semantic snapping ([19]) techniques.

Figure 2.2 shows an example of some of their inference rules and the expressed snap sites. The user is trying to place rectangular shapes to the work space. In this example, (s)he already placed the first object. When the user specifies the first corner of the next rectangle, the system considers the set of predefined inference rules and snaps the opposite corner to one of these.

Hudson and Yeatts wanted their interface builder to meet the following criteria:

- a visual notation for all aspects of user interface design
- direct expression of rule actions in the notation
- facilities for user control over inference
- support for a fairly wide range of inference rules.

The knowledge base of the interface builder holds at least one representation of the user interface being specified. All actions adding to or modifying the design are expressed as modifications to the original data structure. All actions by both the user and the inference engine are coded in the knowledge base. The visual notations of the various aspects of the interface specification are stored there as well. They provide the "visibility" and they are also the basis for both feedback and user control of the inferencing process.

The main technique introduced in this paper is the use of ***semantic snapping to portray actions*** in the knowledge base. Semantic snapping is an extension of the conventional gravity field technique. The decision to snap can be made on the basis of geometry and also on the basis of semantic tests carried out dynamically during dragging. Furthermore this semantic snapping can provide a visual feedback when snaps occur.

Experiments with a small prototype of the system proved that it is very useful when the actions and the consequences of inference rules are immediately apparent to the designer and he/she is provided with dynamic control of the rules as those rules are part of the interface specification process.

The problem with this work is that they tended to use the automated system as a substitute for the human designer. Although they mentioned that the designer has to have control, the described system works independently of the designer (except for the initial inputs). Their main scope was expressing knowledge about snapping; nothing is said about what it is possible to do when designing. The user gets no guidelines as to the possibilities for user interface design directions. FLOWER works together with the designer, it provides guidelines, explanation and suggestions on proceeding with the design.

Using Constraints for Space Planning. Baykan and Fox investigated constraint-directed heuristic search for space planning in [5], [6] and [7]. Space planning involves topological relations such as adjacency and geometrical properties such as shape, dimension, distance and other functions of spatial arrangements. They found that it is natural to express space planning problems in terms of constraints. Experience with space planning programs indicates that computing time was affected by the strengths of constraints and their sequencing. Constraint-directed search attempts to formulate general models for the representation of constraints. The objectives are to identify and represent a variety of constraints and interactions between them (such as conflict, competition and relaxation) for effective utilization during search.

They created *Wright*, a knowledge based design system that uses constraint-directed opportunistic search to generate layouts in different space planning domains. It consists of a knowledge base, a problem solver and a user interface. The knowledge base contains knowledge about the application domain. Wright designs kitchen layouts, thus knowledge is expressed about possible items in a kitchen and their relations. For example, a kitchen can have sink, oven, counter, etc. Counter space has size requirements, refrigerators should not be beside ovens, etc. The problem solver focuses on the different aspects of space planning such as locating, dimensioning etc. based on uncertainty measures associated with constraints.

Layouts are created by configurations of design units. Design units are considered at different levels of detail. The design units form a hierarchy through which there is inheritance of variables, values and constraints.

The highest level of abstraction for representing design states is the spatial level (inside, contains, no overlap etc.). The next level uses one-dimensional relations (region-west-of, horizontally overlapping, etc.). The lowest level of abstraction is the region-line adjacency network (a representation for generating layouts using rectangular regions and horizontal and vertical lines).

It is possible to design at each level of design representations. The goal tree controls the focus of attention on the levels and representations and facilitates switching between them. The goal tree is a hierarchy of goals and constraints starting with general goals representing knowledge of the design domain.

The first stage of problem solving is pre-search analysis based on the initial (given) constraints. The second stage is the opportunistic search. Constraints are selected based on their uncertainties. Information for determining the uncertainties of constraints are: importance of and severity of constraints and the size of design unit affected by the constraint. Uncertainty is used as a measure for rating opportunism of constraints and determining where to focus attention during search.

The system presented by Baykan and Fox is similar to FLOWER in a sense that both design simple layouts. Both systems use constraints to express design goals and knowledge. However, Wright is yet another automated design system where the user provides initial input and the system produces an "answer". There is no interaction between Wright and the user during the design process. Wright was created to be a fast designer where speed was gained through the way it handles the constraint satisfaction problem. Their goal was to produce a system using dependency-directed backtracking. We were fortunate to have a system where this dependency-directed backtracking already in place. Our task then was to develop a means for adding/integrating user interaction.

2.3.2 Interactive Design Systems

We describe three papers in this section. Kochar in [25] is presented first. His work was closely related to ours as he is intended to provide help throughout a design process.

We then turn to the work of Kamada et.al. on visualization of abstract data in [45]. Their work is important for the use of feedback from the user throughout the design process.

Finally, we present the work of Dill et.al ([11] and [13]) on intelligent computer aided design. This work is the predecessor of FLOWER of our research group.

Supporting the User by Presenting Design Alternatives. The activity of design can often be characterized by a search; in other words the designer examines various alternatives at several stages during the design. The problem with current CAD systems that they either expect the designer to have a complete design and just use the system as an intelligent drawing board. In these systems, obviously all choices are made by the user so the system isn't really assisting with the design at all. Other systems generate (sometimes fairly large numbers of) design alternatives which are presented to the user normally one after the other in a sequential manner. The user must then determine on his/her own which one is the best.

The approach taken by Kochar ([25]) to this problem is among all the works reviewed, the most closely related to our own. Kochar's system, *FLATS* is a prototype for design automation via browsing and was constructed to demonstrate the paradigm of cooperation between the user and the computer in CAD applied to the design of small architectural floor plans. The system supports the exploratory aspect of design. A structuring mechanism helps the user explore design alternatives in a systematic way, by varying those properties of the design that are of primary interest.

Again, there is a major problem with this system. Although it does more than just describe in text a set of affected design constraints, it does bombard the user with possible solutions at certain stages of the design, which tends to be overwhelming to the user. Most designers insist on playing an active role; they do not want to be passive bystanders, selecting from a menu of generated designs.

This system is a positive step of course toward presenting design alternatives to a user; still it does little more than the previously described systems, except for generating a set of partial designs as opposed to presenting a complete design. In other words, it still automates the design process, only it does so to a portion of a design. These partial designs are generated based on precoded knowledge, therefore the designer cannot contribute to the design by adding new knowledge once the system started to work.

Integrating the User into the Design Process. Interactive graphical user interfaces based on *direct manipulation* are well established. To ease the burden of the high cost of their creation, User Interface Management Systems (UIMS) are used. However current user interfaces usually consider only the interaction architecture and lack support for a consistent framework that allow visualization and manipulation of high-level abstract data, i.e. the semantics of applications. Kamada et.al. have been doing work (e.g. [21] and [24]) on visualization of abstract application data, i.e. translation of abstract data into pictorial form. They have also extended their one-way visualization framework to bi-directional translation between the data representation of an application and the pictorial representations of the user interface in [45].

To lessen the need for continuously varying the mapping rules between the infinite number of possible representations, they used two intermediate, universal representations, and developed a set of rules for mapping one to the other. The intermediate representations can be left unchanged, even if the application changes. The mapping process handles the following representations:

- Application's Data Representation (AR) -

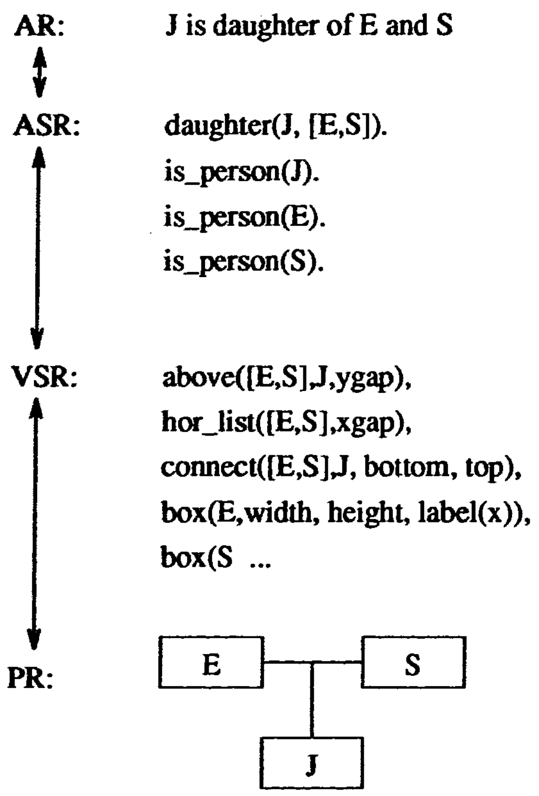
  this is application specific, and can be any kind, e.g. natural language, program listing, data in a database

- Abstract Structure Representation (ASR) -

  this represents the underlying abstract structure, i.e. a set of *relations* among *abstract data*; AR is translated into this form and vice versa

- Visual Structure Representation (VSR) -

  this is the underlying structure of a picture, i.e. a set of *graphical relations* among *graphical objects*;



```
AR:     J is daughter of E and S

ASR:    daughter(J, [E,S]).
        is_person(J).
        is_person(E).
        is_person(S).

VSR:    above([E,S],J,ygap),
        hor_list([E,S],xgap),
        connect([E,S],J, bottom, top),
        box(E,width, height, label(x)),
        box(S ...

PR:
```

Figure 2.3: An Example of Kamada's Model

- Pictorial Representation (PR) -

  this is the representation of the picture to be rendered directly on display devices.

First AR is analyzed, and then ASR data are generated from it. Then the ***visual mapping*** is done, i.e. ASR is translated into VSR. (The mapping from VSR into ASR is called ***inverse visual mapping***). Finally VSR is translated into the target PR. The graphical relation data are first translated into geometrical constraints among ***picture objects***. To determine the actual positions in the display-space, the constraints are solved by constraint solvers.

$$\text{AR} \longleftrightarrow \text{ASR} \longleftrightarrow \text{VSR} \longleftrightarrow \text{PR}$$

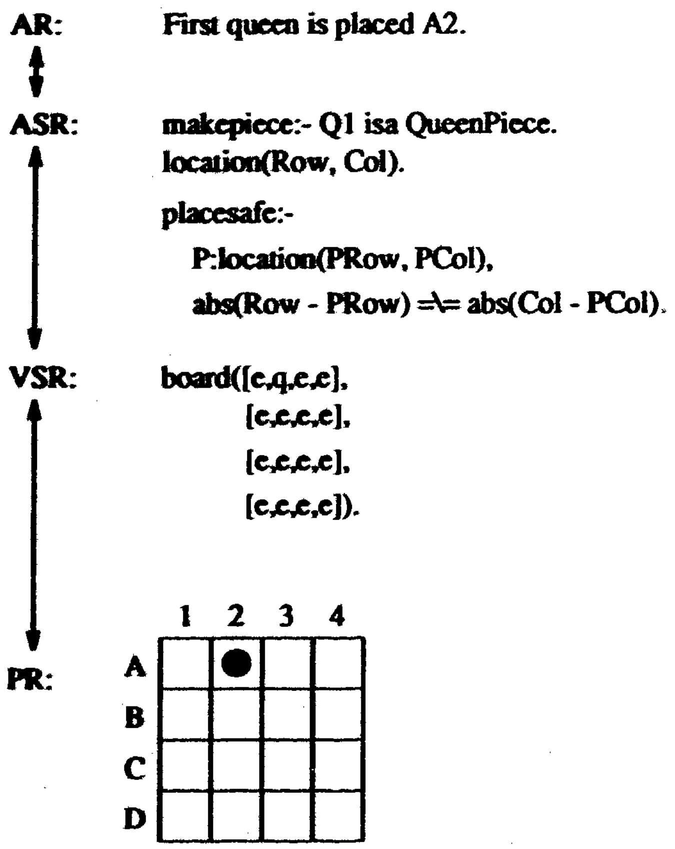
Figure 2.3 illustrates an example of the four types of representations. In this example, the application's data is represented as a natural language sentence "J is daughter of E and S". The corresponding ASR data are *daughter*$(J, [E, S])$, *is_person*(J), etc. The ***daughter*** is an abstract relation and the persons are abstract objects. The corresponding VSR data are *above*$([E, S], J)$, etc. The *above* etc. are graphical relations and the *boxes* are graphical objects. As a pictorial representation, a family tree picture is generated in Figure 2.3.

Similar work has been done in SFU's Intelligent Systems Laboratory in the form of the "nchess" program. Here, after a user places a chess piece, nchess determines the location of the remaining pieces. In this application the chess pieces say queens and the board form the pictorial representation. The VSR data is the *board()* The *board()* describe graphical relations between graphical objects: it describes whether the boxes representing the chessboard should be drawn empty (e) or filled with a placed queen (q). The ASR data are the *makePiece:- Q1 isa QueenPiece, etc.* The *placesafe* is an abstract relation between the abstract objects describing the constraints for safe placement. Figure 2.4 illustrates this example. When the player places a queen on the board PR is modified accordingly. The modified picture is translated then into the board data. The reasoning engine evaluates the placement and the resultant ASR of the board is again visualized to update PR.

Our FLOWER uses the same set of mappings. However, while Kamada's group had to put a significant effort into developing this communication path, it was directly available to us via the external object protocol access to Echidna. In FLOWER, the Echidna knowledge base corresponds to ASR, while the knowledge base/database update routines correspond to VSR. Communication between ASR and VSR structures is facilitated through a module called mediator. Further details are provided in Chapter 4.

We believe that Kamada's work represents an important contribution to the area, since it is one of the very few that acknowledge that user and system must work together: that the user must not be left out of the design process.

Indeed they stated that there wasn't any other work which directly influenced theirs. The same thing is true for the constraint visualization. The problem is that



```
AR:      First queen is placed A2.

ASR:     makepiece:- Q1 isa QueenPiece.
         location(Row, Col).
         placesafe:-
            P:location(PRow, PCol),
            abs(Row - PRow) =\= abs(Col - PCol).

VSR:     board([e,q,e,e],
               [e,e,e,e],
               [e,e,e,e],
               [e,e,e,e]).

PR:
```

Figure 2.4: The Nqueens Problem Described in Kamada's Terms

most systems are limited to just representing constraints and manipulating them. For instance, in Kamada's example of the Othello game, the system does not suggest next steps, nor indicate what choices might be available. We believe such capabilities are desirable and have attempted to provide them with our constraint and domain visualization techniques.

An Intelligent Basis for Design. Our research group at Simon Fraser University started to work on an intelligent CAD project ([11]) with similar objectives as those of FLOWER. First, a protocol analysis was conducted to identify problems and difficulties of the design process. The results of the study was incorporated in the next step, where a system was created to help with home design. The first design task addressed was that of foundation design ([13]). Given a partial design, defined by AutoCAD drawings, the system works with the user to generate joist and beam layout. The system supports the mixed-initiative paradigm, more on initiative from the system. It can generate the entire layout or the user may interrupt the system at any point to specify design constraints.

This system is a predecessor of FLOWER: it has similar architecture and goals. However, FLOWER allows more initiative of the user while giving guidance on completing a design. The system of [13] does not advise on design steps, simply incorporates the designer wishes to the final solution.

CHAPTER 3

FLOWER - Scope and Functionality

3.1 Problem Definition

Designing the layout of a building can be a tedious task even for an experienced designer, when considering many initial requirements. Some routine tasks can be done easily and some of them could be automated and solved by a system. Rules considering building codes and physical laws can be coded and represented in the knowledge base.

In addition to the objective aspects of the design, there are subjective aspects, such as aesthetic qualities. Obtaining such knowledge is difficult; encoding it is more so. This seems to provide even more reason for retaining the designer as an integral part of the design loop.

We think that a system which uses some guidance from a designer but remains independent enough to make some decisions is very much needed. It is important to allow the designer to have control over the design process while it is important to be able to automate as many tedious or repetitive steps as possible and to provide guidance or direction when it is needed. With this motivation, the following objectives were set:

1. to address a major limitation of current CAD technology: to provide interactive, intelligent design assistance but not automated design
2. test the proposed constraint-based, mixed-initiative designer system
3. test how a visual aid (i.e. showing the domains of some design variables and/or constraints between them) can help the designer to accomplish a design
4. gain further insight into the use of the Echidna Expert System for designing purposes
5. create a potentially useful tool for simple layout design tasks.

3.2 Solution Methodology

FLOWER is a design system where user and system can work together as equal partners. The overall design scenario is as follows:

1. The user starts up the system and begins designing a floor plan. (See detailed description of usage in Appendix). (S)he can move rooms around, experimenting with the layout, then finalize their position.
2. The system supervises design decisions to check that *physical laws* are obeyed. The user may not override these relationships. It does however show the available spaces for each room. The user can can try whether a particular placement of a room is valid and see the system's reaction while proceeding toward a complete design.

As an example, it is physically impossible for rooms to overlap. Thus at any stage, when the designer wishes to add a new room, the system will determine the valid space for that room. The system acknowledges a valid placement and the design session can be continued; however, the user cannot continue upon invalid placement, unless (s)he removes the offending room.

3. Additional spatial relationships between existing rooms can be added. FLOWER then checks and notifies the user about problems ›ut lets the user proceed even with "mistakes", The offending rooms will be marked. When the system indicates a problem, it also gives a visual clue about what went wrong with placing that room and another visual clue about fixing the problem.

   For example, the user wants to place a bedroom and a bathroom, requiring them to be *close* to each other. After the user specifies this relationship and places the first room, the system will show possible locations for the second room that fulfill all constraints. (In this case the set of constraints is: the "close" relationship; "rooms must not overlap" and "rooms must be inside of house" physical laws. The system evaluates the placement of the second room and if valid, the design session can be continued; while a misplaced room will be marked but the system takes no corrective action. The system will "explain" though what was wrong with the placement and suggest a corrective action.

   By distinguishing between required and desired constraints, the system allows the user more freedom while making the design decisions. If all constraints had been considered serious, the designer could only make proper design steps. Otherwise the system would reject all steps that did not meet the requirements. This way however, the designer has more liberty to try out ideas and return later to this problems.

   For example, (s)he might say: I want the kitchen *beside* the bathroom but I do not know yet exactly where; I will put it somewhere close, for now. FLOWER will not let the mistake go unnoticed. It will flag the user and suggest corrective action. It will not let the user leave the problem uncorrected indefinitely. However, (s)he may work on other rooms first and later return to correct this problem.

3.3 User's View of the System

The user sees this system as a design partner. The system does not proceed in an automated fashion, leaving the user out of the design decisions, nor does the user complete a design alone only to find out in the end that it does not meet the original requirements. The user can think of the system in the following ways:

- helper

  The system will *suggest* steps for the designer throughout the design process. As an example, it will show the available spaces for a new room based on the entire current constraint set.

- strong critic

  FLOWER also *checks* for serious mistakes during the design. In this case, the system *does not let the user continue* while the problem remains. For example, when a user places a room outside of the predefined house, the user is not able to continue, unless (s)he removes that room.

- soft-hearted critic - teacher

  The system *simply notes* the problem, *explaining* what it was and *suggests* corrective steps.

3.4 Visualization of Constraints

The most important part of our work - in addition to creating a very simple design system - is that FLOWER actually helps the user in the process of design. This help is threefold:

- after the designer selects a room, the system shows all acceptable areas of the placement of that room
- the system explains whether a certain step in the design was successful or not

  a) if a constraint representing a physical law was violated, the system does not allow the placement of any new room, until the problem is corrected

  b) if a constraint representing a user preference was violated, then the placement is allowed but the violation is shown and the offending relationship is indicated
- it also suggests to the designer how to proceed when encountering a design step that contradicts previously specified relationships the user preferred to be held.

The help that FLOWER provides is based on *visual clues*, using colour. Each type of constraint is represented with its own distinct colour. A part of the user interface shows them, to help the user remember the meaning of the colours. The user interface has an array of buttons, one for each room type, with a different colour for each room. Rooms are given the same colour as their buttons.

The designer's task is to place rooms in the floor plan. When a room is selected, the designer can move that room around the work area by the mouse. While trying to select an acceptable place for that room the room is shown in its assigned colour.

The first visual clue to the designer is the valid area of the available space for the placement of that room, shaded to correspond to the colour of the room. To distinguish the representation of the room, the shaded area uses a lower saturation of the same hue. For example, if the colour of the chosen room is red, the shaded area will be pinkish. Figure 3.1 shows an example of such placement. Here, the designer has already placed two bedrooms (shown yellow) and a bathroom (shown brown).

If the designer specified preferences (such as a *beside* constraint) involving the room to be placed, then the shaded area for the room placement will be shown in the colour of that constraint. Figure 3.2 shows an example of that. Suppose the designer

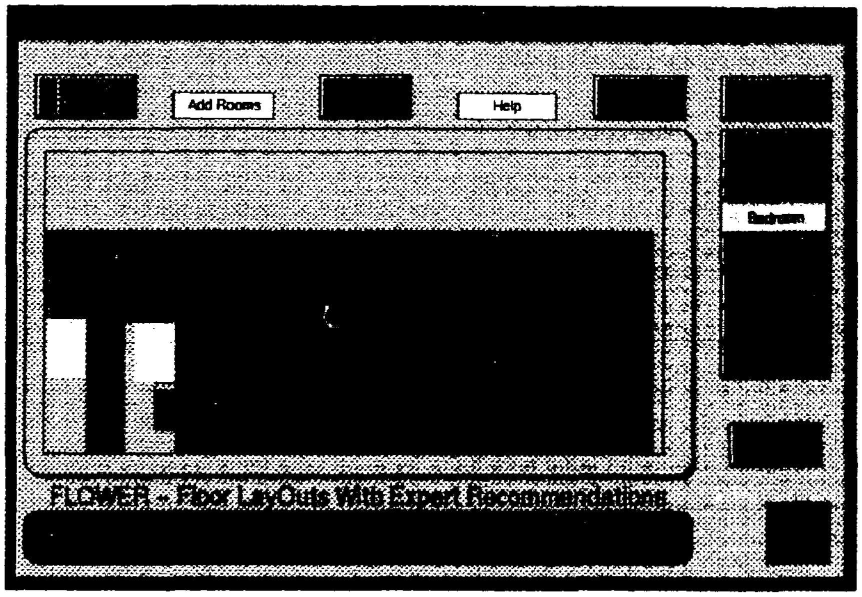


Figure 3.1: Visualization Aids for Placing a Hallway, with No User-Set Constraints

did not place the hallway earlier, but set a constraint of *hallway beside bedroom.* Now the visualization shows that the hallway to be placed must go beside both bedrooms but it still cannot overlap the bathroom. The colour of the shown area is black now, showing the beside relation that must be met.

If the designer ignores the clue, and places a room in an unshaded and hence unacceptable area, the system will respond immediately. The problem will be evaluated and if the placement of that room violated a physical law then the user must remove that room from the floor plan in order to continue with the design. If the room violated a user-specified relationship then the system will indicate that to the user. The appearance of the room will be changed as follows:

- the room is shown outlined only: indicating a problem
- a line will be shown between the offending rooms; the colour of that line will be the same as the colour of the offending constraint, thus explaining what went wrong; if more than one constraint is violated, additional lines will be shown

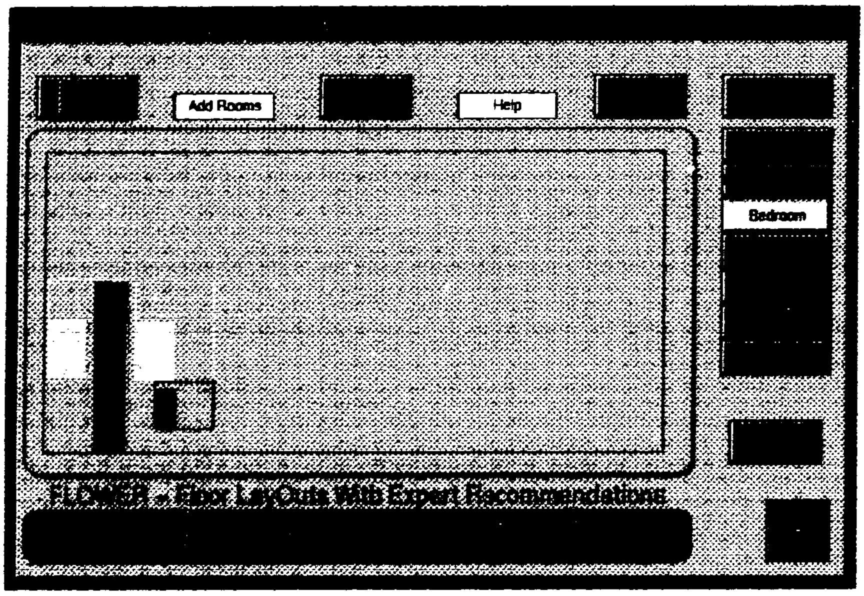


Figure 3.2: Visualization to Place a Hallway, with User-Specified *Beside* Constraint

- there will also be an indication for proper placement of that room in the form of two arrows below the constraint indicator line; the arrows suggest whether the designer should move the room closer (><) or further (<>) away; the arrows will only appear if the correction is possible (if the user specifies contradictory constraints, there is no way to satisfy them, and no arrow appears)

If a room is in several different type of relationships to the existing rooms, e.g. it has to be beside some but far away from others, the available area will still be calculated, but the colour of it will be specific to the room to be placed and not to the constraints. Again, the area will be shown with lower saturation of the room colour.

For example, the designer wants to place a master bedroom now. (S)he set the following constraints: *master bedroom beside bedroom* and *master bedroom close bathroom*. Now, the system will show the available area in light green (the original colour of the master bedroom is green) as this room is in two different relationships. Again, the beside constraint must be met for both bedrooms.

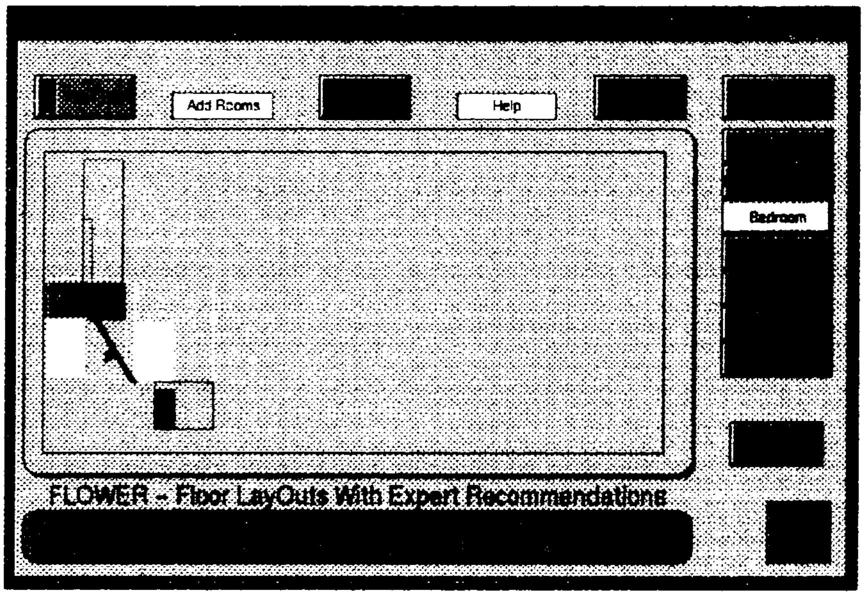


Figure 3.3: Visualization Aids for Placing a Master Bedroom with Several User-Specified Constraint

Figure 3.3 shows the available area for that master bedroom. In this figure we also can see that the user previously tried to place a hallway but ignored the visualization. As a result, that hallway did not meet the user-specified requirements. As we explained above, an explanation of the failure is shown now.

If the user ignores the visualization again and places the masterbedroom improperly, the failure will be presented. This is shown in Figure 3.4. As can be seen from the figure, the lines and arrows correspond to the colour of the failed constraints; thus they are the same colour for both bedrooms (as they both were in the *beside* constraint) but different for the bathroom (as it was in a *close* constraint).

If the user attempts to place that master bedroom again but now accidently places it such that it overlaps the bedrooms, the placement will fail again but this time there will be no explanation why the placement failed. Here too, the room will be shown outlined, but the outline will be thick (shown in Figure 3.5), clearly distinguishable from the previous case. Now, the user must remove this room in order to continue.

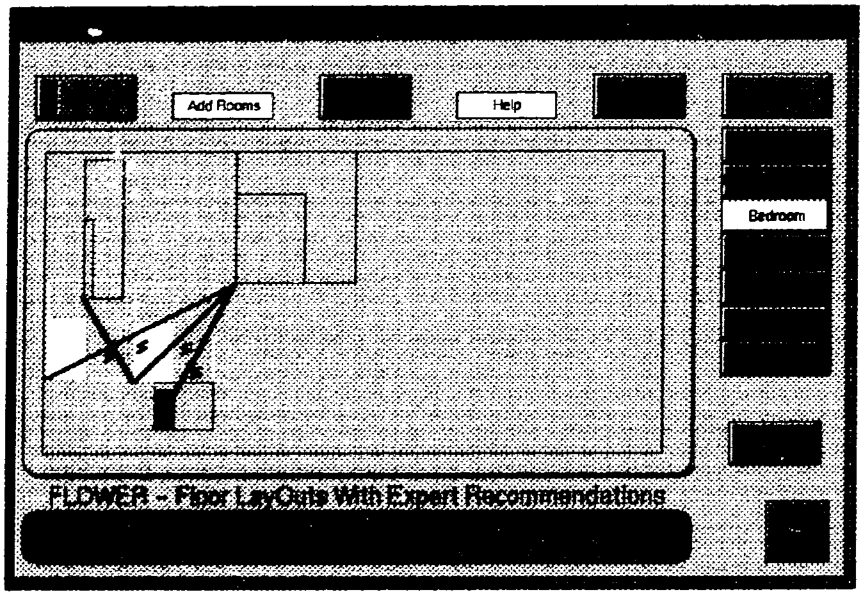


Figure 3.4: Placement of Master Bedroom has Failed

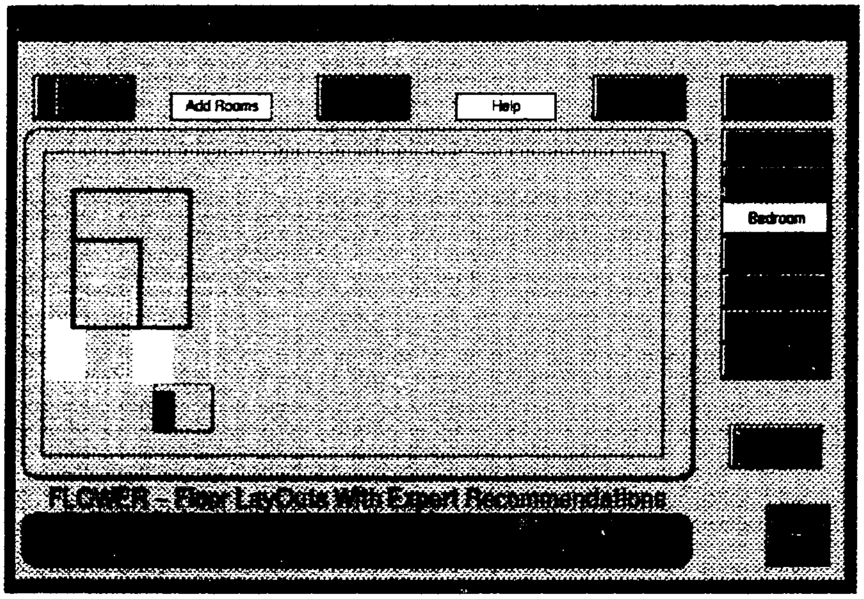


Figure 3.5: Placement of Master Bedroom Fails Due to a *No_Overlap* Constraint

CHAPTER 4

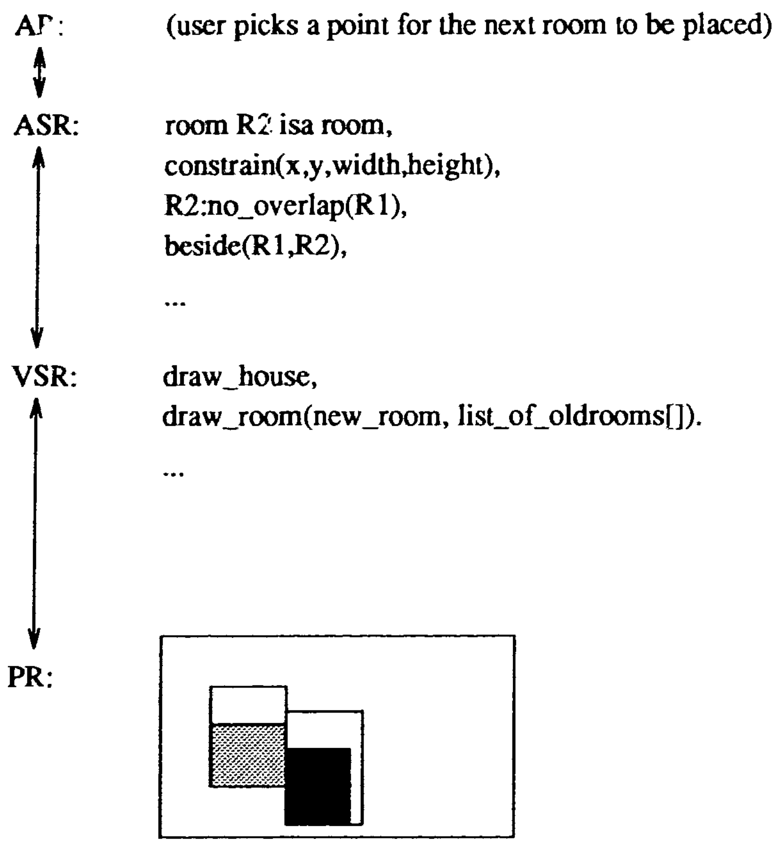
System Architecture

Many researchers (e.g. [5], [11], [13], [18], [25]) suggest that constraints are a natural way to express design goals. However, the lack of readily available constraint processing systems generally resulted in work focused on implementation of those constraints. Kamada et.al in [45] emphasized the importance of a bi-directional translation between the data representation of an application and the pictorial representations of the user interface as a way to involve the user in the design process.

In our case, both the constraint processing system and a described bi-directional translation were already available, allowing us to concentrate on further steps. We wanted to create a truly *mixed-initiative* system. In FLOWER, the system can

1. suggest design steps by displaying the domain of design variables
2. evaluate design steps
3. explain incorrect steps by visual display of constraints
4. suggest corrections.

The structure of FLOWER can also be described in Kamada's terms (Figure 4.1). In FLOWER, rooms form the Pictorial Representation. The VSR data are *draw_house*



```
AR:     (user picks a point for the next room to be placed)

ASR:    room R2 isa room,
        constrain(x,y,width,height),
        R2:no_overlap(R1),
        beside(R1,R2),
        ...

VSR:    draw_house,
        draw_room(new_room, list_of_oldrooms[]).
        ...

PR:
```

Figure 4.1: Structure of FLOWER Described in Kamada's Terms

and *draw_room(new_room, list_of_oldrooms[])* etc. The *draw_room()* describe graphical relations between the *room* graphical objects as it incorporates the representation of minimum and maximum sizes and colour information. The ASR data are the room instantiations (*room R2 isa room*), and the constraints, physical (*R2:no_overlap(R1)* and user-specified (*beside(bedroom, bathroom)*). For example, *no_overlap* is an abstract relation between the abstract objects $R1$ and $R2$. AR corresponds to the user's request for placement of the next room by means of an input device (mouse).

Again, while Kamada's group had to put a significant effort in developing the communication channel, in our case, the mapping between ASR and VSR was already available through our reasoning engine's external object protocol. In our case both AR and PR reside in the graphics side of our application. When the user indicates the placement of a room, AR is recognized by the graphics modules and the mapping between AR and ASR is done through knowledge base update routines and the mediator code. The knowledge base update routines are responsible for the application dependent part and the mediator code is responsible for the technicality of the link. (Sections 4.3 and 4.4 give more details.) ASR is represented in the knowledge base and becomes accessible through this link. VSR is represented by the graphics database. When our reasoning engine evaluates the user's design goal, the result is sent back to the graphics code again through the mediator and the graphics database update routines, thus implementing the mapping from ASR to VSR. Then the updated graphics database is mapped back to the graphics module, (VSR to PR) where the picture of the room is created.

Figure 4.2 shows a block diagram of our system, the components of which are described in detail in the following sections.

4.1 The Knowledge Base

Keeping in mind that we tried to implement a simple layout planner, the following were considered as a set of possible design rules:

1. houses are rectangular
2. rooms are the smallest element of the design, i.e walls, doors, etc. are not considered
3. rooms are rectangular, and their edges are parallel to those of the house
4. size constraints: rooms have a minimum and maximum size
5. restrictive constraints: rooms must be *inside* the house, i.e. the house is a limiting perimeter for their placement
6. topological constraints, i.e. placement of rooms with respect to each other:

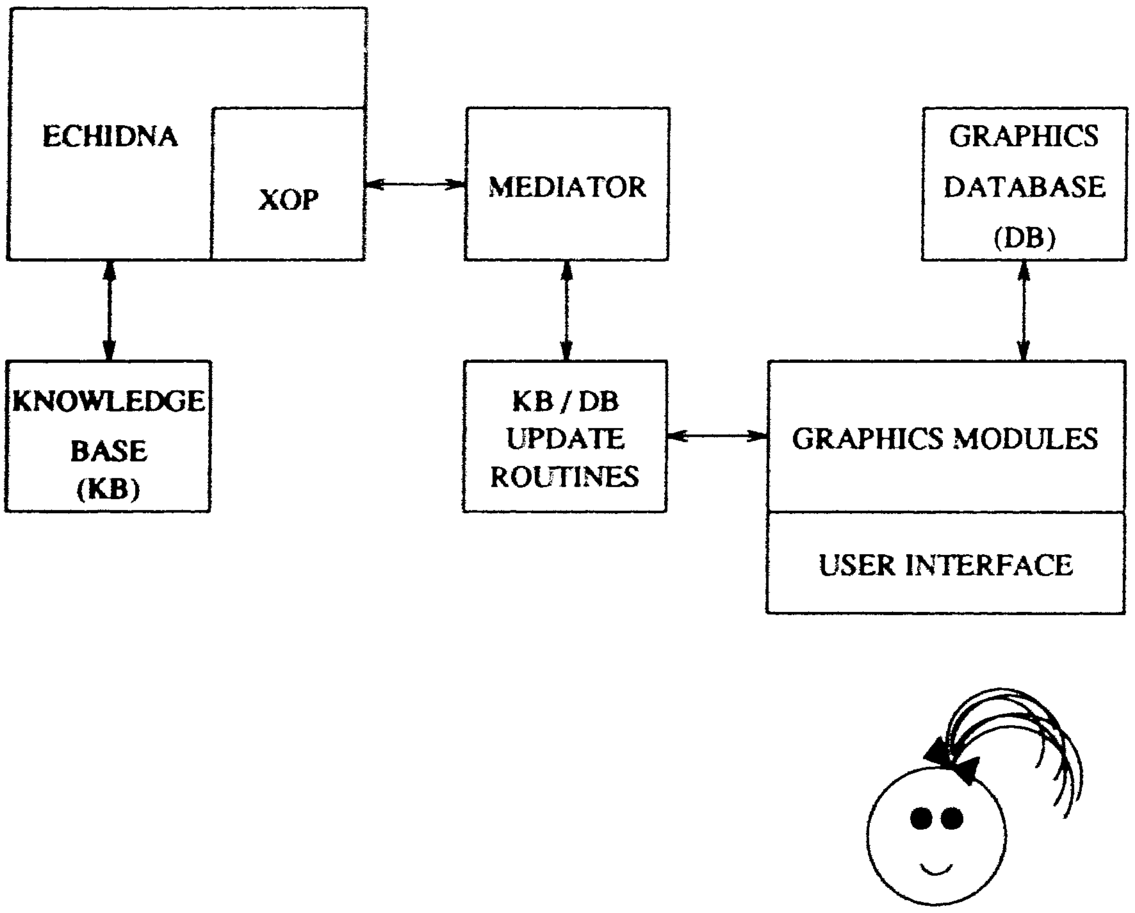


Figure 4.2: System Architecture

a) rooms cannot be on the top of each other

b) a given type of room can be beside or not beside, close, near or far from another specific room type

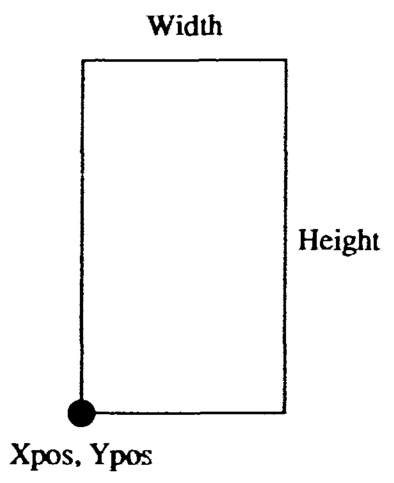
7. functional constraints: rooms may be associated with common functions such as food, sleeping, or baby-space.
8. practicality constraints: access to open air, daylight, airflow, etc.
9. interconnection constraints: hallways or stairways - to connect other rooms
10. accessibility constraints: placement of windows, doors or closets
11. aesthetic constraints

We implemented the first seven rules for this version of FLOWER.

The knowledge base consists of the appropriate schemas. A schema is the unit for representing objects and relations. *Rooms* (shown in Figure 4.3) are represented by a schema with its variables (see design rule 2). Currently rooms are rectangular with a specified minimum and maximum height and width for each room type. In this implementation, we used integer domains for all variables. Rooms are described by their lower left corner (Xpos, Ypos). (See Section 6.1.1 for a discussion on this restriction.) We constrain the Xpos and Ypos variables to be bound. Rooms have minimum and maximum width and height (see design rule 4) by also constraining the domains of the corresponding variables.

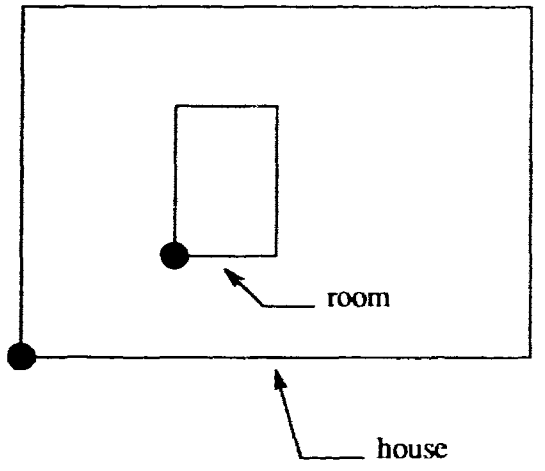
Rooms are gathered in a house which acts as a bounding box (Figure 4.4) and rooms cannot overlap each other Rules 5 and 6 a); these are basic physical relationships and hence cannot be ignored even temporarily by the designer.

Other relationships between rooms such as *beside* or *not beside* constraints (design rule 6 b)) may be specified by the user to be applied by the system. As an example, the designer may want to have the kitchen *beside* the dining room or the kitchen



```
schema room
{
  xcoord Xpos.
  ycoord Ypos.
  roomwidth Width.
  roomheight Height.
  roomtype Type.

% accessors
  all(Xpos, Ypos, Height, Width).
  type(Type).

%constraints
        % size setting
  constrain_x().
  constrain_y().
  constrain_w().
  constrain_h().

        % physical laws
  house().
  no_overlap().

        % designer rules
  beside().
  close().
  near().
  far().
  not_beside().

}
```

Figure 4.3: Room Schema



```
house(xcoord X, housewidth W,
      ycoord Y, househeight H):-

      Xpos >= X,
      Ypos >= Y,
      Xpos + Width =< X + W,
      Ypos + Height =< Y + H.
```

Figure 4.4: Rooms Should Be Inside the House (i.e. room at (Xpos, Ypos) with size (Width, Height) must fit in house of size (W, H) at (X, Y)

Constraints	Meaning	Illustrated on
beside	$d = 0$	Figure 4.6
close	$1 \leq d < 2Dim$	Figure 4.7
near	$2Dim \leq d < 4Dim$	Figure 4.8
far	$d \geq 4Dim$	Figure 4.9
not beside	$d > 0$	Figure 4.10

d: vertical or horizontal distance between rooms;
Dim: width or length of later placed room, depending on d.

Table 4.1: Table of Preference Constraints

should be *far* from the bedroom. Figure 4.5 shows the spatial interpretation of design rule 6 b).

The design rules were expressed in the knowledge base as well. E.g. when two rooms are beside each other, they could be beside either from the left, right, above or below.

Table 4.1 briefly describes the meaning of the topological constraints. The illustrating figures show an example of the corresponding code from the knowledge base.

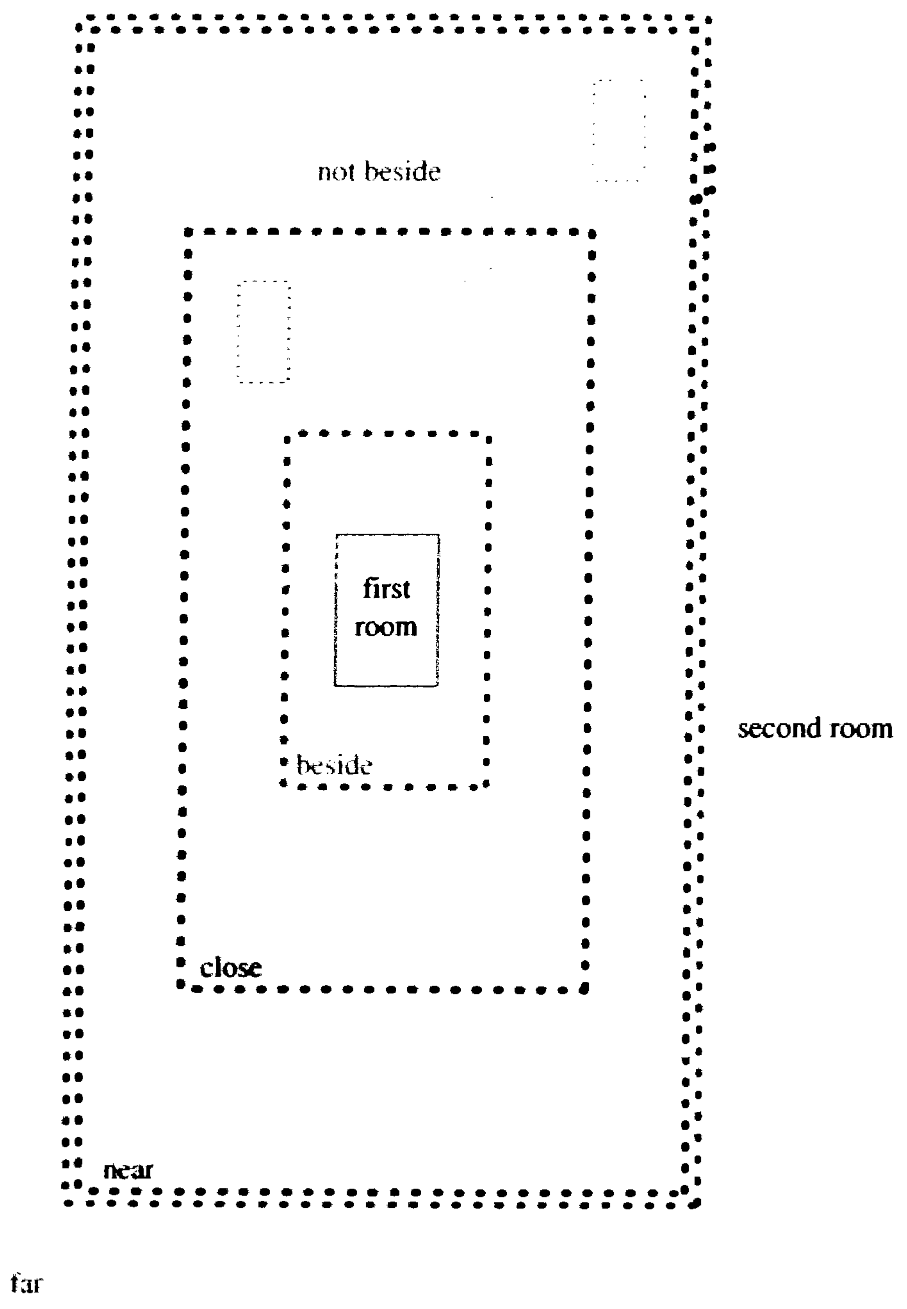
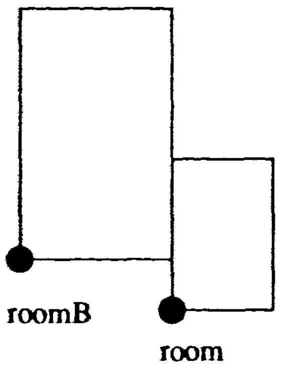


Figure 4.5: Preference Rules (dotted outline of a second room shows its placement with respect to existing (already placed) first room, to meet various constraints)

```
% above

    beside(room Rb):-
      Rb: all(xcoord Rbxpos,
              ycoord Rbypos,
              roomheight Rbheight,
              roomwidth Rbwidth),
    Rbypos =:= Ypos + Height,
    Rbxpos > Xpos - Rbwidth,
    Rbxpos < Xpos + Width.
```

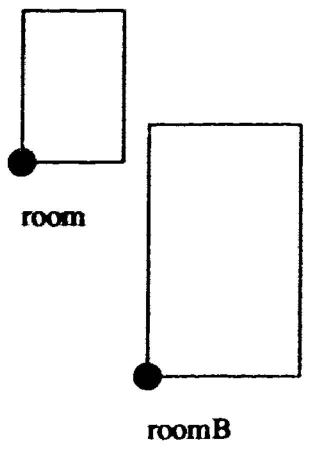
```
%left
    beside(room Rb):-
      Rb: all(xcoord Rbxpos,
              ycoord Rbypos,
              roomheight Rbheight,
              roomwidth Rbwidth),

    Rbxpos + Rbwidth =:= Xpos,
    Rbypos > Ypos - Rbheight,
    Rbypos < Ypos + Height.
```



```
%right
    beside(room Rb):-
      Rb: all(xcoord Rbxpos,
              ycoord Rbypos,
              roomheight Rbheight,
              roomwidth Rbwidth),
    Rbxpos  =:= Xpos + Width,
    Rbypos > Ypos - Rbheight,
    Rbypos < Ypos + Height.
```

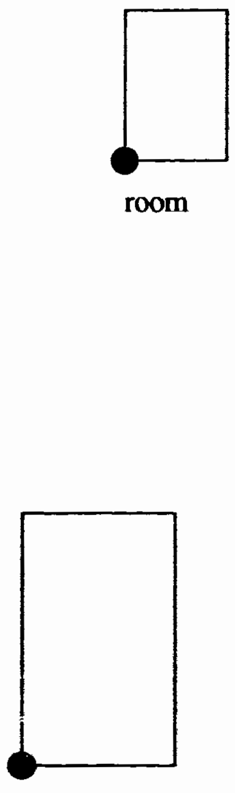
```
% below

    beside(room Rb):-
      Rb: all(xcoord Rbxpos,
              ycoord Rbypos,
              roomheight Rbheight,
              roomwidth Rbwidth),
    Rbypos + Rbheight =:= Ypos,
    Rbxpos > Xpos - Rbwidth,
    Rbxpos < Xpos + Width.
```

Figure 4.6: Beside Constraints: "room" should be beside existing "room B"



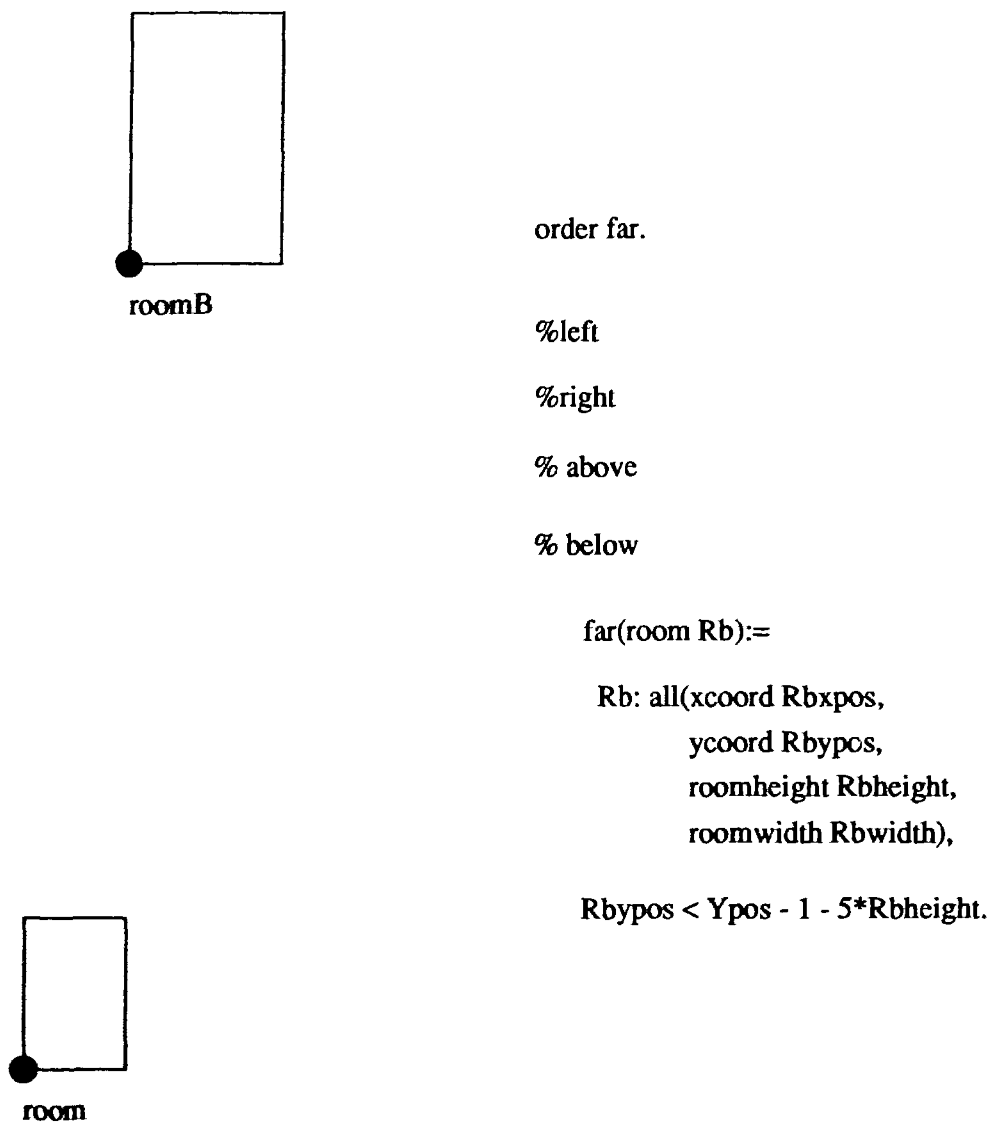
```
order close.

%left
    close(room Rb):-

      Rb: all(xcoord Rbxpos,
              ycoord Rbypos,
              roomheight Rbheight,
              roomwidth Rbwidth),

    Xpos - 1 - 2*Rbwidth =< Rbxpos,
    Rbxpos =< Xpos - 1 - Rbwidthh,
    Rbypos >= Ypos -1 - 2*Rbheight,
    Rbypos =< Ypos + Height + 1 + Rbheight.

%right

% above

% below
```

Figure 4.7: Close Constraints: "room" should be close to existing "room B"



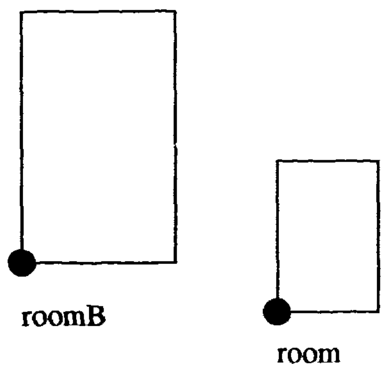
```
order near.

%left

%right

% above

   near(room Rb):-

     Rb: all(xcoord Rbxpos,
             ycoord Rbypos,
             roomheight Rbheight,
             roomwidth Rbwidth),

   Ypos + Height + 1 + 2*Rbheight < Rbypos,
   Rbypos < Ypos + Height + 1 + 3*Rbheight,
   Rbxpos > Xpos - 1 - 4*Rbwidth,
   Rbxpos < Xpos + Width + 1 + 3*Rbwidth.

% below
```

Figure 4.8: Near Constraints: "room" should be near to existing "room B"



```
order far.

%left

%right

% above

% below

    far(room Rb):=

     Rb: all(xcoord Rbxpos,
             ycoord Rbypos,
             roomheight Rbheight,
             roomwidth Rbwidth),

    Rbypos < Ypos - 1 - 5*Rbheight.
```

Figure 4.9: Far Constraints: "room" should be far from existing "room B"



```
not_beside(room Rb):-

    Rb: all(xcoord Rbxpos,
            ycoord Rbypos,
            roomheight Rbheight,
            roomwidth Rbwidth),

% left

    Rbxpos + Rbwidth =\= Xpos
    or
    (Rbxpos + Rbwidth =:= Xpos
        and
    (Rbypos =< Ypos - Rbheight
            or
    Rbypos >= Ypos + Height)),

%right
    ...
%above
    ...
%below
    ...
    .
```

Figure 4.10: Not_Beside Constraints: "room" should not be beside existing "room B"

At this implementation hallways (see design rule 9) are treated as regular rooms, so the designer has to place them along with the others. Also we didn't deal with design rules 10, 8 and 11 at this time (see chapter 6.3 for future work).

Rooms with functional constraints (design rule 7) must satisfy a beside or close relationship. FLOWER first tries to apply a *beside* constraint between the rooms, and if that fails the system tries to apply a *close* constraint. If both fail then the placement of the room will fail and that will be indicated to the user. This is the only way that rooms can be either in a beside or in a close relationship. Otherwise, these relationships are exclusive.

4.2 Constraint Propagation

The expert system used in FLOWER is the Echidna model-based reasoning engine. FLOWER's knowledge base is implemented in the Echidna object-oriented constraint logic programming language. Echidna provides a schema knowledge representation, a logic programming language which supports constraints among objects and a reason maintenance system for efficient dependency backtracking. In Echidna, objects are represented as predicate schemata and they are accessed by unifying goals (logical messages) with the predicates (logical methods) defined within the schema. Schema instances can be created, sent messages, or passed as arguments. More details about Echidna can be found in [43].

Constraints represent relationships between variables. A constraint network is constructed during the design session, where the variables are the nodes and the constraints are the arcs between them. In Echidna, the internal propagation of constraints narrows the domains of the variables involved, enabling a solution to be found more efficiently. A constraint is activated whenever the domain of one of its arguments is refined or bound to a particular value. This process can propagate among those variables that share constraints on their parameters.

4.3 Mediator

As described earlier, the abstract design goals (Kamada's ASR) are expressed in the knowledge base. However, a link had to be created through which a connection can be established between the abstract and the visual representations. This link is established through Echidna's External Object Protocol (XOP). To send information from the user interface code (called application from now on) to Echidna, queries are issued over this link. Likewise, Echidna terms can be unified with the terms constructed in the application. More about XOP can be found in [43] also. This connection (or mapping between ASR and VSR) is done by a combination of number of C++ routines and Echidna codes.

In the knowledge base a method is created to be external. Calls to this external method are made in the same way as to the other internal methods. External methods are the means of letting Echidna know, that it should expect methods be defined elsewhere (not on the knowledge base). External methods are generally used to get queries from users.

An example from FLOWER's knowledge base is shown in Figure 4.11. Here, the "room_maker" external method expects the size parameters of the room to be defined by the user.

When the knowledge base is loaded and the Echidna compiler notices an external method declaration, the compiler asks the application for a method handler for that method. The application creates a handler and gives it to the compiler.

The method handler defines a function which dynamically creates a method instance whenever its method is called from Echidna. That method instance has an associated array of arguments which correspond to the arguments of the Echidna goal.

If an Echidna term which is an argument to an external method is refined or restored, Echidna sends a message to the associated method instance in the application.

```
extern rm_maker(_,_,_,_,_).


schema room
{
.
.
.

        init(Type, {0..63} Id):-
           rm_maker(Id, Xpos, Width, Height, Ypos).
}
```

Figure 4.11: External Method

In this way, other parts of the design system (external to the expert system) can make use of results generated by the reasoning engine, working with the knowledge base. Here, we are specifically interested in changes to the domains of design variables.

For example, consider placing a second room beside an existing room. Since rooms have variable size, the designer does not have to put the rooms precisely beside each other; it is enough to overlap them by this available difference between the minimum and maximum sizes as an indication that (s)he wants those rooms to be beside each other (shown in Figure 4.12).

When Echidna evaluates this placement, and the design goal succeeded, in this case, it will refine the domains of the width variable of the earlier placed room. At the same time, a message is sent to the associated method instance. Based on this message, the pictorial representation can also be updated, reflecting the changes (Figure 4.13).

If the room placement was not successful, i.e. a design goal failed and we undo

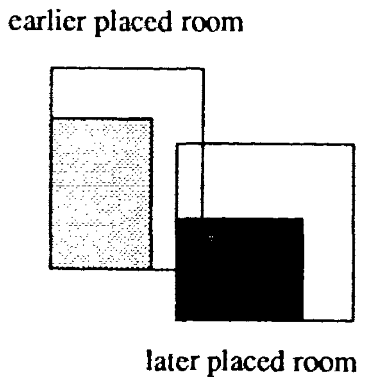


Figure 4.12: Indicated Placement of Two Rooms Beside Each Other

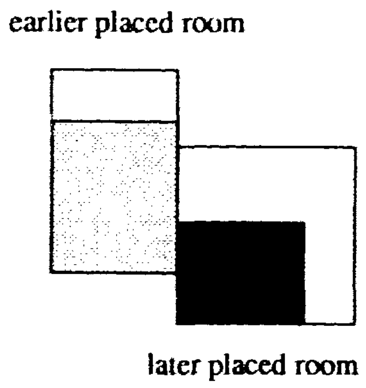


Figure 4.13: Evaluated Placement of Two Rooms Beside Each Other

```
MethodHandler* MyApplication::register_ext(char* name, int nargs)

{
   if (strcmp(name, "rm_maker") == 0)
      return new Rm_maker_Handler();
   return NULL;
}



class Rm_maker_Handler: public MethodHandler

{
public:
   virtual MethodInstance* make_method_instance
      (Argument** args, int nargs, MethodInternal mi)
   {
      return new Rm_maker_Instance(args, nargs, mi);
   };
```

Figure 4.14: Sample of Some Mediator Function

that goal, Echidna will restore the domains of the affected variables to their state before that goal was issued.

The mediator supports communication between the reasoner and the graphics part of FLOWER. The routines here are those responsible for the technicality of the communication flow. Functions supported include creating the link, loading the knowledge base, customizing the application, creating the handlers, issuing goals, undoing goals, disconnecting the link, etc. Figure 4.14 shows an example. Creation of a method handler for the *rm_maker* external method is shown on the top; and creation of the method instance is shown below.

To have the full functionality of the mapping between AR and PR, another module was created that contains the highly application dependent code. This is described in the next section.

4.4 Knowledge Base / Database Update Routines

Routines (written in C) to formulate and to issue the appropriate Echidna goals, also to receive information from Echidna. These routines are more specific to the application. Examples of such routines are getting parameter domains for a specified room, establishing ground value for parameters of a specified room, constraining the values of some parameter (or combination of parameters) of two rooms to be the same, etc.

For example, when the designer places a room by the input device, an AR is created. We need the mapping from AR to ASR, or in other words, we have to issue the corresponding design goals. Formulation of these goals is done here, in the knowledge base update module. Then, the system issues these goals to Echidna through the mediator module. After Echidna evaluated the design goal, the results are sent back through the mediator code again. Then the interface update routines take this information, forming a mapping between ASR and VSR. The interface (graphics) module will be notified by the interface update module about the changes, and it will present the pictorial representation (or mapping is done from VSR to PR). Figure 4.15 shows a knowledge base update routine.

4.5 Graphics Module

Following our earlier thread of describing our work in Kamada's terms, the graphics module is responsible for generating the pictorial representation (PR) of the design objects. It also supports the user interface. The user interface is described in the next chapter. Visualization functions are also supported here. The visualization however, is not part of the general mapping; it is calculated here entirely. The reasons behind

```
int kb_set_all(int rm_id, float xpos, float ypos, float height, float width)

{
char goal[80];
int goal_num;

   sprintf(goal, "R%d:all(%d,%d,%d,%d).", rm_id,
                 (int)xpos, (int)ypos, (int)height, (int) width);
   return(issue_goal (goal, &goal_num);
}
```

Figure 4.15: An Example of Knowledge Base Update Routines

this decision are described in Section 6.1.3.

FORMS ([34]) was used to provide the layout of the user interface and GL (SGI's Graphics Library) was used to manipulate the graphical objects and show the results of the visualization.

Rooms are represented in the same way in the graphics database, as they are in the knowledge base, with their lower left corner and their width and height. Each room type (kitchen, bathroom, etc.) has a minimum and maximum width and height.

Rooms are represented visually as shown in Figure 4.16. The minimum size is shown filled and the maximum size is shown outlined. When Echidna notifies the graphics module that the domain of a variable has changed, the representation is changed accordingly.

Visualization support is also provided by the graphics module. As we recall, FLOWER shows the following visual indications:

1. suggestions for next design steps
2. explanation of incorrect steps
3. suggestion for corrections.

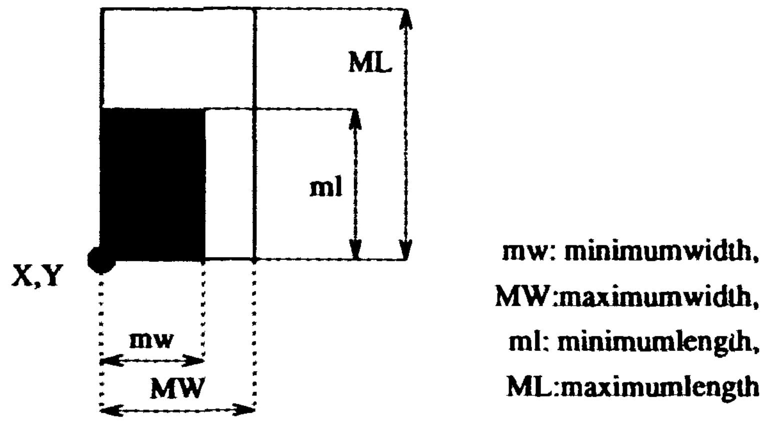


Figure 4.16: Pictorial Representation of a Room

These routines will be described next.

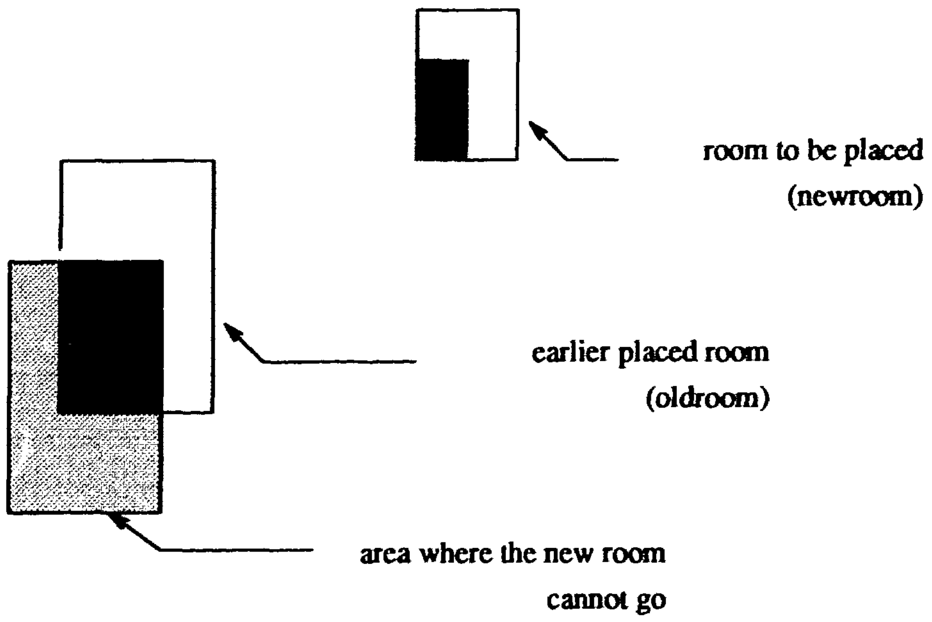
Suggestion of design steps. The suggestion will be made by showing that area where the lower left corner (room reference point) can go. The area is calculated based on the sizes of the rooms and the constraints involved.

Figure 4.17 shows how the shaded area is calculated for a no_overlap constraint.

The suggested area for the placement of a room will consider all constraints involving that room. First, subareas are calculated based on each constraint, then intersection of the subareas will result in the final area presented to the user.

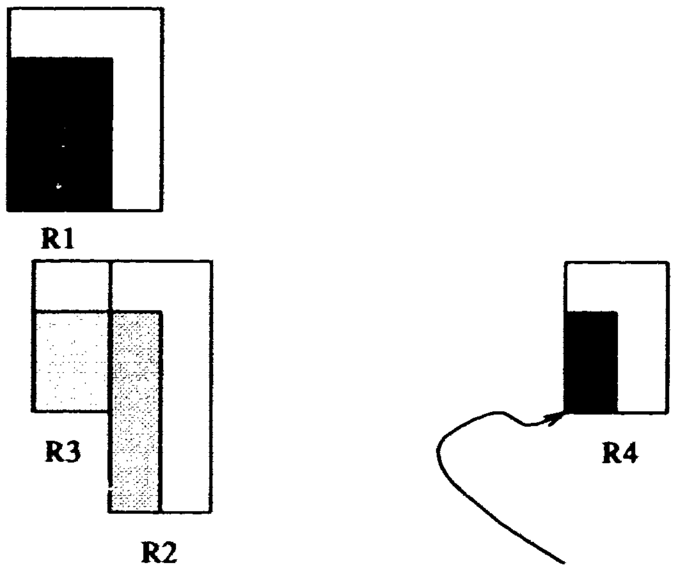
For example, suppose the user has already placed three rooms (R1, R2 and R3) and now wants to place a fourth room (R4) beside both R1 and R2 without overlapping any rooms. Figure 4.18 shows this situation. First, the subareas for each beside constraint will be calculated (Figure 4.19), and intersected (Figure 4.20). Next, the areas for the no_overlap constraints will be calculated (Figure 4.21). Finally, the areas calculated based on the no_overlap constraints will be subtracted from the previously calculated intersection. Figure 4.22 shows the remaining final area. Only this final result is presented to the user.

Explanation of incorrect steps. If the user ignored FLOWER's suggested



```
draw_rectangle(x,,y,width,height);

draw_rectangle(oldroomx - newroomx,
                oldroomy - newroomy,
                newroomminwidth,
               oldroomminheight + newroomminheight,
               NO_OVERLAPCOLOR);

draw_rectangle(oldroomx,
               oldroomy  - newroomy,
               oldroomminwidth,
               newroomminheight,
               NO_OVERLAPCOLOR);
```

Figure 4.17: Visualization of Valid Placement Area for New Room



R1, R2, R3 are existing rooms,
R4 is to be placed beside both R1 and R2,
without overlapping any existing room

Figure 4.18: Details of Calculation for Suggested Placement - part 1

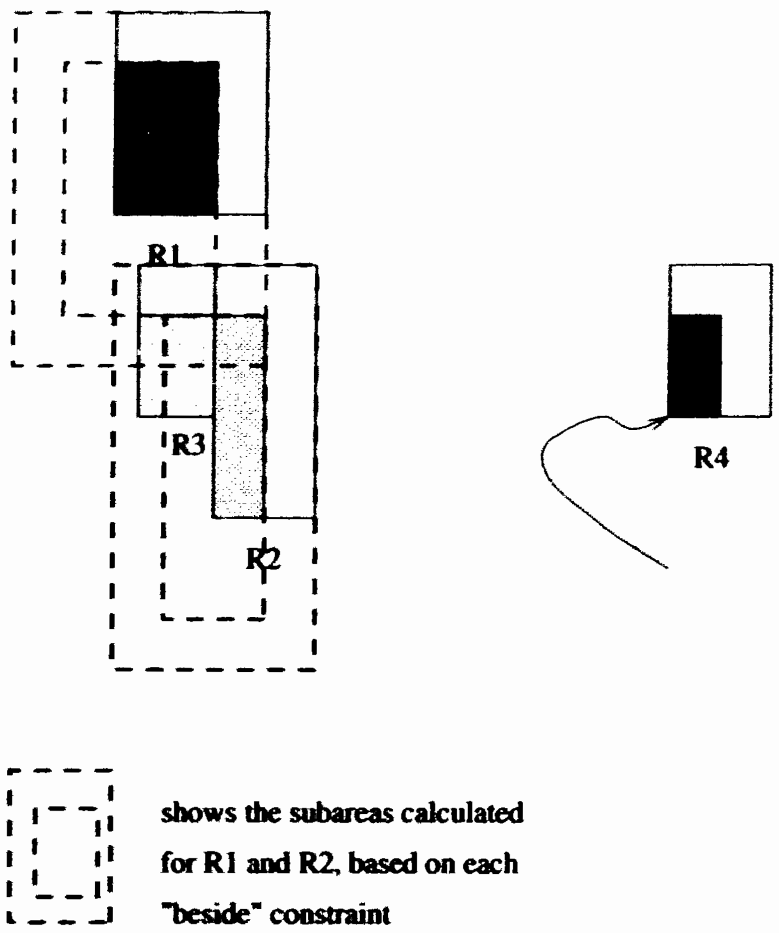
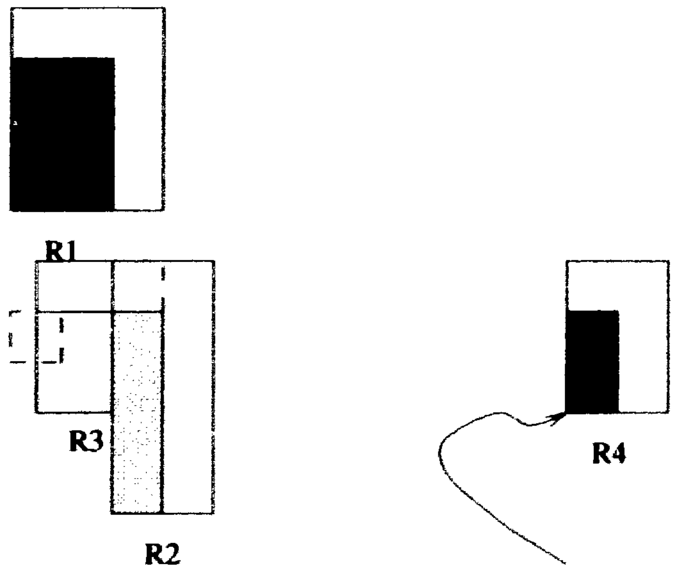


Figure 4.19: Details of Calculation for Suggested Placement - part 2



shows the intersection
of "beside" subareas:
lower left corner of R4 must go here
for it to be beside both R1 and R2

Figure 4.20: Details of Calculation for Suggested Placement - part 3

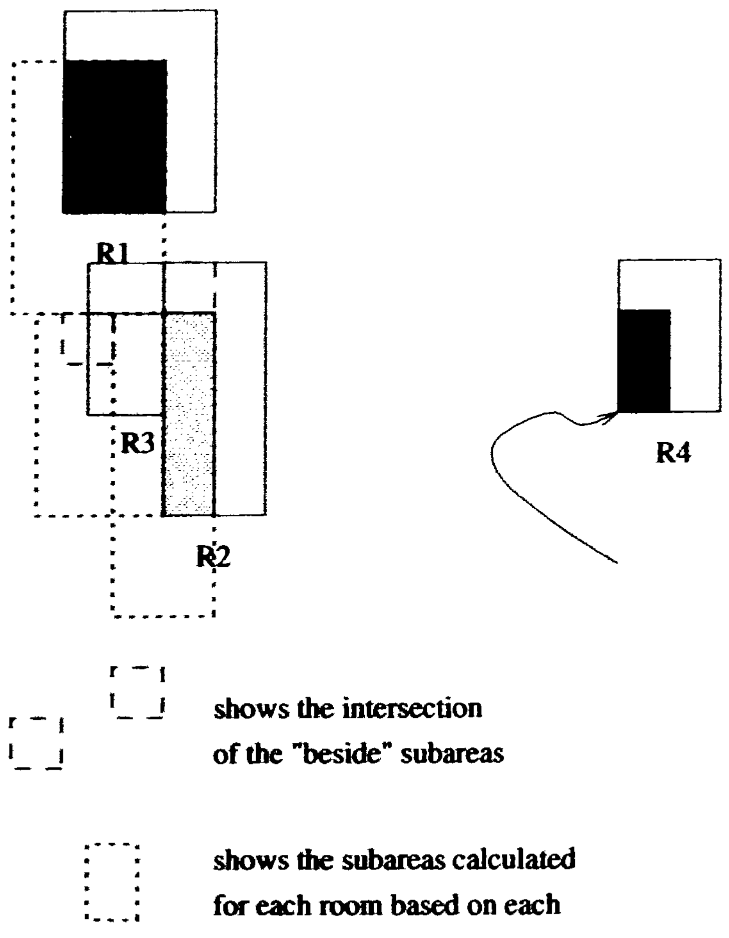

R1
R3
R2
R4
shows the intersection
of the "beside" subareas
shows the subareas calculated
for each room based on each
"no_overlap" constraint
here, the area shows where the
lower-left corner of R4 cannot go

Figure 4.21: Details of Calculation for Suggested Placement - part 4

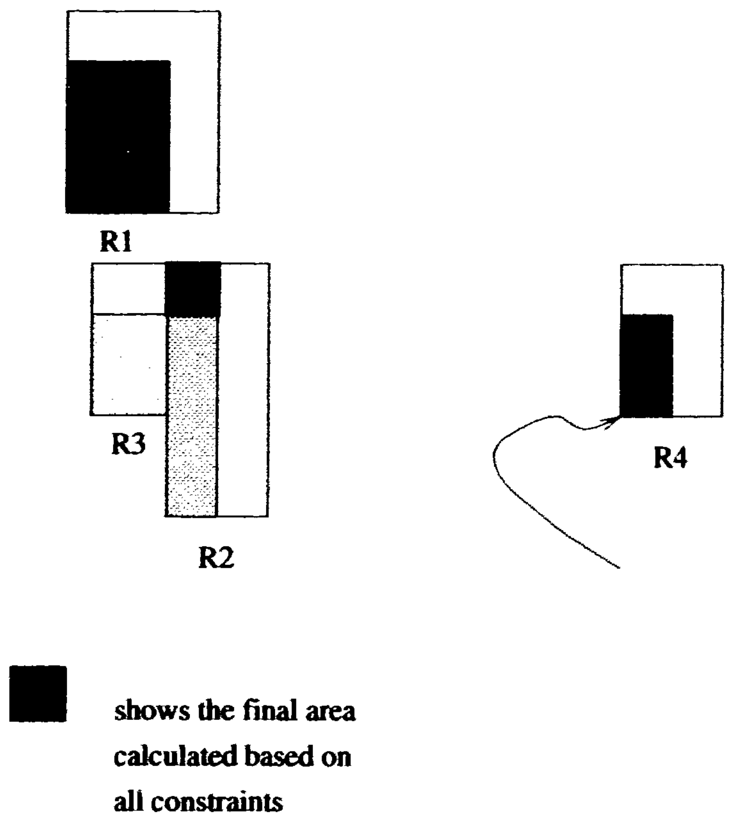


Figure 4.22: Details of Calculation for Suggested Placement - part 5

placement, a visual explanation of this mistake will be presented. Through the mapping from ASR to VSR, the database update module will notify the graphics module about the failure of a design goal and the visual representation (PR) will be as described earlier in Section 3.4.

Suggestion for correction. On encountering a problem step, the graphics module will show, based on the failed constraints, whether the rooms should be further away or closer to each other to satisfy the failed relationship, as described earlier in Section 3.4

CHAPTER 5

User Interface

5.1 The Main Interface

The user can interact with FLOWER via the following interfaces. The main interface (Figure 5.1) is responsible for almost all actions. The main area of it is the drawing board of the designer. Here (s)he can specify the perimeter of the house and place the rooms.

There are several buttons and menus are placed around the drawing area.

- Floor Plan

  Used to specify the perimeter of the house. The user must do this at the beginning of the design session, before any room placements.

- Room Buttons

  Room buttons are located at the right side of the user interface and show room types and colours. The room buttons are grouped together. On selecting a room

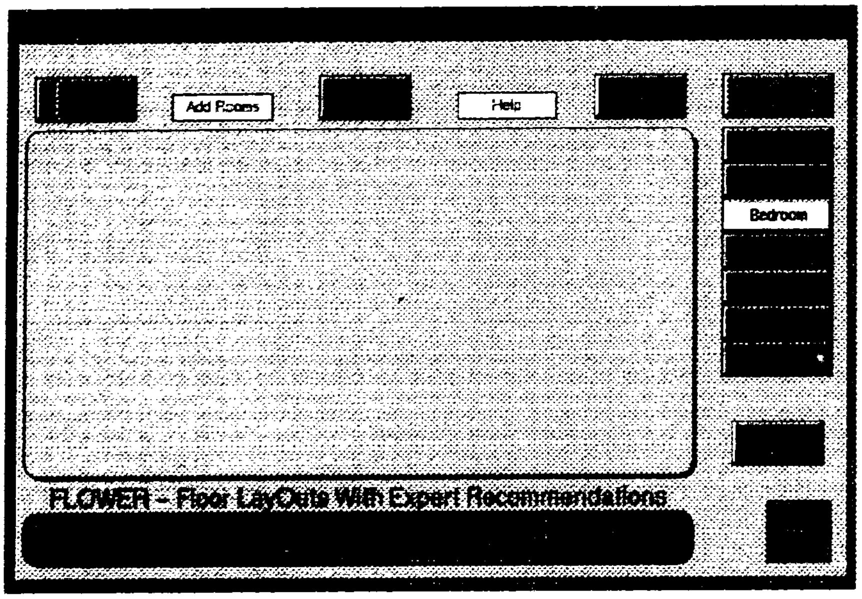


Figure 5.1: FLOWER - Main User Interface

button, all other room buttons are released, and a new room of the selected type is displayed, ready to be dragged into place. It will have the same colour as the room button. The user can drag the room around the drawing board and place it. Upon placement, all appropriate constraints will automatically be applied.

- Add Rooms

  Rooms can also be added by specifying all their parameters. The user can pick a point and then rubberband a box representing a new room. This room has not only its lower left corner constrained but its upper right corner as well. However, it will not have a preset minimum or maximum size: all parameters are completely the user's choice. Since there isn't anything known ahead about this room, there will not be any visualization available for it. However, when this room is placed. the no overlap and bounding box constraints still have to held.

- CONSTRAINTS

This button invokes the *Constraint Specification* interface (described in the next section).

- Help

  Help invokes the *Help* interface (described in section 5.3).

- Colours

  A table is shown to the user as a reminder/explanation of the colour usage (see in Figure 5.2).

- REMOVE

  The user can remove those rooms in a failed relationship.

5.2 Constraint Specification

The user can state his/her preferences by the means of the Constraint Specification interface (shown in Figure 5.3).

The user can specify constraints either using menus or (s)he can input her/his own constraint. A box is shown with all added constraints. This box can also be cleared entirely or individual constraints can be deleted from it.

5.3 Help Screen

The user is provided with help about the usage of the system. A box is shown with the available help when activating the Help interface (shown in Figure 5.4).

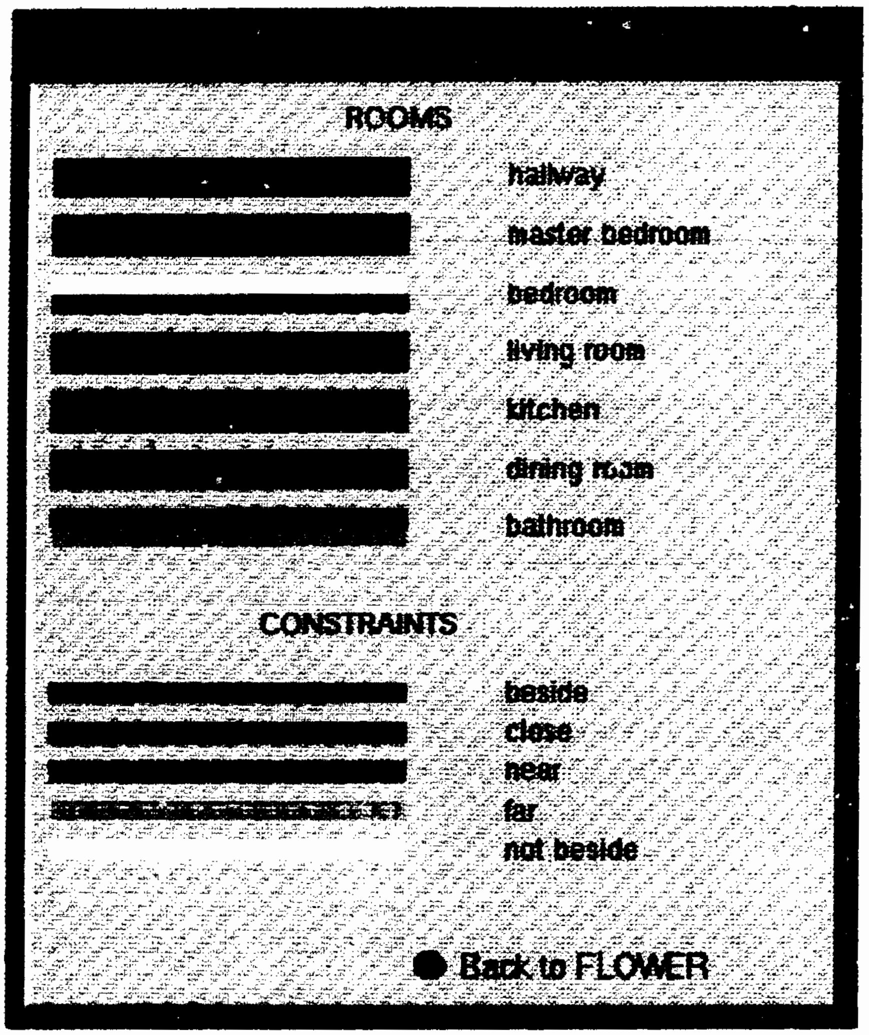


Figure 5.2: FLOWER - Colours

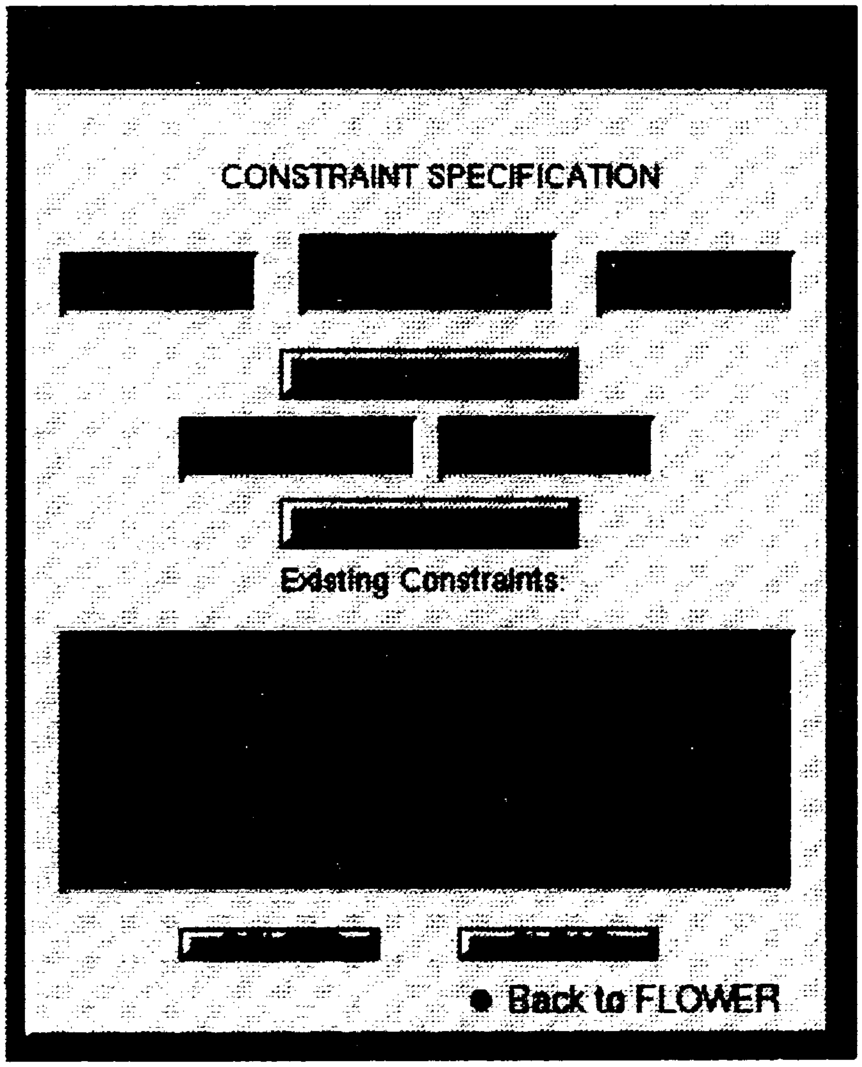
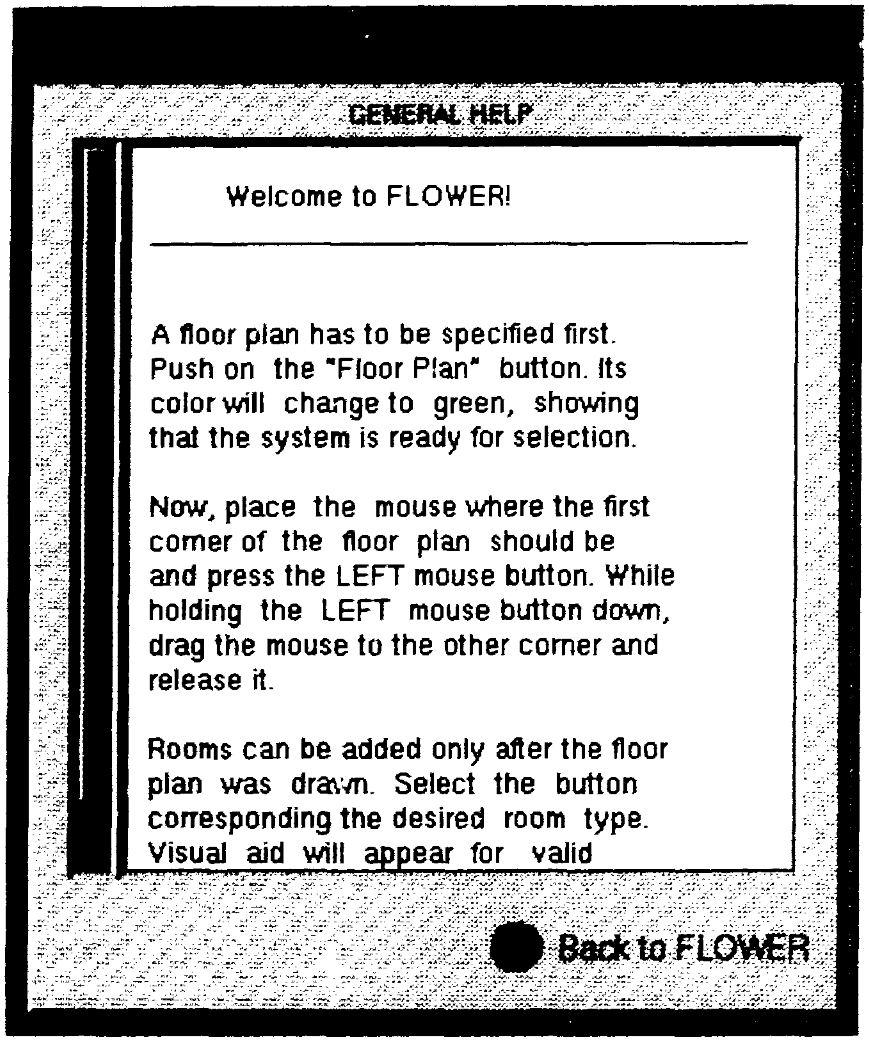


Figure 5.3: FLOWER - Interface for Constraint Specification



Welcome to FLOWER!

A floor plan has to be specified first. Push on the "Floor Plan" button. Its color will change to green, showing that the system is ready for selection.

Now, place the mouse where the first corner of the floor plan should be and press the LEFT mouse button. While holding the LEFT mouse button down, drag the mouse to the other corner and release it.

Rooms can be added only after the floor plan was drawn. Select the button corresponding the desired room type. Visual aid will appear for valid

Figure 5.4: FLOWER - Help Interface

CHAPTER 6

Discussion and Summary

6.1 Discussion on Implementation

6.1.1 Rationale Behind Fixing Lower Left Corner

In our implementation a room is positioned based on its lower left corner. The following describes the reasons for this.

An earlier approach we tried was to represent a room just with its edges, i.e. *Right*, *Left*, *Bottom*, *Top*. Then we would include constraints describing the physical realization of a room, such as the left edge is to the left of the right edge and that the bottom edge is below the top edge, etc.

We then encountered a problem that resulted in the requirement to have a constraint which pins down the location of a room. The reasons for this arise from the constraint solving techniques used in CLP languages. This is best illustrated through the use of an example:

If the only constraints on *Right* are:

- $Right - Left \leq 10$
- $Right - Left \geq 5$

and Echidna has calculated that $domain(Left) = \{1, 2, 3, 4, 5\}$ then conceptually we expect the following:

- if $Left = 1$ then $domain(Right) = \{6, 7, 8, 9, 10, 11\}$
- if $Left = 2$ then $domain(Right) = \{7, 8, 9, 10, 11, 12\}$
- if $Left = 3$ then $domain(Right) = \{8, 9, 10, 11, 12, 13\}$
- if $Left = 4$ then $domain(Right) = \{9, 10, 11, 12, 13, 14\}$
- if $Left = 5$ then $domain(Right) = \{10, 11, 12, 13, 14, 15\}$

However, Echidna will return:

- $domain(Left) = \{1, 2, 3, 4, 5\}$
- $domain(Right) = \{6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$

This leads to having a room in which

- the distance from the minimum left edge to the maximum right edge is greater than the maximum width allowable for the room, and
- the distance from the maximum left edge to the minimum right edge is less than the minimum width allowable for the room.

It is necessary to "pick" a value for one of the edges, say $Left = 5$ for Echidna to return $domain(Right) = 10, 11, 12, 13, 14, 15$. Unfortunately, this requires that we lose a degree of freedom.

A similar argument has us fix the *Bottom* edge.

Since either way we would lose degrees of freedom, we decided to specify those positions directly. This way we gained in execution time while losing a degree of freedom.

6.1.2 Alternative Methods of Removing Freedom

We have shown that we are required to give up two location degrees of freedom in order to maintain width and height consistency in our database. Our choice of fixing the bottom left corner was arbitrary and made in the interests of keeping the implementation simple.

An alternative implementation would require that the designer be required to specify constraints which eliminate a degree of freedom in both the vertical and horizontal directions whenever a room is added. This could be done by providing the designer with a choice of how to specify the location constraint — either

1. by explicitly placing a room somewhere within the floor plan, or
2. by specifying that an edge of this room is bound to an edge of another room

The explicit specification in method 1 is fairly simple to obtain. It could be implemented in one of the following ways:

- By having the designer select which corner of the room to fix and then have him move the room around the screen until in the desired position.
- Have the designer select an edge to fix in the horizontal direction, then allow him to move that edge until it is in the desired position. Repeat for the vertical direction.

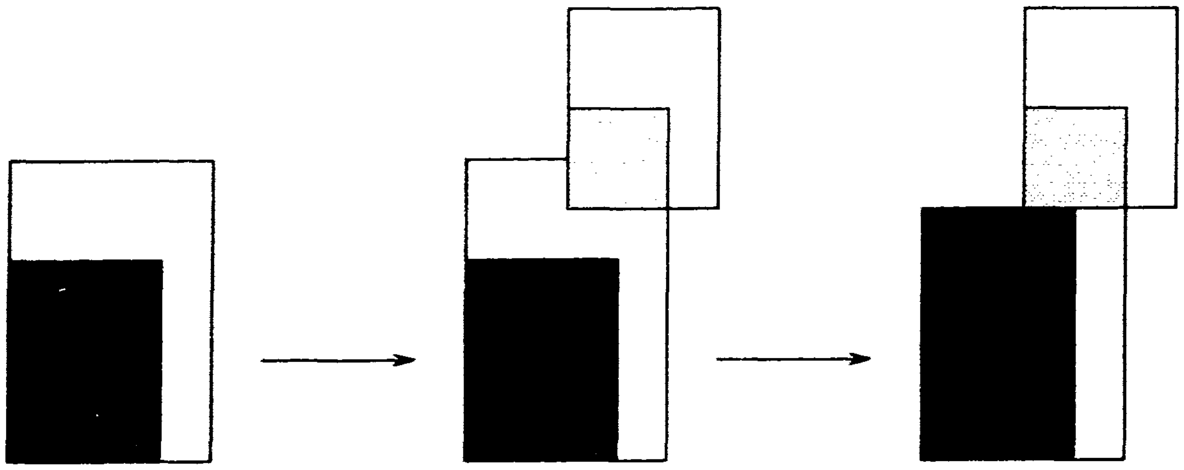


Figure 6.1: Inserting an Adjacent Room *Above* an Existing Room

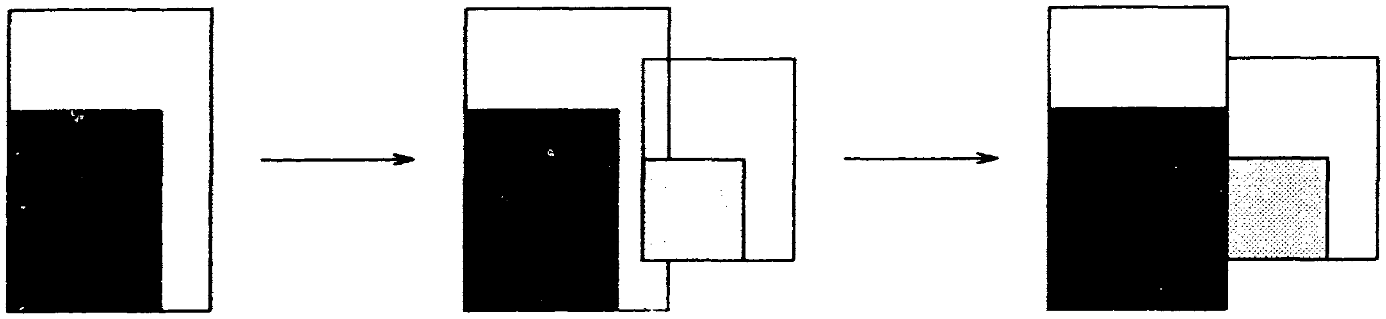


Figure 6.2: Inserting an Adjacent Room *Right* of an Existing Room

The desired way of implementing method 2 would be to have the designer place the maximum room size at a given location and to have the system automatically insert edge bindings between the new room and any other rooms it happens to be "beside". The system could ensure that both a horizontal and vertical degree of freedom has been removed before allowing the addition.

If — in order to propagate the new constraints — Echidna changes the domain of an edge variable, a message will be sent to the graphics which names the object whose edge domain has been altered. The application then knows that the new room must be adjacent to the old room and adds a beside constraint (Figures 6.1 and 6.2).

However, since our implementation fixes the lower left corner of the room, the domains of the *Width* and *Height* variables contain the only degrees of freedom that

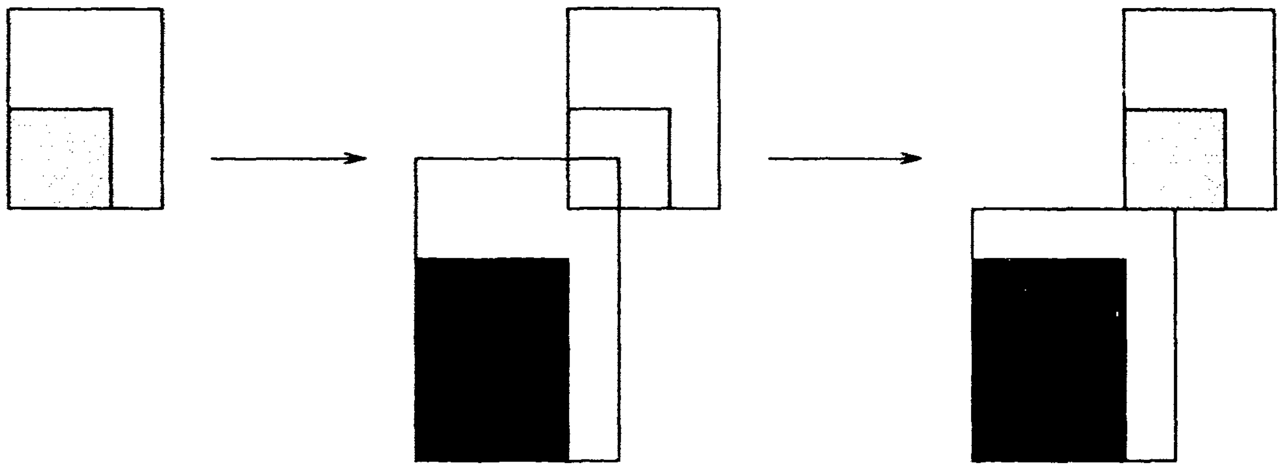


Figure 6.3: Inserting an Adjacent Room *Below* an Existing Room

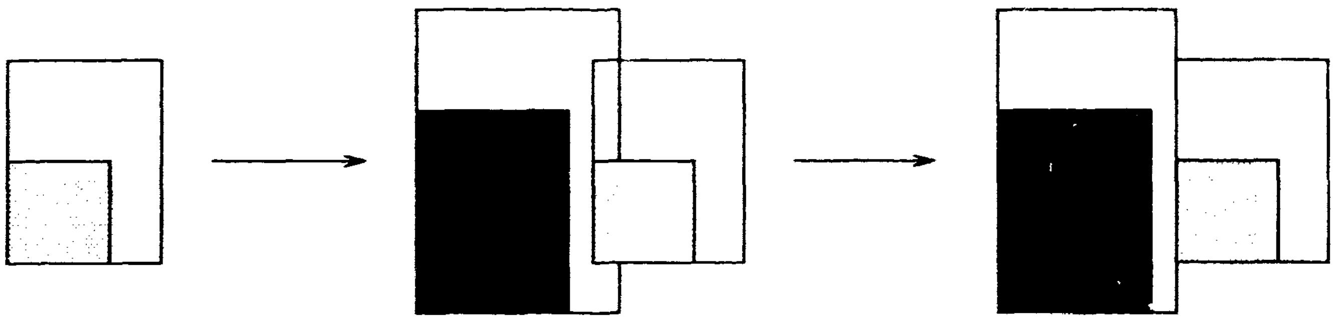


Figure 6.4: Inserting an Adjacent Room *Left* of an Existing Room

Echidna can use to propagate the constraints of the new room. This means that inserting a room below or to the left of an existing room will change the domains of the top and right edges of the new room – meaning that Echidna will *not* name the object to which our new room is adjacent. Thus, the beside constraints will not be imposed (Figures 6.3 and 6.4). The no-overlap constraints are, however, adequate to reduce the top and right edges of the new room so that the maximum size "fits" within the allowable space.

It would be nice to have Echidna inform the application as to which object caused the domain change of the new room. However, this would involve meta-reasoning and this is not part of any CLP language. In the absence of this information, a possible way to alleviate the problem would be to maintain a redundant database (outside Echidna) of all of the objects and provide search routines (within the graphics module) which could determine the identity of the needed object.

6.1.3 Division of Labour

A fundamental choice was needed between creating a system where all intelligence is encoded in the knowledge base or a system where the designing job is divided between the expert system and another (or more) high-level module. Obviously, other modules were created e.g. for visualization purposes. We decided on encoding only the basic schemas and their constraints in the knowledge base. As a result, evaluation of design goals are done by the reasoning engine but the routines that suggest the available steps by visualizing domains of design variables are a part of the graphics module.

The decision was based on the following considerations:

- One of the purposes of this research was to demonstrate that *constraint and domain visualization* can help a designer in his/her work. It seemed more natural to calculate those visual aids in the graphics side that dealt more with pure geometry than logical relations.
- We think that the time to calculate that area is important from the user's point of view. We also are aware of the fact that the reason behind being fast in geometrical calculation is due to the very simple geometrical shapes (all rooms are rectangular in this implementation). If rooms have more complex shapes, the graphics side visualization might not necessarily be faster.

Alternatively, calculating areas for visualization purposes could have been done within Echidna, even with the existing simple knowledge base. We investigated the following methods:

1. obtaining all possible solutions and extracting the valid placement area from there

This could be implemented fairly easily: when the user indicates that (s)he wants to place a new room, a goal can go to Echidna: get all solutions for the placement of that room. After getting all solutions, Echidna can return a list of those rooms and then the application can extract the needed coordinates for the area. Then the "get all solutions" goal can be undone and the user can proceed with the placement. However, we note that CLP-based systems were never intended to calculate all points in a 2D solution space, though in principle it is possible to do so.

However, this will take a considerable amount of time, especially at the beginning of the design session, when almost the whole area of the house is available for room placement. On the other hand, the calculation of valid placement area on the graphics side is very fast as there we only compute bounds of the needed area, solving a simpler problem than that expected from the reasoning engine.

We expect that with a fairly constrained design space, the time for Echidna to compute all solutions would be considerably shorter, and the graphics computing time would be increasing, though it would likely still be faster.

2. trying to solve the constraints without actually getting the variables ground, and expecting that the domains will give the acceptable areas.

   Unfortunately, the domains do not give acceptable areas. We run into exactly the same problem as described earlier in 6.1.1. Suppose, the user wants to place a room without overlapping a previously placed room. Now, the user expects the system to show the available area for the lower-left corner of the new room *before* actually placing the room. In other words, the lower-left corner of the room is not yet constrained. If we were to issue the no_overlap constraint and check how the domains of X and Y change, Echidna returns the domains of both variables unrefined. This will result in an area much bigger than is actually expected if we were to draw this. If one of these variables is constrained, then and only then will Echidna return the "expected" results.

3. by using real intervals and using *notin* [1] (which currently works only for constants, but once implemented could give a perfect solution if we only had two rooms) For simple rectangular rooms the graphics side visualization will still be faster.

   Another issue to investigate was the use of real interval variables, instead of integers. Our original no_overlap constraint was formulated as follows:

$$\begin{aligned}(Xpos &\geq Rbxpos + Rbwidth)or && (6.1)\\ (Xpos + Width &\leq Rbxpos)or && (6.2)\\ (Ypos &\geq Rbypos + Rbheight)or && (6.3)\\ (Ypos + Height &\leq Rbypos). && (6.4)\end{aligned}$$

Those variables that belong to the previously placed room start with Rb (e.g. Rbxpos etc), and the others (e.g. Xpos etc.) belong to the new room. Unfortunately, the *or* operator is not yet implemented in Echidna for constraints containing variables which have real interval domains. Thus, the constraint had to be reformulated. An obviously easy implementation of this constraint could be

$$\begin{aligned}&notin(Xpos - Width,\ \ Rbxpos,\ \ Rbxpos + Rbwidth), && (6.5)\\ &notin(Ypos - Height,\ \ Rbypos,\ \ Rbypos + Rbheight). && (6.6)\end{aligned}$$

However *notin* currently works only for intervals defined by constants. Once this is implemented, it will give a perfect, reasonably fast solution, if we only have no more than two rooms altogether. If we had more than two rooms, Echidna will still not be able to simply return a description of a complex shape

[1]**notin(E1, R1, R2) means that E1 is not any value in [R1,R2]**

(a big rectangle with many little rectangular holes) describing all no_overlap constraints. We would get severely split intervals for both X and Y variables, and the graphics module will have to compute that complex shape.

6.2 Evaluation

6.2.1 Complexity

Use of different colours for different constraints could be a problem if the total complexity of the system is high. The complexity of the system can be described by the number of rooms and the number of relationships between them.

Assigning colours to rooms is not a problem since in the current system the number of room types is very limited. The number of constraints could be a problem if we were to extend our system to be a full-scale CAD system. At this point however, the user can only specify five types of spatial constraints (beside, not beside, near, close, far). Right now we are using different shades of grey to represent constraints. Adding more and different type of relationships will result in a problem. It is unlikely that we will run out of colours, but it is very likely that the user will not be able to distinguish between them.

Unfortunately, this is a serious problem, as we cannot rely on hoping that the user will not want to add new constraints. One solution could be that the user is presented with a subset of constraints at any given time. This is a restriction but it avoids the confusion of many colours.

Another way of dealing with this would be to support some other visual representation. For example, a flag (something attached to an object) could show some of the constraints. Again, this could also represent a problem if we have too many constraints. Figure 6.2.1 shows a flag indicating two rooms being in a *beside* constraint.

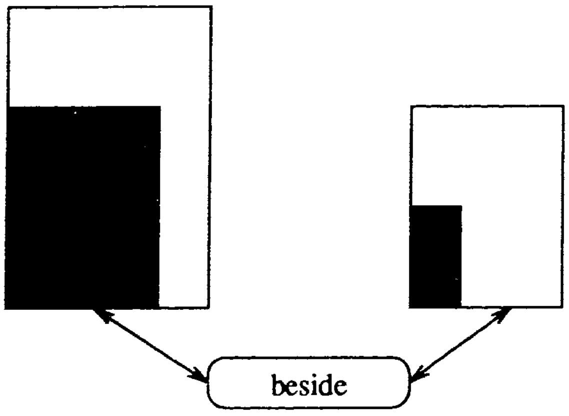


Figure 6.5: Using a Flag for Visualization

We could also use a combination of colours and flags. We could show the available areas of placement in the constraint's colour, but indicate a violated constraint with a flag.

6.2.2 Rooms with Several Relationships

Naturally, when designing a floor layout, rooms are expected to appear in more than one relationship. As long as the designer is placing them properly there will be no problem. However, (again naturally), the user cannot be expected to do so. Our system will then give a visual representation of the problem.

When the newly added room is part of several constraints, the room will be connected by lines representing all failed constraints to all other rooms in the failed relationships.

If many rooms are involved, the design space gets slightly(?) cluttered with the lines. However, the designer can easily solve this problem by removing the offending room.

We have to realize though that the aim of the designer is to properly pláce the rooms after experimenting with some of the possible problems. If the designer is just arbitrarily placing the rooms, thus causing a problem for himself, well, even the best systems cannot deal with him.

6.2.3 Suggestion for Failure Correction

At this implementation FLOWER provides some suggestions for how to proceed if constraints have failed. When it "explains" the failure by drawing the failed constraint lines between the offending rooms, arrows are displayed to show whether the offending room has to be moved closer or further. This is shown however, for each lines separately; thus if the room was in many failed relations it is very difficult to see the correct direction.

A better solution would be to have FLOWER move the offending room to a place where the relations are satisfied if that exists. This corrective action does not have to be accepted by the user, (s)he could overrule it by removing the room from the workspace. The difficulty of this method lies in the fact that there will be many ways to correctly place the room, and the system would have to try to match the placement to the one that the user specified earlier (but failed). Matching could for example be based on the closest distance from the user picked placement point to an available point.

6.3 Future Work

As we stated in earlier chapters, FLOWER is a first attempt to generate an intelligent, mixed-initiative design system with domain and constraint visualization. It can be improved in various ways starting from little improvements (such as routines that facilitate file saving, loading, restarting). These additions will not improve the functionality of the system but will make it more user-friendly.

Another visual improvement is to change the pictorial representation of the room to avoid the current asymmetry. It can be done for example, by having the room reference point be the middle of the room. In this way, we still remove degrees of freedom but at least we obtain symmetry.

6.3.1 Functional Improvements

- implementing *new* constraints

  Only a limited number of constraints were implemented in this first version of FLOWER. We implemented dimensional, restrictive, topological and functional constraints. However, many other constraint types would be useful additions:

  Accessibility constraints. When the designer finished placing rooms, the program could automatically insert doors between them. The doors could be accepted or rejected or moved as the designer wishes.

  Windows could be also inserted in outside walls, and after completing the window design, a 3D representation of the house could be presented to the user.

  Closets should be inside other rooms, with openings to them.

  Practicality constraints. Window placement could be affected by constraints, such as access to sunshine. Size and number of windows should also be influenced by these constraints.

  Interconnection constraints. In the current version we treated hallways just as regular rooms. However, hallways have different properties: e.g. they must connect rooms together.

  Stairways should connect different floors together and stair location should be the same place at both levels.

  Aesthetic constraints. Criteria still need to be identified, along with methods for implementing these constraints. However, this is an area where the user should rely on her own intuition more than in any other case. If *pretty* can be coded, it won't be *pretty* any more..

- *moving* rooms

  Another extension is to allow the user to "change his/her mind" and move an already inserted room. In this way he/she would get complete design freedom. A relaxed version of this is to let the user move the room that was placed last, as long as (s)he did not indicate further placement.

- *non-rectangular* rooms

  Rooms only are rectangular in the current version. It will be another major extension path to allow the placing of arbitrary shaped (but still Manhattan) polygons.

  The graphics system would need to be extended to allow the designer to specify the shape of each room. One problem is that if the user can draw any shape, then the visualization will have to wait, until the user finished the drawing. Formulating the constraints in the knowledge base is another issue. Formulating close/near/far constraints would be somewhat easier as the distance between rooms can be measured between some defined "midpoints" of the polygons. However, for the beside constraints it will be more difficult.

- *more initiative* from the system

  We can free the designer from some repetitive tasks by allowing partial automation at some stages of the design. E.g. when the rooms are placed already, the system could automatically place the passages. Clearly the user still has to accept the placement.

  Another initiative of the system can be placement of a room based on the specification of the designer. E.g. the designer can indicate the placement of a room and associated constraints. As a response, the system can still show the available area for the placement of that room while presenting a default placement to the user. Again, the designer can accept the placement or change the location of the room with the advice of the system.

6.4 Summary and Conclusion

The objective of this research was to explore interactive intelligent design with visual aids to the designer. We developed a system that showed that constraints are a natural way to express design goals. Maintaining those constraints is managed by the system and is not the responsibility of the user. The user is not eliminated from the design process but rather is incorporated. Because an explanation facility should be an integral part of an intelligent design system to meet users' needs for feedback on their actions, FLOWER provides such explanation via visualization of design variable domains and constraints on them. The system we developed also works with the user in a mixed-initiative style. In this way, FLOWER supports *interactive* intelligent design by providing visual suggestions or guidelines on how to proceed with the design, checking design decisions, suggesting corrective actions and marking problem steps.

As a result of implementing this system we have addressed a major limitation of current CAD technology: we have shown that it is possible to actively help a designer with the design process, without automating it. We have shown that visualization of variable domains and design constraints can provide this active help. Finally, we have shown that the FLOWER system can provide intelligent help for designing simple floor layouts.

Appendix A

FLOWER - User's Manual

Defining the Floor Plan

In order to startup the design, the user must specify the perimeter of the floor plan. If he/she fails to do so, none of the functions of the program can be activated. The floor plan defines the boundaries of the building (i.e. "outside concrete walls"). Constraints are automatically created to ensure that no part of any room can be outside of this region.

To specify the boundaries of the house, the user must select the *Floor Plan* button. The red light on this button will change to green by the selection. The user can the move the mouse into the *Workspace* area and specify one corner of the floor plan (this is the only case when the corner is not restricted to be the upper-left corner), by holding the left mouse button. She/he can then move the mouse while still holding the button and rubberband a rectangular area. Releasing the button will specify the other corner of the floor plan.

Specifying Constraints

At the beginning of the design session or any time later during the actual design the designer can specify constraints related to the location/placement of the rooms. To add the constraints the designer has to choose the *Constraints* buttons. When this button is pushed a little window appears on front of the designer. There are three small rectangles in the window. In the first and the last rectangles names of rooms are shown (i.e. kitchen, living room, etc.) and in the middle rectangle the types of available constraints are shown. The constraints implemented at this moment are spatial by nature: beside, close, near, far. The designer does not have to type in his/her choices - just has to scroll through the available possibilities by clicking the right mouse button. When (s)he made a final choice in all three windows (s)he can let the system know this by clicking on the *Back to the System* button. These constraints are taken into consideration by the system when the designer actually tries to place those rooms part of that constraint.

Adding Rooms

After the floor plan has been defined, the room buttons became activated. There are eight types of room. The types correspond to eight buttons of the user interface.

Seven of them are located on the left side of the screen grouped together, showing the similarities of the rooms. The difference between these rooms lies only in their size. To visually distinguish between them, each button is shown in different colours. The room icons hold the same colour information as their buttons. Otherwise each room has a defined minimum and maximum size and a reference point, the lower left corner. The reference point of a room is the place where the room gets inserted into the floor plan.

The following rooms are implemented: hallway, master bedroom, bedroom, living room, kitchen, dining room, bathroom. The minimum size of these rooms are somewhat corresponding to room sizes in reality, i.e. the master bedroom is bigger than the bedroom and the living room is bigger than the dining room. Since the sizes are given as range, the designer still has some freedom to make or break these conventions.

To add a room, the designer selects the button representing the desired room type. As soon as the room type is selected a shading of the valid positions for placement of the reference point of that room.

After the room type is selected and the shadow appears, the designer doesn't have to insert that room. He/she can pick other type of room and look at the available area for that one.

When the designer presses the left mouse button down the default room appears. Moving the mouse with the left button held down will move this default room around the design area. When the mouse button is released, the default room will be temporarily drawn with the lower left corner at the mouse location. The room is then added to the knowledge base.

The remaining one type of rooms can be added by a different menu. By choosing the *By Size* menupoint from the *Add Rooms* menu, the designer can add a free sized room to the design. He/she can then move the mouse into the work space area and specify the upper-left corner of the floor plan by holding the left mouse button. She/he can then move the mouse while still holding the button and rubberband the new room area. Releasing the button will specify the other corner of it. There is no available placing information for this type of room, since its size is completely free, so it could fit anywhere.

The free sized rooms can be used to fill the floor plan up completely. When the user inserts only the previous type of rooms, there could be "empty" spaces in the floor area. By inserting a free sized room to those places, the designer can fill up the gaps. (These rooms can be considered as closet spaces or extra hallways.)

The addition of a room causes the following constraints to be automatically inserted into the Echidna knowledge base:

- The room must be contained within the floor plan.
- The room must have a width/height smaller than the minimum and larger than the maximum for that room type, when this range is specified, i.e. it is not a free sized room.
- The bottom left corner of the room is fixed to be the point which the designer selected
- The new room must not overlap an existing room.
- Other constraints that the designer specified before regarding this room.

In the event that the new room overlaps an existing room, the addition of this room is rejected. All the goals are undone and the room must be removed from the floor plan.

However, when the designer specified constraints are not met, the system gives the following visual clues: the appearance of the original room changes, it will be shown outlined and an explanation is provided about the failure. Also a suggestion is given about possible corrective steps.

If the addition of these constraints to the knowledge base is consistent, then the new room will be redrawn to reflect any changed edge domains. The visual aid for insertion is cleared. If the designer wishes to add the same type of room again, he/she could do so, without needing to push the same type of button again. To activate the visual aid again he/she has to move the mouse in the work space area. The new shadow will appear showing again the actual available places. Obviously the designer can choose from the other rooms, if he/she wishes to do so.

As it was previously mentioned, each room are shown in different colour. By clicking on a room button, this colour will be turned of and the pushed button is shown in white. In this way, the designer can see which room was selected. Furthermore, the appearing shaded area will also correspond to the same colour. To still clearly show the room icon, the shaded area will appear with using the same hue, but in low saturated. So e.g. the colour of the chosen room is red, then the shaded area will be pinkish.

Getting Help

To make the program even more user friendly, help facilities are added to it. The designer can ask for help at any time of the session by choosing from the *Help* menu. A little window will appear and brief information about the usage can be obtained. Based on the selection a scrollable text will be shown about the various aspects of the program. Help files are available about how to use the program in general; how to draw a floor plan; how to add rooms, what the difference is between the room buttons and the room menu, i.e. between the constrained sized and free sized rooms; what the shaded area means; what happens when a constraint fails and how to exit the program. A button (*Back to FLOWER*) on the help screen has to be pushed in order to get back to the program again, and the design could be continued.

Leaving the Program

When the design session is finished, the user can exit by hitting the *Exit* button. A built-in safety feature exists against accidental exiting. The user is asked whether really meant to quit or it was just an accidental mouse movement. If he/she didn't mean to leave, he/she can return to his/her design and just simply continue the design session.

REFERENCES

[1] Ö. Akin. How do architects design. In *Artificial Intelligence and Pattern Recognition in Computer Aided Design*, pages 65–103. IFIP North Holland Publishing Company, 1978.

[2] Ö. Akin. Expertise of the architect. In *Expert Systems for Engineering Design*, pages 173–198. Academic Press, 1988.

[3] S.R. Alpert. Graceful interaction with graphical constraints. *IEEE Computer Graphics and Applications*, 13(2):82–91, 1993.

[4] A. Barr and R. Barzel. A modelling system based on dynamic constraints. *ACM SIGGRAPH Computer Graphics*, 22(4):179–188, 1988.

[5] C.A. Baykan and M.S. Fox. An investigation of opportunistic constraint satisfaction in space planning. In *Tenth International Joint Conference on Artificial Intelligence*, pages 1035–1038, August 1987.

[6] C.A. Baykan and M.S. Fox. Constraint satisfaction techniques for spatial planning. Extended Abstract, October 1988.

[7] C.A. Baykan and M.S. Fox. Opportunistic constraint-directed search in space planning. In *IFIP Working Group 5.2 Workshop on Intelligent CAD*, pages 1–6, 1988.

[8] E. Bier and M.Stone. Snap-dragging. *ACM SIGGRAPH Computer Graphics*, 20(4):233–240, 1986.

[9] M. Blonsky. *On Signs*. Basil Blackwell, Oxford, 1985.

[10] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM TOG*, 5(4):345–374, 1986.

[11] T. Calvert, J. Dickinson, J. Dill, W. Havens, J. Jones, and L. Bartram. An intelligent basis for design. In *IEEE Pacific Rim Conference on Computers, Communications and Signal Processing*, pages 371–375, Victoria, BC, May 9-10 1991.

[12] P.P. Chen. The entity relationship model - towards a unified view of data. *ACM Database Systems*, 1(1):9–36, 1976.

[13] J. Dill, J. Jones, and Stefan W. Joseph. Intelligent computer aided design. *International Journal of CADCAM and Computer Graphics*, 8(2):175–184, 1993.

[14] C.L. Dym, R.P. Henchey, E.A. Dellis, and S. Gonick. A knowledge-based system for automated architectural code checking. *Computer-Aided-Design*, 20:137–145, 1988.

[15] W.H. Fawcett. Design knowledge in architectural CAD. *Computer-Aided-Design*, 20:83–90, 1988.

[16] F.S. Frome. Improving colour CAD systems for users: Some suggestions from human factor studies. *IEEE Design and Test*, pages 18–27, 1984.

[17] J.S. Gero, M.L. Maher, and W. Zhang. Chunking structural design knowledge as prototypes. In *Artificial Intelligence in Engineering Design*, pages 3–21. Elsvier, 1988.

[18] T.R. Henry and S.E. Hudson. Using active data in a UIMS. In *ACM SIGGRAPH Symposium on User Interface Software*, pages 167–178, Banff, AL, Canada, October 17-19 1988.

[19] S.E. Hudson. Adaptive semantic snapping - a technique for semantic feedback at the lexical level. In *CHI'90*, pages 65–70, April 1990.

[20] S.E. Hudson and A.K. Yeatts. Smoothly integrating rule-based techniques into a direct manipulation interface builder. In *UIST'91*, pages 145–153, Hilton Head, SC, November 11-13 1991.

[21] T. Kamada. *Visualizing Abstract Objects and Relations - A Constraint Based Approach*. World Scientific, Singapore, 1989.

[22] T. Kamada and S. Kawai. Advanced graphics for visualization of shielding relations. *Computational Vision, Graphical Image Processing*, 43(3):294..312, 1988.

[23] T. Kamada and S. Kawai. A simple method for computing general position in displaying 3D objects. *Computational Vision, Graphical Image Processing*, 41(1):43–56, 1988.

[24] T. Kamada and S. Kawai. A general framework for visualizing abstract objects and relations. *ACM TOG*, 10(1):1–39, 1990.

[25] S. Kochar. A prototype system for design automation via the browsing paradigm. In *Graphics Interface '90*, pages 156–166, Halifax, NS, May 14-18 1990.

[26] H. Levkowitz and G. Hermann. Colour scales for image data. *IEEE Computer Graphics and Applications*, 12(1):72–80, 1992.

[27] M.L. Maher. HI-RISE and beyond: directions for expert systems in design. *Computer-Aided-Design*, 17:421–427, 1985.

[28] M.L. Maher. Expert systems for structural design. In *Expert Systems in Engineering*, pages 147–161. Springler-Verlag, 1987.

[29] M.L. Maher. HI-RISE: An expert system for preliminary structural design. In *Expert Systems for Engineering Design*, pages 37–52. Academic Press, 1988.

[30] J.H. Maloney, A. Borning, and B.N. Freeman-Benson. Constraint technology for user-interface construction in Thinglab II. In *OOPSLA*, pages 381–388, 1989.

[31] B.J. Meier. ACE: A colour expert system for user interface design. In *ACM SIGGRAPH Symposium on User Interface Software*, pages 117–128, Banff, AL, Canada, October 17-19 1988.

[32] B.A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, San Diego, CA, 1988.

[33] G. Nelson. Juno, a constraint-based graphics system. In *ACM SIGGRAPH*, pages 235–243, July 1985.

[34] M.H. Overmars. Forms library; a graphical user interface toolkit for silicon graphics workstations version 2.2. Department of Computer Science, Utrect University, 1993.

[35] R. Oxman and J.S. Gero. Designing by prototype refinement in architecture. In *Artificial Intelligence in Engineering Design*, pages 395–412. Elsvier, 1988.

[36] J.C. Platt and A.H. Barr. Constraint methods for flexible models. *ACM SIGGRAPH Computer Graphics*, 22(4):279–288, 1988.

[37] Z. Pylyshyn. The design process: Lecture notes. Lecture Notes of University of Western Ontario, 1988.

[38] T.W. Reps. *Generating Language-Based Environments.* MIT Press, Cambridge, MA, 1984.

[39] P. Rheingans and B. Tebbs. A tool for dynamic explorations of colour mappings. In *ACM SIGGRAPH Symposium on User Interface Software*, pages 145–146, 1990.

[40] M.A. Rosenmann and J.S. Gero. Design codes as expert systems. *Computer-Aided-Design*, 17:399–409, 1985.

[41] M.A. Rosenmann, J.S. Gero, P.J. Hutchinson, and R. Oxman. Expert systems application in computer-aided design. *Computer-Aided-Design*, 18:546–551, 1986.

[42] D.D. Seligmann and S. Feiner. Automated generation of intent-based 3D illustrations. *ACM SIGGRAPH Computer Graphics*, 25(4):123–132, 1991.

[43] S. Sidebottom, W. Havens, and S. Kindersley. Echidna constraint reasoning system (version 1): Programming manual. Expert Systems Laboratory. Centre for Systems Science, 1992.

[44] I.E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Spring Joint Computer Conference*, pages 329–346, Montvale, NJ, 1963.

[45] S. Takahashi, S. Matsuoka, A. Yonezawa, and T. Kamada. A general framework for bi-directional translation between abstract and pictorial data. In *UIST'91*, pages 165–174, Hilton Head, SC, November 11-13 1991.

[46] E.R. Tufte. *Envisioning information.* Graphics Press, Chesire, CT, 1990.

[47] A. Witkin and M. Kass. Spacetime constraints. *ACM SIGGRAPH Computer Graphics*, 22(4):159–168, 1988.

[48] A. Wolfe, M. Brown, D. Greenberg, M. Keeler, A.R. Smith, and L. Yaeger. The visualization roundtable. *ACM SIGGRAPH course notes*, 22(19):2–12, 1988.

[49] R.S. Wolff. Visualization in the eye of the scientist. *ACM SIGGRAPH course notes*, 22(19):13–20, 1988.

[50] B.V. Zanden, B.A. Myers, D. Giuse, and P. Székely. The importance of pointer variables in constraint models. In *UIST'91*, pages 155–164, Hilton Head, SC, November 11-13 1991.