NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
AN APPROACH TO CONCURRENT ENGINEERING OF USER INTERFACES

Sai Rani Vallurupalli

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 1995
© SAI RANI VALLURUPALLI, 1995
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ABSTRACT

AN APPROACH TO CONCURRENT ENGINEERING
OF USER INTERFACES

Sai Rani Vallurupalli

The user interface of a software system is an artifact, through which the user and the system interact with each other. The user interface supports the user with a medium for communicating with the underlying software system. Therefore, it is often the principal determinant of a system's success. It is not given adequate importance in the development of software systems. There is no software process model where in the necessary importance is given to the various stages of the user interface development. In today's competitive world of computer applications, it has become necessary for fast prototyping of software systems. This has become more essential with regard to the user interface subsystem which exhibits look and feel characteristics. There should be a software process model which will let the user interface development and the computational part of the software development, to take place more or less concurrently. In this thesis, we propose an Advanced Evolutionary Prototyping Model. It has two characteristics: (a) The various stages of user interface development are clearly identified; (b) It supports the concurrent development of the user interface with other parts of the software.

The Advanced Evolutionary Prototyping Model perceives the software development as two loosely coupled, concurrent processes: the user interface process, and the computational process. The activities involved during the various phases of the computational process affect the user interface process only at the pre-determined synchronization points in the Advanced Evolutionary Prototyping Model. The Concurrent User Interface Methodology (CUIM), developed and discussed in this thesis, provides a systematic procedure to put the Advanced Evolutionary Prototyping Model into practice.
As a part of CUIM, different notations are proposed for various stages of the user interface development. By means of a case study, this thesis also demonstrates the feasibility of the Advanced Evolutionary Prototyping Model in general, and the CUIM methodology in particular. We believe that the various steps in CUIM when followed, would result in an easy to use, and easy to learn user interface for a given software system, and reduce the total software development time.
Dedication

To my loving husband
Srinivas
Acknowledgment

At this moment, I would like to thank many people who helped me to get through the tough times and who made this thesis possible.

First and foremost, I would like to thank my supervisor Dr. T. Radhakrishnan for both his supervision and financial support. I am very thankful for the opportunity I had, to work with Dr. Radhakrishnan during my master’s. His many meetings were a source of motivation and inspiration throughout my research. Moreover, his insistence on excellence and quality is a learned lesson which I will carry during the rest of my career. I consider Dr. Radhakrishnan to be much more than a supervisor. He is a father figure who cares very deeply about the welfare of his students.

The financial support provided for my work, by BNR and NSERC, through a research grant to Dr. T. Radhakrishnan is gratefully acknowledged.

I would like to thank the undergraduate students in Computer Science Department, for their cooperation during the testing of the user interface developed. Next, I would like to thank Joanna Sienkiewicz, and Judit Barki whom I could always count on for being my good friends. Our time in the lab, discussing different issues is memorable. I would also like to thank all my family friends who helped me when the need came.

To my daddy & amma, to whom this means so much, I want to thank them for their love and encouragement. Also to my brother-in-law and my dear sister for their love and inspiration, and for their e-mails which always made me laugh.

Last, and the most, I like to give a special thanks to my husband who was always there whenever times became difficult. I deeply appreciate his care, warmth, and moral support during my studies at Concordia.
Contents

List of Tables xi

List of Figures xiii

1 Introduction 1

1.1 Software Crisis 2

1.2 State of the User Interface Development Process 3

1.3 Supporting Concurrency in the Software Development Process 5

1.4 Overview of the thesis 7

2 User Interface Design Models 9

2.1 What is a Model? 9

2.2 Software Process Models 10

2.3 User Interface Design versus Software Design 14

2.4 Designing the User Interface 16
2.4.1 Using Well-Defined Principles and Techniques ................. 16
2.4.2 Mimicking the Existing Software Process Models ............... 18
2.4.3 Using Ad-hoc Methods ........................................... 20
2.4.4 Other Existing Models for User Interface Design ............... 21
2.5 Conclusion ..................................................................... 27

3 Concurrent Engineering - An Overview ................................. 20

3.1 A Definition .................................................................. 30
3.2 Extending the Concept to User Interfaces - Advantages ........... 30
3.3 Concurrent Engineering in User Interfaces ............................ 31

3.3.1 Methodologies that Support Concurrency in User Interfaces . 32
3.3.2 Evolution of CUIM ................................................... 35
3.4 Applying the Advanced Evolutionary Prototyping Model ........... 38
3.5 Dialog Specification in CUIM .......................................... 47

3.5.1 Extended State Transition Diagrams .............................. 48
3.5.2 Interaction Diagrams .................................................. 52
3.6 Summary ....................................................................... 54

4 User Interface Analysis in CUIM ........................................ 56

4.1 Constructing the User Profile .......................................... 56
4.2 Specifying the Dialog between the User and the Interface ....... 61
4.3 Specifying the Interactions between the IOBs and COBs ............ 73
4.4 Verifying the Extended State Diagrams with the Interaction Diagrams 79
4.5 Ensuring Consistency between IAD & CAD .......................... 83

5 User Interface Design in CUISM ..................................... 86

5.1 Dialog Design ....................................................... 87
  5.1.1 Identifying Appropriate Dialog Styles - Cell Matrix Method .... 87
  5.1.2 Integrating the Multiple Dialog Styles .......................... 94

5.2 Refining the Dialog between the User & the Interface .......... 95

5.3 Static Structure of the Interface .................................. 100
  5.3.1 Listing the Class Charts ...................................... 100
  5.3.2 Depicting the Spatial Organization of the Interface ........... 103
  5.3.3 Specifying the Class Descriptions .............................. 108

5.4 System Interactions ................................................ 111
  5.4.1 Behavioral Specification ....................................... 111
  5.4.2 User Interface Configuration ................................. 114

5.5 Reviewing the Design ............................................ 123

5.6 Ensuring Consistency between IAD & IDD .......................... 132

5.7 Ensuring Consistency between IDD & CDD .......................... 136
6 Implementation and Conclusions

6.1 Implementing the User Interface ........................................ 139

6.2 Testing the User Interface .............................................. 141

6.3 The Hyper-media Design Tool .......................................... 152

6.4 Conclusions ............................................................... 155

6.5 Future Work .............................................................. 157
## List of Tables

4.1 User Profile .................................................. 60

4.2 User Goals ..................................................... 62

4.3 Overview of the Dialog Specifications .......................... 63

4.4 Data Dictionary for transition values *to & from* a function call ..... 68

4.5 Data Dictionary for IOBs in CAS ................................ 74

4.6 Data Dictionary for COBs in CAS ............................... 75

4.7 Data Dictionary for Internal Events in Interaction Diagrams .... 76

4.8 State-IOB Association Table .................................. 82

4.9 FunctionCall-COB Association Table .......................... 82

4.10 Event Verification Table ...................................... 83

4.11 Necessary Changes to the COBs ............................... 85

5.1 Dual view Association Table .................................. 126

5.2 Class Attribute Table .......................................... 128

5.3 Command-Callback Table ...................................... 130
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Constraint-Invariant Association Table</td>
<td>131</td>
</tr>
<tr>
<td>5.5</td>
<td>Event-Callback Table</td>
<td>132</td>
</tr>
<tr>
<td>5.6</td>
<td>Event Correspondance Table</td>
<td>133</td>
</tr>
<tr>
<td>5.7</td>
<td>Node-Class Association Table</td>
<td>134</td>
</tr>
<tr>
<td>5.8</td>
<td>Action Response Table</td>
<td>135</td>
</tr>
<tr>
<td>5.9</td>
<td>Overview Table</td>
<td>137</td>
</tr>
<tr>
<td>6.1</td>
<td>Callback Verification Table</td>
<td>145</td>
</tr>
<tr>
<td>6.2</td>
<td>Error List</td>
<td>149</td>
</tr>
<tr>
<td>6.3</td>
<td>Test Results</td>
<td>151</td>
</tr>
</tbody>
</table>
List of Figures

2.1 The Spiral Model (from Boehm '88) .................... 13

3.1 Advanced Evolutionary Prototyping Model .................. 37

3.2 Current Advising Procedure ......................... 44

3.3 Automated Advising Procedure ...................... 46

3.4 Symbols in Extended State Transition Diagrams .............. 51

3.5 Basic Symbols used to build Interaction Diagrams ............ 52

3.6 Interaction Diagram: when the user enters the account number . . . 53

4.1 Top level state transition diagram of the Course Advising System .. 64

4.2 The 'pref choi' sub-conversation .......................... 66

4.3 The 'pref course' sub-conversation ........................ 67

4.4 The 'pref time' sub-conversation ........................... 67

4.5 The 'pref campus' sub-conversation ........................ 68

4.6 The 'pref load' sub-conversation ........................... 69

4.7 The 'unpref choi' sub-conversation ........................ 70
6.3 The selector Interface ........................................ 142
6.4 The preferences Interface .................................. 142
6.5 The constraints Interface .................................. 143
6.6 The advisor Interface ....................................... 143
6.7 The Hyper-media Node-Link Diagram ..................... 153
Chapter 1

Introduction

Computers are playing a vital role in our day to day life. They are being used in almost all the disciplines of life by people with varying background, knowledge and task experience. Both, computer users and software developers accept that just being able to do a task on a computer is not the only important factor. The question asked now is: "Can this task be performed with ease, on a computer?" Therefore, user interface design of computer systems has become an important research topic.

The user interface of a system relates itself with three things: the system, the user of that system and the way in which both of them interact. It is the user interface that provides the user with a language for communicating with the system. Barfield[Barf93] gives an analogy that shows how unacceptable the current state of computer-user interface design would be if it had occurred in other, more familiar, areas of design for use. "Imagine you are at the exhibition talking to a salesperson. Just listen to this! It's got a nine-inch blade, Sheffield steel, tempered cutting edge, and here are some glossy pictures of finely cut up vegetables produced by it. It can do this and this. You are convinced, you buy it, get it back home and opened the box. You discover that it hasn't got a handle! The salesperson wasn't lying, it has got all the features he listed, and you can use it to chop and slice vegetables so that they look just as good as those in the glossy pictures he showed you. The problem is that it's awkward, it's complicated, it's time consuming, and it involves learning
completely new ways of chopping vegetables dictated by the silly user interface design of the knife!" Therefore, what is needed is to take the problem of engineering the user interface as seriously as any other part of software engineering.

Scientific analysis of user interface design is a novel area of research. Most of the research in the literature focuses on small parts of human-computer interaction problem. Norman[Norm84] proposes high-level models which address the important cognitive relationships between users and computers. Shneiderman[Shne87] identifies critical issues on which developers should focus when creating user interfaces. However, not much research has been done in developing a fully articulated process, which addresses all the phases of user interface development. As a consequence, user interface developers are left with little guidance on how to develop user interfaces. Methods and techniques that help user interface designers to make good design decisions for a given product with regard to its users are necessary.

1.1 Software Crisis

In the early days of computing, in order to get the computer to do something useful, the process of programming was viewed essentially as how to place a sequence of instructions together. The program was written generally by a mathematician or an engineer to solve an equation of interest to him or her. The problem was just between the programmer and the computer. There was no distinction between the programmer and the end user of the application. As computers became cheaper and more common, people with different background started using them. The growing need in software development lead to the development of higher level languages which made it easier and quicker to develop applications. But still, the activity of getting the computer to do something useful was essentially done by a person who was writing a program for a well-defined task. This resulted in the creation of a new profession: "the programmer". Rather than doing the job by themselves, people started assigning the task of writing a program to the programmer. Thus, applications were developed, by just writing a program. But it was found that, as new requirements evolved, subsequent changes to the code structure were expensive and the results became
less reliable. Many solutions were proposed and tried. Some suggested that the solution lay in better management techniques. Others argued for better languages and tools. The final consensus was that the problem of building the software should be approached in the same way that engineers build other large complex systems such as bridges, machinery, and airplanes. These problems and suggestions resulted in introducing the *design* phase prior to coding. But, since the user and the developer of the software product are not the same person, the user had to specify the task in different notations. The developer then interpreted this specification and translated it into a precise programming notation. This sometimes resulted in the developer misinterpreting the user's intentions. These problems underscored the need for a *requirements* phase prior to the design and coding phases. Thus, software engineering is viewed as having an entire life cycle, starting from conception and continuing through design, and development.

The process we follow to build, deliver, and evolve the software product, from the inception of an idea all the way to the delivery and final retirement of the system, is called a *software development process* [Ghez91]. Many different life cycle *models* have been proposed, to organize the software development process that enable us to produce high-quality software products reliably, predictably, and efficiently. An overview of the most common software life cycle models is given in Chapter 2.

### 1.2 State of the User Interface Development Process

An user interface acts as an intermediary between the user and the system. The user interface forms the part of the software to be developed. Currently, software engineers employ different methods for the design and development of user interfaces. In the article, "Designing the Star User Interface", the authors Smith, et. al. [Smit82] use some general principles and techniques for developing the user interface. Some times, the customer himself does not know the complete set of requirements. New requirements evolve as the product is developed. As a result, Draper and Norman [Drap85] suggest that an iterative/prototype modeling must be adopted during user interface
development. The GOMS model [Kier88] for developing user interfaces, mimics the prototype model specified in traditional software engineering. This model focuses on "how to do it", with regard to the intended tasks accomplished. Ad-hoc methods which combine the design of the user interface with the software design are also proposed [Sutc91]. Sutcliffe and McDermott [Sutc91] propose a method for user interface design which builds on Structured Analysis/Structured Design (SA/SD) [Pete87].

In today's environment, software is developed not for personal use by a programmer, but for people with little or no background in computers. By developing the user interface together with the software system, software engineers tend to develop the software by focusing on efficiency of code and flexibility of architecture, and they often overlook or over simplify the real issues and constraints of the end user. This result in the development of software systems whose user interfaces does not satisfy the end user. Chapter 2 gives a detailed discussion of how the skills required for user interface design are different from the skills required for software design.

Some other models which focus on the user's cognitive interactions with the different components of the user interface are proposed in [Norm84], and [Fole82]. These models are concerned in structuring the components of an user interface to be developed. Even though these models deal with understanding the user intentions, they do not provide a well-defined process which could guide the development of an user interface.

Also, many researchers ([Bilj88], [Drap85], [Hill86], [Radh93]) emphasize the need for a rigorous separation of the functionality of a software system from its user interface. Distinguishing the user interface from the application is a reasonable design principle because a proper modularity permits the user interface and the application to evolve more independently. Separation simplifies the construction and modification of the user interface. It allows teams with different expertise to work together.

From the above, we see that there do not exist methods, which provide a systematic procedure for the design and development of user interfaces, and to link the development of the functional(non-interface) part of an application system with that of the user interface. Therefore models are needed to support the design and development of the user interface which provide linkage to the functional part of the
software system. Such models could then lead to methodologies and tools for user interface design.

1.3 Supporting Concurrency in the Software Development Process

The principle of separating the user interface design from the functional design of a software system ([Bilj88], [Drap85], [Hill86], [Radh93]), remains sterile unless some way is provided to put it into practice. This dissertation accepts this principle of separation, and develops a feasible model, which also reduces the software development time. The Advanced Evolutionary Prototyping Model proposed as a means in this thesis allows two teams of different experts to work concurrently on the same software project. The Advanced Evolutionary Prototyping Model, perceives the software development as two loosely coupled concurrent processes: the user interface process, and the computational process. Since well-defined methodologies which support the computational process, already exist, this thesis focuses only on the user interface process. The activities involved during the various phases of the computational process affect the user interface process only at the synchronization points in the Advanced Evolutionary Prototyping Model. The user interface process in the Advanced Evolutionary Prototyping Model, contains the analysis, design, implementation and evaluation phases. The various activities that are to be carried out during each of these phases of the user interface process are collectively named as a Concurrent User Interface Methodology (CUIM).

The Concurrent User Interface Methodology developed and presented in this thesis, supports concurrency in the software development process. The analysis phase in CUIM discusses how the user model and the system model are constructed. CUIM also discusses the verification process that is to be carried out at the end of the analysis phase.

The design phase in CUIM concentrates in achieving the requirements analyzed during the previous phase. This design phase models how the user’s view of the
interface can be constructed. The user model constructed during the analysis phase is refined to correspond to the dialog decisions made during the design of the user interface. The transformation steps from the user's view to the designer's view are described. The design phase in CUIIM also specifies the behavior of all the interface classes(objects) and the interaction relationship with other classes(objects) in the system. CUIIM also discusses the verification process that is to be carried out at the end of the design phase.

The next phase is the implementation phase. The output of the design phase serves as an input to the implementation phase. CUIIM specifies how the various design activities assist the user interface implementer in constructing the look of the user interface, in specifying the callbacks and in writing the code for the callbacks. We also discuss the verification process that is to be carried out at the end of the implementation.

The "Course Advising System" (CAS), is a software system developed by two graduate students, J. Barki and K. Duong. Its goal is to advise the undergraduate students in Computer Science course selection. We use CAS as a running example for discussing the various activities modeled in CUIIM. During the evaluation phase in CUIIM, we conducted user testing in order to evaluate the user interface that is developed following the CUIIM methodology. The test results obtained are also presented.

We therefore demonstrate how the application of CUIIM methodology,

• enables concurrent working in the software development process, and

• leads to a systematic procedure for designing user interfaces, which when supported by a development environment, could help in quicker development of software products.
1.4 Overview of the thesis

The goal of this thesis is to promote concurrency in the software development process by providing a systematic approach for the design and development of user interfaces. In chapter two, we survey the existing human-computer interaction research which focuses on models for developing user interfaces. The different methods the software engineers use for designing user interfaces are described in detail.

The third chapter deals with extending the concept of concurrent engineering for user interface design. This chapter introduces the Advanced Evolutionary Prototyping Model proposed, to achieve concurrency in the software development process. The advanced evolutionary prototyping model suggests a systematic procedure for developing prototypes. In order to put this model to practical use, we proposed the CUIIM methodology which is discussed in this thesis.

Chapter four presents the user interface analysis phase in CUIIM. This chapter discusses how the interactions between the user and the interface, and the interactions between the interface part and the computational part of the software system can be specified. The verification process that is to be carried out at the end of the analysis phase is also discussed.

In chapter five, the various design activities modeled in CUIIM are discussed. This chapter discusses how the appropriate dialog styles that satisfy the user are identified during the dialog design. The dialog between the user and the interface, specified during the analysis phase, is refined to correspond to the dialog decisions made during the design. The steps which translate the look of the user interface to the designer's view are described. The behavior of all the interface classes and the interaction relationship with other classes in the system are described next. The verification process that is to be carried out at the end of the design phase is also described in this chapter.

Chapter six is concerned with the implementation and evaluation phases in CUIIM. This chapter discusses how the various design activities assist the user interface implementer in constructing the look of the user interface, in specifying the callbacks
and in writing the code for the callbacks. The verification process that is to be carried out at the end of the implementation phase is also described in this chapter.

The evaluation phase in CUIM is discussed next. We conducted user testing of the interface developed, in order to evaluate it. The test results obtained are presented in this chapter.

A Hyper-media Design Tool (HDT) which supports the various activities in CUIM has been proposed. A short description of the features of the HDT tool are also given in Chapter 6. Finally, the conclusion of this thesis and some insights for future work are presented.
Chapter 2

User Interface Design Models

Awareness of the importance of human-computer interaction is spreading. Over the past few years, a number of significant advances have been made in the area of user interfaces. This Chapter surveys the existing human-computer interaction research which focuses on models for developing user interfaces. Since, user interface design models is the focus of this Chapter, a interpretation of the term model is given in the next Section. A quick overview of the various software process models is given in Section 2.2. Section 2.3 discusses how user interface design which was given a back seat in the past years, is now shifted to the front. This Section also reasons out why the skills required for user interface design are not the same as those required for software design. In Section 2.4, the different methods the software engineers use for designing user interfaces are described. Finally, the "Conclusion" Section at the end of this Chapter points out why the various models studied here are incomplete.

2.1 What is a Model?

A methodology is a process followed to build, deliver and evolve the software product, from the inception of its idea, all the way to the delivery and final retirement of the system. Methodologies can generally be classified according to the concepts involved and the way they organize the development effort. Therefore, we now define a process
model or a model as an abstraction of a methodology into various phases, where the output of each phase is presented as a document. The primary functions of a model are to determine the sequence and interaction of the various phases involved in the software development and evolution, and to establish the transition criteria for progressing from one stage to the next. These include completion criteria for the current phase plus choice criteria and entrance criteria for the next phase. Consequently, a methodology focuses on how to navigate through each phase and how to represent phase products (for example: structure charts, state diagrams etc..)

A methodology guides the software engineers in their work in all phases of software development; increases confidence in what they are doing, teaches inexperienced people how to solve problems in a systematic way, and by encourages a uniform, and standard approach to problem solving. It promotes a certain approach to solving a problem by preselecting the methods and techniques to be used for verification and validation during the software development process. Tools are developed to support the application of techniques, and support the methodology.

2.2 Software Process Models

Many new paradigms[Agre86] for software development have been proposed in the last decade. This Section captures the various software process models which enable us to produce software products.

1. The Code and Fix Model

The software production process in this model basically consists of two steps:

(a) Write code.

(b) Fix the code to eliminate errors, enhance existing functionality, or add new features.

The code-and-fix model has been the source of many difficulties and deficiencies. After a sequence of changes, the code structure becomes so messy that subsequent fixes would be very expensive, and the results become less reliable.
These problems underscored the need for a design phase prior to coding. Another reason behind the inadequacy of the code-and-fix model was the frequent discovery, after development of the system, that the software did not match the user's expectations. So the product either was rejected or expensively redeveloped to achieve the desired goals. As a result, the software development process was unpredictable and uncontrollable, and products were completed over schedule and over budget and did not meet quality expectations. This made the need for a requirements phase prior to the design and coding phases.

The failure of the code-and-fix process model led to the development of more structured models which make the development process predictable and controllable. The waterfall model's approach (described next), helped eliminate many difficulties previously encountered on software projects.

2. The Waterfall Model

The software development process for the waterfall model is structured as a cascade of phases, where the output of one phase constitutes the input to the next one. Each phase, in turn, is structured as a set of activities that might be executed by different people concurrently. The various phases involved in the waterfall model[Royc70] are:

(a) Feasibility Study
(b) Requirements Specification
(c) Design Specification
(d) Coding and Module Testing
(e) Integration and System Testing
(f) Delivery and Maintenance

The result of the requirements specification phase is the requirements specification document which documents what the analysis has produced. The result of the design specification phase is the design specification document, which contains the software architecture and a description of what each module is intended to do and the relationships among modules. The output of the coding and module testing phase is an implemented and tested collection of modules.
Verification is an independent activity of the waterfall life cycle. Appropriate verification is done at every stage on various kinds of activities and following suitable standard procedures. In most cases verification is performed as a process of quality control by means of reviews, and walk-throughs. Its goal is to monitor the quality of the application during the development process. A primary source of difficulty with the waterfall model is that, it emphasizes fully elaborated documents as completion criteria for early requirements and design phases. Since the waterfall model does not anticipate changes, whenever changes are required, the software engineers tend to make changes only by modifying the code, without propagating the effects of those changes to changes of the specifications. Thus, specification and implementation gradually diverge, thereby making future changes to the application even more difficult to perform. Also, updating the affected requirements and design specifications are difficult because they are usually textually documented, and changes are difficult to make and trace back. These concerns led to the formulation of the evolutionary development model[Boeh88].

3. The Evolutionary Development Model

The evolutionary development model consists of various phases which are expanding increments of an operational software product, with the directions of evolution being determined by operational experience. This approach consists of a step wise development, where parts of some stages are postponed in order to produce some useful set of functions earlier in the development of the project. Increments may be delivered to the customer as they are developed which is referred to as evolutionary or incremental delivery.

The development begins by analyzing an increment at the requirements level. Each increment is then separately designed, coded, tested, integrated, and delivered. Increments are developed one after the other after feedback is received from the customer. This model suggests that as users actually use the delivered parts, they start to understand better what they actually need. This leads to changes in the requirements for further increments and revisions of the original plan.
Figure 2.1: The Spiral Model (from Boehm '88)

In contrast to the waterfall model, a change may be taken into account very easily in the evolutionary model.

4. The Spiral Model

Another model that accounts for a structured development of a software is the spiral model[Boeh88]. The main characteristic of the spiral model is that it is cyclic and not linear like the waterfall model. Each cycle of the spiral consists of four stages(Figure 2.1): and each stage is represented by one quadrant of the Cartesian diagram. The radius of the spiral represents the cost accumulated so far in the process; the angular dimension represents the progress in the process.

Each cycle of the spiral begins with the identification of the objectives of the portion of the product under consideration, in terms of qualities to achieve (such as performance, functionality, ability to accommodate change, etc.). The alternative means of implementing this portion of the product (design A, design B, whether to buy, or reuse) are also identified during stage 1. Finally, the constraints imposed on the application of the alternatives (cost, schedule, etc.) is identified. In stage 2, the alternatives are evaluated relative to the objectives
and constraints. The different sources of risk are identified at this stage. Consequently, stage 3 involves the formulation of a cost-effective strategy for resolving the sources of risk. This may involve prototyping, simulation, benchmarking, etc.. After evaluating the risks, the next stage consists of planning for the next iteration of the spiral which is determined by the relative remaining risks. The risk-driven sub-setting of the spiral model steps allows the model to accommodate any appropriate mixture of a specification-oriented, prototype-oriented approach to software development. After each cycle of the spiral, unstated requirements are checked as part of the robustness of the application.

The spiral model may be viewed as a meta model, because it can accommodate any process development model. By using it as a reference, one may choose the most appropriate development model (for example, evolutionary versus waterfall model). It incorporates many of the strengths of other models and resolves many of their difficulties. However, this model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk. Suppose a team of inexperienced developers produces a specification with variation in levels of detail such that, a great elaboration of detail for the well-understood, low-risk elements, and little elaboration of the poorly understood, high-risk elements. Unless there is an insightful review of such a specification by experienced developers, this type of model will give an illusion of progress during a period in which it is actually heading for disaster.

2.3 User Interface Design versus Software Design

Until the past few years, developing an user interface for a software system has been thought of as a trivial job. Mayhew[Mayh92] points out that there exists a communication gap between "professional" software engineers and the "non-technical" users of software systems because the software engineers do not have the expertise in understanding the user needs. This communication gap is being carried over to the human-computer interface. The software engineer tends to judge his or her own work by criteria that may have little to do with the needs and constraints of the end user. For example, efficiency of code and flexibility of architecture may be admirable.
engineering goals, but may have little or nothing to do with the success of a computer system in being accessible to and supportive of a particular type of user. Even though software experts slowly learned that their logical structures and jargon are obscure and alien to non-professionals, user interface design was still given a back seat.

Since the user interface mediates between two main participants: the operator of the interactive system (the user) and the computer hardware and software that implement the interactive system, each participant imposes requirements on the final product. The operator is the judge of the usability and appropriateness of the interface; the computer hardware and software are the tools with which the interface is constructed. Consequently, an interface that is useful and appropriate to the operator must be constructed with the hardware and software tools available. Because of the complexity of both components, the construction of the user interface involves making many decisions about how to employ the tools available to best satisfy the user. Developing the user interface is further complicated by the fact that the customer may not have a complete idea of the requirements for the system being constructed and may have preconceptions about the interface that are expensive and difficult to implement. The software developers realized that the user interface accounts for approximately 50 percent of the total life cycle costs for interactive systems[Myer89], and the diverse use of computers in homes, offices, factories, hospitals, hotels, and banks has stimulated widespread interest in user interfaces. As a consequence of this, software professionals thought that a high-quality user interface is in fact important and user interface design which was given a back seat is now shifted to the front. This resulted in a vested interest in creating a user interface in a way it satisfies the customer and in using the best available tools and techniques.

Designing user interfaces is a complex and highly creative process that blends intuition, experience, and careful consideration of numerous human and technical issues. Any decision regarding the design of the user interface to specific functionality should be based on a sound and thorough knowledge of the user. The rationale behind "knowing the user" is that the designer will be able to decide what level of support the user requires. A conceptual model of the system should be explicitly designed and effectively presented through the user interface. The user interface designer should design the tools that most effectively fits the user. Since look and feel of the
user interface is an important aspect, the user interface designers should design the appearance of the user interface in a way that is both attractive and compatible with the operator's expectations. Evaluating the design provides the best predictor of the success of an interface. Different techniques such as, mock-ups and simulations are used to demonstrate the appearance and dynamic behavior of the user interface, which is quite different with respect to traditional software evaluation. Therefore, we see that the skills required for user interface design are not the same as those required for software design.

2.4 Designing the User Interface

This Section discusses the different methods the software engineers use for designing user interfaces.

2.4.1 Using Well-Defined Principles and Techniques

- The Xerox Star

Even though the article presented by Smith, Irby, Kimball, Verplank, and Harslem[Smit82] does not propose a formal model for developing user interfaces, it presents some principles and techniques about how and how not to design user interfaces.

Most of today's software developers recognize the importance of focusing on user interfaces early in the development process; but this was not the case ten years ago. Therefore, we can point out that the most significant decision made by the developers of the Star was to focus on the user interface "before" the rest of the software was developed. The following are the main goals that were pursued in designing the Star user interface[Smit82].

1. Familiar user's conceptual model: The first task that has been carried out during the design process is to decide what model is preferable for users of the system.
2. Seeing and pointing vs. remembering and typing: The Star user interface does not hide things that burden the human short term memory.

3. What You See Is What You Get (WYSIWYG): Star adheres to this principle by displaying documents which include typographic features such as boldface, italics, superscripts, and layout features such as embedded graphics, page numbers, headers, and footers.

4. Consistency: The Star user interface maintains consistency by providing similar mechanisms whenever they occur. For example, the task of selecting a set of characters from a textual object and the task of selecting a line from a graphical object is done in a similar fashion.

5. Simplicity: The Star user interface promotes simplicity by making the system uniform and consistent and by minimizing the redundancy in the system.

6. Modeless interaction: In the Star user interface the object (noun) is almost always specified before the action (verb) to be performed. This helps make the command interface modeless.

7. User tailorability: This system is designed with provisions for user extensibility.

Prototyping has taken on a fundamental role in user interface development [Maso83]. The design of the Star user interface also focused on the importance of prototyping. While many of these principles are commonly agreed among today’s human-computer interaction research community, they were not as well known when the Star was being developed. However, the developers of the Star, particularly in their succeeding article [John89], emphasize that these principles were appropriate for their system’s needs. Systems with similar users and capabilities could benefit from these guidelines. Therefore, we can conclude that the design of the Xerox Star system is based on a systematic strategy which is based on principles of good human-computer interaction. The Star is a good example to keep in mind during the user interface design process.
2.4.2 Mimicking the Existing Software Process Models

Draper and Norman present a number of important insights into user interface development [Drap85]. The overall theme of this article is that lessons learned in software engineering may prove useful in addressing challenges in user interface design. It is difficult to perform a formal analysis on a user interface because it is hard to develop a complete set of requirements. Therefore, more empirical methods of testing and benchmarking are required. Due to the fact that the quantitative principles upon which one can predict design decisions are not well known, an iterative strategy/prototype modeling must be adopted with emphasis on the testing and validating between phases. The user-program interaction needs testing by exercising each possible "branch" of the interaction. Since, test procedures for user interfaces do not exist, development of good test plans, of a good pool of users upon whom the tests will be run, and careful observation and evaluation of the results are necessary.

One of the models proposed in the literature, for constructing the user interface is the GOMS model. This model mimics the prototype model specified in traditional software engineering.

- The GOMS Model

In [Kier88] Card, Moran, and Newell propose a model for describing the user's cognitive structure. It is a representation of the "how to do it" knowledge that is required by a system in order to get the intended tasks accomplished. GOMS model basically consists of the following:

- a set of Goals
- a set of Operators
- a set of Methods needed to accomplish the specified goals and
- a set of Selection rules for choosing the appropriate method for achieving the goal

**Goals:** A goal is something that the user tries to accomplish. The analyst (a person who constructs the GOMS model) identifies and represents the goals
that typical users will have. A set of goals usually will have a hierarchical arrangement in which accomplishing a goal may require first accomplishing one or more subgoals.

**Operators:** Operators are actions that the user executes. The behavior of the user is described as a sequence of these operators.

**Methods:** A method is a sequence of steps that accomplishes a goal. The definition of a method includes the fact that it is known prior to the initiation of the task.

**Selection:** Selection is the process of choosing one method over another. A set of rules are described such that the appropriate method is selected.

GOMS task analysis allows the analyst to repeatedly make decisions about how users view the tasks in terms of their goals and how they decompose a task into its subtasks. The authors suggest that the analyst should make judgment calls during the task analysis, because it is not possible to collect data on how users view and decompose the tasks. By starting with listing the user's top-level goals, then defining the top-level methods for these goals, and then going on to the sub-goals and sub-methods, the analyst will be in a position to make decisions about the design of the user interface directly in the context of what the impact is on the user. Once the methods are written, this model suggests that the analyst choose some task instances to check the accuracy of the methods. The GOMS model so constructed represents different aspects of the implementation of the user interface.

Taking the Goals, Operations, Methods, and Selection rules described during the GOMS analysis, the interface is constructed. The user interface developed is then tested and the various methods the user uses to accomplish a task are observed/recorded. After getting the feedback from the user, the GOMS analysis is iterated to match the user actions.

The user interface designer can make use of the tasks modeled by the analyst, when prototyping approach is followed during user interface design. This model is also useful for predicting times for user tasks.
2.4.3 Using Ad-hoc Methods

User interface design is also carried out by many people using ad-hoc methods. In many such situations, the design of the user interface is combined with software design. One of the models discussed in the literature which falls into this category is the model proposed by Sutcliffe and McDermott [Sutc91].

- The Structured Analysis/Structured Design Method

Sutcliffe and McDermott propose a method for user interface design which builds on structured analysis/structured design (SA/SD) [Pete87]. This method uses SA/SD notations for interface specification and works from requirements analysis to detailed design.

The analysis phase consists of three activities: analyzing the user tasks, describing the user characteristics, and analyzing the user views. The authors say that the objective of describing the user characteristics is to obtain a thorough knowledge of individual users to predict how they may react to tasks and interface design styles with different complexities. They point out that the user characteristics influence the strategic choice of the interface. The user's structural knowledge of the system and its external appearance are captured during the user view. During the analysis phase, the system is decomposed into functions, or discrete pieces of work which achieve a given goal. Then, a Data Flow Diagram (DFD) portraying the map of the system is built by linking the functions (portrayed as circles) with the data connections, called data flows which are illustrated as connecting arrows. The specification phase ends with choosing the interface design styles in light of the requirements and characteristics of the target users. At the end of this stage a review of the specifications is done.

Task analysis is followed by task allocation and design. This method uses Structured English Descriptions [Pete87] for specifying task allocations. The Structured English Descriptions express processes in simple English phrases composed of nouns for data, and verbs for actions. These descriptions list the tasks that are allocated to human, to computer, and to both. Task design aims to match task demands to the user's abilities.
The final phase is the detailed design. At this stage, DFDs are transformed into structure charts following the procedures of Structured Design[Pet87]. The structure charts depict the architecture of the software system. During the detailed design phase, the Structured English Descriptions from the design phase are mapped to dialog sequences. The mapping process is as follows:

- Transform the Structured English Descriptions into a dialog diagram with the mapping:
  \[
  \text{Input actions} = \text{transition arcs} \\
  \text{Output actions} = \text{diagram nodes (states)}
  \]

- Each dialog is elaborated to include control and support arcs, for example: prompt and error messages, defaults, help and escape facilities.

- Sub-dialogs are joined together and the higher level control dialogs are added by creating a hierarchy of dialog-interface modules.

The interface module hierarchy diagram so constructed shows the executable sequence of screens and their dialog sequences. They are verified with users.

Distinguishing the user interface design from the application design is recommended by [Bil88], [Drap85], [Hill86], and [Radh93]. But the model proposed in [Sutc91], does not provide a clear cut division between the two. Also, this model does not support iterative design of user interfaces, which is necessary[Drap85] for user interface design.

### 2.4.4 Other Existing Models for User Interface Design

The key issue for user interface development is modeling the user. None of the models discussed in the previous subsection capture this. Therefore this subsection focuses on some of the models given in the literature, which give priority in modeling the user.

*User Interaction* Models are models of the components of a user interface based on user's cognitive interactions with these components. The authors[Norm84],[Fole82]
claim that proposing models of how users interact with the different components of the user interface helps us to gain better understanding of how to develop user interfaces. Each of these models propose a somewhat different approach for understanding user’s interactions. The Stages and Levels model, and the Linguistic model which are described below, correspond to User Interaction models.

All the activities which occur during the lifespan of a software system, starting with project initiation and ending with system replacement constitute a life cycle[Cut88]. Extending this concept to the area of user interfaces we can define the User Interface Life Cycle Models as, those models that describe all the activities which correspond to the user interface of a software system. The Star Life Cycle model is also included in the following:

- The Stages and Levels Model

This model concerns with the overall process of interaction with the computer. In [Norm84], Donald Norman states that the interaction between the user and a computer system involves four different stages of activities. Norman proposes that the full cycle of stages for a given interaction involves:

1. Forming the Intentions: Norman defines intention as the internal, mental characterization of the desired goal. The intention is what the user wants to accomplish. He says that forming intentions may not even be a conscious activity.

2. Selecting an Action: Selection is the mental state of determining what are the actions that will be used to accomplish the task. There should be some list of available operations from which to choose. Once selected, this process includes the determination of a particular command or a sequence of commands in order to initiate the operations.

3. Executing the Action: The process of entering the selected commands into the computer is execution. This implicitly suggests that one of the commands informs the computer that the rest of the commands should be processed. Norman stresses that intention and selection are mental activities, whereas execution is a physical activity which involves entering information into the computer.
4. Evaluating the Outcome: Norman observes that the results of the actions need evaluation, and that evaluation is used to direct further activity.

Each of the different stages described above, facilitates the user interface designer to concentrate on different aspects of the user interface. An inference of how the stages proposed by Norman leads for user interface design is given below.

The initial stage of *forming intentions* allows the user interface designer to concentrate on issues like “What is possible?” Given the system facilities and the current status of the system, the designer can design the various possible intentions the user can have.

The *selection* stage helps the designer to decide what information (the different commands available and so on) should be made available to the user. This stage facilitates the designer to choose either menus or help commands which allow the user to determine the possible commands.

The *execution* stage facilitates the designer to focus on the types of input modes to be provided to the user. The designer concentrates on issues like:

1. What is the form of command language (if command language is chosen as the input mode)?
2. How are ill-formed sequences to be handled?
3. How much support should be provided for the user at this stage?

Finally, the evaluation stage facilitates the designer to tackle with issues of how to provide feedback to the user by informing him whether the operation has been completed successfully or whether it has failed.

Even though the model proposed by Norman cannot be thought of as a complete process for user interface design, as seen above, the different stages of activities described in his model provide implications to the design stage.
• The Linguistic Model

The authors Foley and Van Dam[Fole82] define the user-computer interface as being composed of two languages: one supporting the user communicating with the computer, and the other which supports the computer communicating with the user. They suggest that the first is expressed with actions applied to various interaction devices, while the second is expressed graphically to form displayed images and messages. They suggest that there are four major activities which must be addressed in order to completely define a user-interface. The four major activities are:

1. The Conceptual Design: This can be thought of as the user’s model. It consists of the key application concepts which must be mastered by the user. The conceptual design activity is composed of objects, relationships between the objects, and operations on the objects. As a simple example, the author uses a text editor. In the text editor, the objects might be the lines and the files. One relation is that the files are sequences of lines. Operations on line objects include insert, delete, move and copy. Operations on file objects include create, remove, duplicate, and rename.

2. The Semantic Design: The detailed functionality of the system is defined by the semantic design. This describes what information is needed to perform each operation, what are the different errors that could occur, and what are the side-effects that may be produced by each operation. Foley and Van Dam points out that this design activity involves defining meanings. It does not identify the sequence of how to perform particular operations.

3. The Syntactic Design: The sequences of inputs and outputs are defined during the syntactic design. Syntactic inputs include lists of tokens needed for a particular operation to occur. Syntactic design defines how to construct correct sentences. Whether the sentence has any semantic meaning is a “semantic” design issue. Syntactic outputs include symbols and drawings as well as sequences of characters. They also include spatial information.

4. The Lexical Design: The actual hardware primitives which are necessary
to build tokens are specified during the lexical design. Input devices such as keyboard/mouse provide for forming the input tokens. Output consists of whatever is available on the particular hardware, such as characters or graphics.

Through various activities that are involved during the design, this model addresses different aspects of the same design but from a different level of abstraction and detail.

Both the user interaction models discussed above provide a structure for the components of an user interface to be developed. This structure is based on a human-centered model of the activities involved in the interactions between humans and computers. Even though human factors is a focus for design of the user interface, none of the User Interaction models discussed before provide a well-defined process/methodology which guides the development of the complex structure of an user interface. All the User Interaction models achieve one particular purpose (which is, understanding user's intentions) without addressing the wider aspects of systems development. Therefore, they do not cover the whole systems development life cycle.

- The Star Life Cycle Model

Hartson and Hix[Hart89] say that the life cycle for interface development should not "naturally" follow the traditional software development life cycle, with its top-down, linear sequence of somewhat isolated activities for requirements, design, implementation, and testing. The authors comment that the attempt to impose the classical "waterfall" paradigm[Ghez91] on interface development are undoubtedly the cause of many bad interfaces.

The authors hypothesize that their interface development life cycle most naturally occurs in "alternating waves" of two kinds of complementary activities:

1. Typical early activities of interface development are bottom-up, based on concrete dialog scenarios, often augmented with state diagram like representations which provide a direct representation of logical sequencing of end-user navigation among screens. Scenarios and state diagrams are
translated into supervised flow diagrams which show both control flow and data flow.

2. Subsequent activities involve top-down, step-wise decomposition and structuring. Activities that are bottom-up, synthetic, empirical and related to end-user’s view alternate with activities that are top-down, analytic, structuring and related to a system view. These two kinds of development activities reflect different kinds of mental modes, which the authors call as “synthetic” mode and “analytic mode”. An interface developer may alternate between these analytic and synthetic modes of mental activity several times within a single phase of development activities and within a short time period.

The different activities involved in the Star life cycle model are listed below:

1. Task Analysis/Functional Analysis
2. Requirements/Specifications
3. Conceptual design/Formal design representation
4. Prototyping
5. Implementation and
6. Evaluation

Since rapid prototyping is a key to support evaluation and iterative refinement, this model supports rapid alternation between prototyping and other development activities, especially design. Output of the design activity is used during this phase to produce executable program code. The evaluation stage is the very heart of the Star life cycle and necessitates different kinds of support tools at different times during the development activities. For example, support needed to evaluate designs is different from that needed to evaluate the real application system after implementation. The authors suggest that UIMS support for evaluation of application system design should include traceability to relate the design to the requirements, and even some very early prototypes to test out various interaction styles. The authors say that tools that capture log data files of end-user activities during interaction with the system are also needed to support evaluation after application system implementation.
2.5 Conclusion

We observe that the user interface design and development should be done throughout the software development process. Software engineers realized that the user interface design should start when the project starts, and should not be done as a patch work in the later phases of software development. The various methods software engineers use for designing the user interfaces are described in this Chapter.

In the GOMS model[Kier88] proposed by Kiers, once the analyst completes writing the methods, he is required to choose some task instances to check the accuracy of the methods. This checking is done by executing the methods using hand simulation, and noting the actions generated by the method. These sequence of actions are then verified manually to see whether they are the correct ways to execute the tasks. If the methods do not generate correct action sequences, the methods are modified such that they execute correct task instance. But this type of verification would be very tedious for large projects. Also, specifying a step-by-step description of the methods as a textual description makes the model cumbersome. In the method proposed by Sutcliff and McDermott[Sutc91] the authors did not make it clear of how the specifications during the analysis phase are reviewed. Also, no indication is given about how the detailed design is reviewed. Mockups and reviews play an important role[Bass90] in user interface design, but the model proposed by Foley and VanDam[Fole82] does not suggest any reviews to be done during the various design activities. Even though, the Star life cycle[Hart89] deals with different issues at various stages of the life cycle, this model does not try to model the user domain which is very important for user interface design[Mayh92].

Despite the advances in human-computer interface development, there exists only a very few methodologies which guide the design and development of the user interface according to a systematic procedure. Even, these methodologies lack to link the development of the computational (non-interface) part of an application system with that of the user interface. Although the model proposed by Sutcliffe[Sutc91] represents the whole system (user interface component and the functional component), it does not distinguish the design of the user interface from the application design.
But many researchers [Bilj88, Drap85, Hill86, Radh93] suggest that the user interface should be distinguished from the application for reasons such as modularity, reuse, and for rapid development of software systems. Therefore, issues such as separation of concerns and concurrency of user interfaces are the main focus for Chapter three.
Chapter 3

Concurrent Engineering - An Overview

In future, the success of a software system will be the result of understanding the user needs, and developing an user interface to meet those needs. We will need powerful functionalities, but a simple, and clear interface. We want ease of use, but also ease of learning. Interface designers find themselves constantly confronted with these kinds of conflicting goals. Methods and techniques are needed to help interface designers to effectively manage the design and make good design decisions for a given product with regard to its users. When the competitive climate of an entrepreneur changes rapidly, product development time also becomes more critical. Getting a quality product to market fast is the name of the game. This chapter introduces a concurrent methodology for the development of a user interface design. It is our belief that the concurrency introduced in the software development life cycle, will help to reduce the product development time.

This chapter starts by giving a definition of concurrent engineering. Section 3.2 lists the advantages of extending the concept of concurrent engineering for user interface design. In section 3.3, the methodologies that support concurrency in user interfaces are discussed. The evolution of the concurrent user interface methodology CUIDM, is also discussed in this section. Section 3.4 captures the entry point in the
Advanced Evolutionary Prototype model. In section 3.5, different dialog specification models used in CULM are described. The summary of this chapter is given in Section 3.6.

3.1 A Definition

Concurrent Engineering is a systematic approach for integrated product development that embodies different teams working at the same time and cooperatively, towards accomplishing a common goal. The phrase “Concurrent Engineering” usually applies to human work[Huyn93]. Concurrent Engineering aims for improving the logistics within a development process, making sure that the right things are done at the right time, by the right people. Concurrent development can be thought of as an advanced form of software-factory approach[Aoya93]. The conventional factory approach focuses on productivity and quality with little attention to the development cycle time, and the total cost over multiple cycles. In contrast to this, concurrent development’s systematic way of developing software systems shortens the development cycle.

3.2 Extending the Concept to User Interfaces - Advantages

The existing paradigms for user interface development, discussed in chapter 2, are basically sequential; the development activities take place sequentially. For concurrent engineering, this trend has to be changed; the user interface and application development activities must be performed concurrently. The following justifications apply:

- Expertise in the development process: From chapter 2 we see that, the skills required for user interface development are different from that required for software development. It is not very common to find the expertise required
for user interface design and that for the application design to be in one person. This kind of job specialization is likely to occur more and more in the future. The concurrent engineering methodology permits different groups of designers to inter-work systematically in the development process. Draper and Norman[Drap85] observe that some of the successful interface designs (for example, the Xerox Star and the Apple Lisa) have been developed by teams that worked exclusively on user interfaces.

- **Ease of construction:** In contrast to the sequential development process, the concurrent-development of the user interface and the application processes allow us to divide the software into two major subsystems and incrementally construct each of these subsystems.

- **Coping with changes in requirements:** As discussed in chapter 2, the customer himself may not have a complete idea of the user interface requirements for the system being constructed. Therefore, the requirements of a user interface keep evolving. Since concurrent engineering of user interface and the application promotes separation of concerns, and the coupling between the two subsystems being low, coping with changing requirements would become easy.

- **Length of development cycle:** Concurrent Engineering in user interfaces leads to shortening the software development cycle. Mikio[Aoya93] claims that when compared to sequential development time, there was a 75% reduction in development time when they applied concurrency for developing a software system. By separating the user interface from the application development and incorporating concurrency in these two developments, we can hope to reduce the software development time.

### 3.3 Concurrent Engineering in User Interfaces

Researchers[Drap85, Wass85] in the area of human-computer interaction proposed some strategies that support concurrency in user interfaces. The various strategies and the methodologies that support the separation of user interface design from the application program design are discussed below.
3.3.1 Methodologies that Support Concurrency in User Interfaces

- Modular Decomposition

Draper and Norman[Drap85] present a strategy for separating the interface design from the program design. This strategy views a software system into three sets of modules: program as one set of modules, the user interface as another set, and the user comprising the third set. The virtue of this view is well known in the software engineering community; namely, as long as the communication structure and the data representation are well specified and adhered to, the modules can be developed more or less independently and changed without too much effect on the rest of the system. They propose that one might consider a user interface to be "run" on a human being analogous to the way that other software programs are run on a computer. The authors propose that the user and the program parts of the user interface can be thought of as coroutines each communicating with one another. The communication between the Interface and the User can be changed in any arbitrary manner as long as the communication between the Program and the Interface is not affected. This requires, that the only part of the system that may interact directly with the user is the Interface Module. A well-defined protocol between program modules and interface modules can be defined at the outset, leaving the interface designer free to change the interface without interfering with the independent development of program modules.

The authors suggest that when evaluating or debugging a user interface program, not only must the software part be debugged, but so must the interactions between the user and the system. This can be especially challenging since it is quite difficult to obtain a full specification of the user which, in turn, means that it is difficult to design to a user's defined behavior.

In this article, the authors present two important observations:

- New languages would be beneficial for representing the dialog between the user and the computer.
- Developing documentation for the user of a system is part of the process of developing a user interface. The authors point out that the term "documentation" should not be viewed in a fine grain fashion. Necessary information should be given to the user from a number of sources which include: normal displays, error messages, manuals, and tutorials.

Even though this article does not provide a holistic methodology for designing user interfaces, the suggestions and observations made by the authors are important to keep in mind.

- USE Methodology

User Software Engineering (USE) methodology developed by Wasserman, Pircher, Shewmaker, and Kerstern [Wass85] is based on the prototype model for developing software systems. The key aspect of the USE methodology is the ability to rapidly create system prototypes, presenting user view of the evolving system. The authors emphasize on the user interaction portions of the system. They say that the user interface provides the user with a language for communicating with the system. The interface can take many forms, including multiple choice menu selection, a command language input, a database query language, or natural language-like input. In all cases, however, the normal action of the program is determined by user input, results, requests for additional input, error messages, or assistance in the use of the system.

This methodology focuses on the analysis, design and implementation phases of the development process. The authors use a modified form of transition diagrams for specifying the user interface of the system to be developed. Each node in the transition diagram correspond to a machine state and the output displayed at that point, and the various arcs from a node correspond to alternative user inputs anticipated in that state. The analysis step serves to identify the major functions, and the required inputs and outputs. The concern of the design phase is to determine how the user can request those functions and how the output will be displayed. The third step in the USE methodology is the creation of an executable version of the user interface defined during the design stage. The executable version is then jointly explored both by the developer and the user. Accordingly, modifications are made to the original design, and
the analysis, design and implementation phases are iterated until one or more acceptable interfaces are found.

To create an executable version of the user interface, a system called RAPID/USE has been developed. This system consists of two components: the Transition Diagram Interpreter (TDI) and the Action Linker [Wass82]. The TDI was designed to accept an encoding of the USE transition diagrams, and this encoding, is called dialog description. The dialog description can be produced in either of the following two ways:

1. Draw the USE diagrams by using a graphical tool, called the transition diagram editor [Mill84]. Once the USE diagrams are drawn, the editor automatically generates the "dialog description".

2. The textual description of the USE diagrams is manually re-written into a "dialog description", that can be given as input to the TDI.

Input to TDI consists of one or more dialog descriptions, each description represented as a transition diagram.

The Action Linker part of RAPID/USE allows programmed actions to be associated with the transitions. This linker provides linkage to routines written in C, Fortran, or Pascal. Thus, RAPID/USE is used both for building and validating user interfaces (TDI). However, this methodology is not a complete solution for concurrent user interface design/development due to the following reasons:

1. The different dialog styles chosen to communicate with the user, affect the ease of use and ease of learning of the user interface developed [Mayh92]. But the USE methodology, does not follow any systematic procedure for designing the dialog styles which are appropriate to the users of the system.

2. It does not propose any verifications to be done at the end of each phase, while developing the user interface.

3. At the end of the user interface design, the function calls (represented in the form of rectangles), are substituted by the appropriate program modules. In this methodology, no provision is made for verification between the user interface modules and the program modules.
3.3.2 Evolution of CUIM

The Concurrent User Interface Methodology (CUIM), which is developed and discussed in this thesis, provides a systematic approach for concurrent engineering of user interfaces. The design methodology CUIM, supports the separation of the user interface subsystem from the program subsystem and serves in reducing the software development time.

CUIM is based on the Advanced Evolutionary Prototyping Model of software development. The traditional evolutionary prototyping model [Ghez91] is modified and coined as Advanced Evolutionary Prototyping Model. Its main goal is to promote two different sets of people working simultaneously on program development and user interface development. The various development activities that are sequential in the traditional model are carried out concurrently in the Advanced Evolutionary Prototyping Model. The Advanced Evolutionary Prototyping Model consists of the following distinct phases:

1. User Interface Analysis
2. User Interface Design
3. User Interface Implementation and
4. User Interface Testing

The outcome of each phase is marked by a clearly stated set of documents. At the end of each phase we ensure that the various documents produced are consistent.

In the evolutionary prototype model [Ghez91], several prototypes are developed. These prototypes do nothing more than display the interfaces on the computer screen and activating "dummy functions" when specific services are requested by the user, during interaction with the system. The different prototypes developed might differ in the layout of interfaces, in the sequences of possible operations performed and so on. Some of these will be throwaway prototypes, but some may be chosen to be evolutionary. Once, the user has selected the preferred prototype, the
program modules that accomplish what is requested by the various functions that are to be activated as a consequence of user interaction are designed and developed. The functions developed are then linked with the interfaces developed before. Thus, the prototype gradually evolves into the final system. The Advanced Evolutionary Prototyping Model differs from the existing model in the following two ways:

1. Rather than developing the prototypes by trial and error, the Advanced Evolutionary Prototyping Model suggests that a systematic procedure for developing prototypes should be followed. Towards this end, the Advanced Evolutionary Prototyping Model incorporates the analysis and design stages prior to the development of the prototype. The advantage of doing this is, the analysis and design stages reduce the number of prototypes to be developed. Therefore we can hope, an interface which satisfies the user can be developed in shorter time.

2. The steps of designing and implementing the program modules are carried out concurrently with the developing and testing of the user interface modules.

In the Advanced Evolutionary Prototyping Model, the user interface development life cycle and the program development life cycle are perceived as two concurrent processes: the user interface process and the computational process. A view of these processes is depicted in Figure 3.1. The objects associated with the two processes will be referred to as Interaction Objects (IOBs) and Computational Objects (COBs) respectively. The group of software engineers associated with the user interface process will be referred to as the Interface Group and the group associated with the computational process will be referred to as the Computational Group.

In Figure 3.1, an ellipse denotes a phase of development whereas a rectangle represents the output of the preceding phase. Given the specifications of the product to be developed, by the customer, the software engineers (Interface Group + Computational Group) prepares the Informal Requirements Document (IRD). This document is nothing but a clearly stated SRS (Software Requirements Specification). The IRD acts as a bridge between the customer and the software engineers. The IRD serves as input to the analysis phase of both the user interface process and the computational process. The output of the interface analysis phase is the Interface Analysis
Figure 3.1: Advanced Evolutionary Prototyping Model
Document (IAD), and the output of the interface design phase is the Interface Design Document (IDD). CAD and CDD corresponding to the computational process represent the Computational Analysis Document and the Computational Design Document respectively. From Figure 3.1, it is obvious that the output of one phase serves as input to the next phase. The refinement of the activities carried out during the analysis phase, to match the design decisions, is implied, and is not shown explicitly in Figure 3.1. The various activities involved during the analysis and design phases of the computational process affect the user interface process only at the synchronization points in Figure 3.1. These points mark the step at which the user interface group and the computational group must interact, to ensure consistency between the user interface part and the computational part of the software system. Unless otherwise specified, the various phases discussed later on correspond to the user interface process.

Since well-defined methodologies which support the computational process, already exist, this thesis concentrates only on the user interface process. The activities involved during the analysis and design stages of the user interface process, are modeled in CUIM. CUIM aims at providing complete coverage of the user interface process, by giving guidance from the analysis phase down to implementation and testing. By applying the CUIM methodology for user interface design and development, the user interface engineers could concentrate on different aspects that lead to produce user interfaces that promote ease of use and ease of learning.

3.4 Applying the Advanced Evolutionary Prototyping Model

We describe the Concurrent User Interface Methodology (CUIM) using an example. Since our main concern is designing the user interface, an interactive application is selected. The interactive Course Advising System (CAS) has been chosen for this purpose. Following the advanced evolutionary prototype model, the first step is to prepare the IRD document for the application chosen. The informal requirements of CAS are specified below.
3.4.1 General Description

The CAS system, is intended to advice undergraduate students in selecting those courses that are offered during the academic year, towards achieving their degree. The UI to the advising system is to be designed in such a way that it mimics the existing advising system, where ever possible. The rationale for doing so is to let users encounter minimal changes in the use of this new service. The various features the system provides are discussed in the IRD. Taking the requirements (user domain, functionality etc..) presented here, the UI to the advising system will be designed.

The expected user population for CAS is described in section 3.4.2. The various modes of interaction with the system are specified in section 3.4.3. In section 3.4.4, the current advising procedure is discussed. In Section 3.4.5, the functional requirements of the interactive advising system are described. Section 3.4.6 discusses the hardware and software environments for the course advising system.

The style and presentation of an IRD is not fixed. This document should give a clear picture of the system to be developed, by describing the ways that the users follow currently to do the job, the knowledge that the system should have before it is initiated, and so on.

3.4.2 User Profile

The general characteristics of the users of the course advising system include people with: at least a high school degree, computer literacy ranging from low to high, no prior training in the use of CAS, low frequency of use, and discretionary use of the system. Therefore, the user domain includes people with different levels of motivation (high/low due to fear, high due to interest). Since we cannot control the level of motivation of the users, we aim at an interactive system that minimizes the negative emotions of fear, boredom, and the like; and maximizes job satisfaction, and thus motivation. Towards this end, the user interface should be easy-to-use and easy-to-learn. The users for this system are the students. So, the term “students” will be used synonymously for “users”. For conciseness, the word “his” is used instead of “his/her”.

39
3.4.3 Modes Of Interaction

1. Input Modes: The different modes through which the user can interact with the system are:

   - Keyboard and Mouse: A student uses the mouse pointer for selecting various options given by the system. The user types in the choices (course number, time, and so on) using the keyboard.

2. Output Mode: The mode through which the system responds to the user is:

   - Display Screen: The system uses the display screen for displaying all the information and messages to the user.

The modes of interaction can be enhanced by providing features such as audio, touchscreen etc.. Considering the issues such as feasibility and cost, the above modes are chosen for I/O.

3.4.4 Current Advising Procedure

At the beginning of every academic year (Fall Semester), every student receives a registration form, the student's up-to-date transcript, the course schedule booklet, and an appointment card to consult an advisor.

- **Registration Form:** The different sections the registration form contains are as follows:

  - Student Information: In this section the student has to fill in his name, ID, and the program which he is registered in.

  - Course Information: In this section, the student lists all the preferred courses. The student has to fill in the course code, the course number, the term in which he wants to take the course and the section he is willing to attend.

  - Approval Information: This section is to be filled in by the advisor during the advising session. In this section, the advisor indicates his approval of the courses listed in the registration form.
• **Transcript:** The *transcript* contains the following information: At the top is the name and mailing address of the student. In the top right hand side is the ID #, the date of birth information and the home phone number of the student. Below the name and mailing address is a tabular section which contains information describing to which program the student is admitted to, the option name, minimum credits required for achieving the degree, the basis of admission, and the prerequisites (if any) the student should take. Below this, is the list of courses the student has completed. The list describes the following: the course name, the course number, the term the course was taken, the section, course credits indicating the number of credits the course is entitled to, the grade obtained in this course, and the credits granted. Below the course information is the program status section. This section indicates the program name, option, minimum credits required for achieving the degree, credits completed, student status (permanent resident or visa student or independent student), and the GPA since entry into program. A sample copy of the transcript is shown on the next page.

• **Course Schedule:** The *course schedule* booklet contains information pertaining to various courses that are offered by each department during the academic year. It includes a list containing the course number, the course name, the different sections available for this course, the time of day when this course is offered, the professor’s name, and the department that is offering this course.

• **Appointment Card:** The *appointment card* contains the date, the time and the location the student is supposed to be present for registration.

As a first step, the student fills in the form by listing his preferred courses, satisfying all the prerequisites and co-requisites for the course(s) listed towards accomplishing the degree and with no conflicts in the course timings. Not all students fill their registration forms:

• Some students in their first year do not know how to select the courses.

• Some students are not clear of the courses they should take and cannot come to a conclusion in their course selection.
**STUDENT RECORD**

**COURSE**

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>NAME &amp; NO</th>
<th>SECT</th>
<th>CR HRS</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*ADMITTED TO* 01/09/90

**B. COMP. SC.**

**OPTION SOFTWARE SYSTEMS**

**MIN. CREDITS REQUIRED : 90.00**

**ON BASIS OF: COLLEGE OF ED/DAWSON 1985-90**

**MUST TAKE WITHIN PROGRAM PHYS 205 PHYS 225**

**FALL-WINTER 90-91**

<table>
<thead>
<tr>
<th>COURSE DESCRIPTION</th>
<th>NAME &amp; NO</th>
<th>SECT</th>
<th>CR HRS</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLLEGE OF EDUCATION 1985-87</td>
<td>2.00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PROGRAMMING METHODOLOGY**

<table>
<thead>
<tr>
<th>COURSE DESCRIPTION</th>
<th>NAME &amp; NO</th>
<th>SECT</th>
<th>CR HRS</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP 231</td>
<td></td>
<td>/2</td>
<td>V</td>
<td>3.00</td>
</tr>
<tr>
<td>COMP 244</td>
<td></td>
<td>/2</td>
<td>T</td>
<td>3.00</td>
</tr>
<tr>
<td>COMP 291</td>
<td></td>
<td>/2</td>
<td>VV</td>
<td>1.00</td>
</tr>
<tr>
<td>MATH 242</td>
<td></td>
<td>/2</td>
<td>01</td>
<td>3.00</td>
</tr>
<tr>
<td>MATH 262</td>
<td></td>
<td>/2</td>
<td>01</td>
<td>3.00</td>
</tr>
<tr>
<td>MATH 282</td>
<td></td>
<td>/2</td>
<td>51</td>
<td>3.00</td>
</tr>
<tr>
<td>COMP 244</td>
<td></td>
<td>/4</td>
<td>02</td>
<td>3.00</td>
</tr>
<tr>
<td>COMP 291</td>
<td></td>
<td>/4</td>
<td>X</td>
<td>1.00</td>
</tr>
<tr>
<td>MATH 243</td>
<td></td>
<td>/4</td>
<td>01</td>
<td>3.00</td>
</tr>
<tr>
<td>MATH 263</td>
<td></td>
<td>/4</td>
<td>02</td>
<td>3.00</td>
</tr>
<tr>
<td>MATH 283</td>
<td></td>
<td>/4</td>
<td>AA</td>
<td>3.00</td>
</tr>
</tbody>
</table>

**FALL-WINTER 91-92**

<table>
<thead>
<tr>
<th>COURSE DESCRIPTION</th>
<th>NAME &amp; NO</th>
<th>SECT</th>
<th>CR HRS</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP. ORG. &amp; ASSEMBLY LANG.</td>
<td>COMP 220</td>
<td>/2</td>
<td>TT</td>
<td>4.00</td>
</tr>
<tr>
<td>LANGUAGE LABORATORY - FORTRAN</td>
<td>COMP 293</td>
<td>/2</td>
<td>S</td>
<td>1.00</td>
</tr>
<tr>
<td>INTRO-THEORETICAL COMPUTR SC</td>
<td>COMP 335</td>
<td>/2</td>
<td>V</td>
<td>3.00</td>
</tr>
<tr>
<td>MATH 271</td>
<td></td>
<td>/2</td>
<td>A</td>
<td>3.00</td>
</tr>
<tr>
<td>MATH 351</td>
<td></td>
<td>/2</td>
<td>01</td>
<td>3.00</td>
</tr>
<tr>
<td>COMP 298</td>
<td></td>
<td>/4</td>
<td>Z</td>
<td>1.00</td>
</tr>
<tr>
<td>COMP 352</td>
<td></td>
<td>/4</td>
<td>XX</td>
<td>3.00</td>
</tr>
<tr>
<td>MATH 361</td>
<td></td>
<td>/4</td>
<td>V</td>
<td>3.00</td>
</tr>
<tr>
<td>ENCS 281</td>
<td></td>
<td>/4</td>
<td>L</td>
<td>2.00</td>
</tr>
<tr>
<td>PHYS 205</td>
<td></td>
<td>/4</td>
<td>01</td>
<td>3.00</td>
</tr>
<tr>
<td>PHYS 225</td>
<td></td>
<td>/4</td>
<td>01</td>
<td>1.00</td>
</tr>
</tbody>
</table>

01/06/92 TRANSFERRED TO HONOURS SOFTWARE SYSTEMS

DATE OF ISSUE: 02/11/94
In such situations the student can fill in the registration form during the advising session and discuss his difficulties with the advisor. It is at this stage CAS can be very helpful.

If the student has filled in the form by listing his preferred courses, prior to seeing an advisor, then the advisor checks (manually) the registration form. If the student is unclear of his preferences, then the advisor interacts with the student and asks him of his preferences/constraints in terms of courses, time schedule, campus and workload he is willing to accept. In either case, the advisor suggests the courses.

The advising session ends when the advisor has approved the registration form. The existing advising procedure is shown schematically in Figure 3.2.

### 3.4.5 Functional Requirements of CAS

- **Prerequisites To The System:**

  The database containing the following information should be made available before initiating the course advising system.

  - The transcripts of all the students.
  - The course schedule, listing the courses offered by each department.
  - The calendar stating the degree requirements and pre-requisite requirements outlined by the department.

- **Course Advising System - Services:**

  CAS is an interactive system, which intervenes in each and every action the user does and corrects him whenever something goes wrong. This system helps the user in completing his task easily. The following are the services provided by the system.

  - **Initialization:** When the student enters his id, the system performs a validity check on the id entered by the user. If the id entered is invalid, the system informs the user of the invalid input. Otherwise, the system retrieves the student’s transcript from the database, and gives the following options to the user:
Figure 3.2: Current Advising Procedure
1. Specify the preferences.
2. Specify the constraints.
3. Request for advice.
4. Terminate the advising session.

- **Specify the Preferences/Constraints:** The system allows the user to specify his preferences/constraints, in terms of courses, time schedule, campus, and work load (maximum/minimum number of courses, maximum/minimum number of credits). The system performs a validity check on each preferred/unpreferred course number entered by the user. The system informs the user if the course number given is invalid, or the student did not do the pre-requisite for the course.

- **Give Advice:** Based on the preferences/constraints given by the user, the pre-requisites the student has completed, and the course schedule set by the department, the system suggests a list of courses which the student should take, towards achieving his degree. The suggested course list given by the system contains the course number, course name, the instructor for that course, the day, the time, and the term in which the course is offered. After suggesting the list of courses the student should take, the system provides different options to the user:

1. The system asks the user if he has any more preferences/constraints.
2. If the suggested course list is acceptable to the user, then the system prepares a (hard) copy of the suggested courses.
3. Terminate the advising session.

The automated advising procedure is shown schematically in Figure 3.3.

### 3.4.6 Design Constraints

The platform and tools that aid the development of CAS are described below:

- Hardware Environment
Figure 3.3: Automated Advising Procedure
1. The user interface to the automated advising system will be implemented on the Sun Sparc10 Stations (running SUN O/S) under the X-Windows environment.

2. The user interface will be mouse-driven.

- Software Environment

1. The user interface to the automated advising system will be implemented in MOTIF using the UIM/X toolkit.

2. The user interface will be developed by using different colors identifying different components of the interface.

The informal requirements specification of the automated advising system is presented. The main qualities of the interface that are intended to provide are the short learning time and the ease of use. Such features are necessary in this type of applications, because of high task importance, low frequency and discretionary use of the system.

3.5 Dialog Specification in CUIM

The next phase in the advanced evolutionary prototype model is the user interface analysis. During the requirements analysis phase, the user model and the system model are constructed without regard for eventual implementation. The dialog between the user and the interaction objects, and between the interaction objects and the computational objects are captured during this phase. CUIM suggests the use of two major representation techniques for constructing user models and system models during the analysis phase: extended state transition diagrams, and interaction diagrams respectively. Since the dialog specifications play a vital role in CUIM, this section details the notations used during the analysis phase.
3.5.1 Extended State Transition Diagrams

Transition diagrams [Wass79] have long served as a means for unambiguous specification of programming languages [Neil77]. Transition diagrams are used by many people [Case82, Jaco83, Kier83] as a formal specification technique for describing user interfaces to interactive systems. The extended state transition diagrams [Wass85] retain the formalism, yielding an unambiguous method for dialog specification. These diagrams were selected in preference to other specification models (e.g. Backus-Naur form [Leeu90], command language grammar [Mora81] etc.) largely on the basis of relative comprehensibility.

An extended state transition diagram is a network of nodes and directed arcs. Each state transition diagram is associated with a textual description. The different sections in a textual description are: Actions, Diagram/Sub-Conv, Variable Declarations, Define, and Node Specifications (Figure 3.4).

A node is shown by a circle, representing a stable state awaiting some user input. Each node within a diagram is labeled, and an output message may be displayed when a node is reached. The section Node Specifications, in the textual description shows the message displayed (Figure 3.4) when a transition is made to that node. There exists one start node and one end node, for each transition diagram. The section Diagram, in the textual description, specifies the diagram name. Scanning of the diagram begins at a designated entry point (start node) and proceeds until an exit node is reached. Each arc shown by an arrow connects two nodes together, and represents a state transition based on some input. The input is designated by a string literal, such as "quit". One arc emanating from each node may be left blank, in which case it becomes the default transition, and is taken only when the input fails to match that specified on any other arcs. An operation is shown by a small square (Figure 3.4) with an associated integer. An action may be associated with a transition to represent an operation that is to be performed whenever a specific arc is traversed. The same action may be associated with more than one arc. The Actions section (Figure 3.4) of the text attached to the transition diagram, lists the various function calls invoked. Intuitively, one can see that paths may contain arbitrary strings and that the state transitions can invoke arbitrary operations. The distinguished inputs then lead to
different states from which other input symbols may cause yet additional actions.

The different features provided by the extended state diagrams are:

1. Specify the *formatting and layout* (Figure 3.4) of system output. The following are the notations used for formatting the system output:
   - In order to specify a message which begins on row 2, and at column 5, we specify it as:
     
     \[ r2, c5, 'message1' \]
   
   If the above line is followed by:
   
   \[ r + 1, c5, 'message2' \]
   
   Then it denotes that, message2 should be displayed at column 5, and on the succeeding row of message1.
   - Rather than counting spaces to find the correct starting point, in order to center a message, the symbol “c.” is used.
   - The symbol “eol” denotes the end-of-line.
   - The symbol “nl” denotes a new line, and indicates that successive messages should be displayed on the next line.
   - The symbol “cs” denotes clear screen.

2. Display the input text given by the user as part of the output specification. *Variables* are used in such contexts. All the variables and their types are declared in the Variable Declarations section, of the textual description. The variable name can be shown on one or more arcs in a diagram. When such an arc is traversed, the input string is assigned to that variable. In Figure 3.4, the account number entered by the user is assigned to the variable “acct_no”.

3. The extended state diagrams provide facilities for different forms of *input processing*. The appearance of the “!” followed by a character on an arc means that a single “character” input is used to cause a transition. Also, some times the input string ends with nonstandard terminators such as: esc, tab. This is facilitated by giving a list of zero or more alternative terminators to the left of the “!” and the length, if fixed, is given to the right (Figure 3.4). For such cases,
input is read until a nonstandard terminator is received, and then truncated to the given length.

4. The symbol "+" on an arc denotes that, no user input is required, for the transition to take place. And, the symbol "@" denotes that any character input causes the transition to occur.

5. For highly interactive applications, there is much more dialog between the user and the interface. Therefore to improve the comprehensibility of the dialog, sub-conversations are used in the state transition diagrams. A sub-conversation is represented as a named rectangle on a transition arc (Figure 3.4). A sub-conversation is a refinement of the dialog between the user and the interface. The various states involved in a sub-conversation are shown as a separate diagram. In such cases, the "Diagram" section of the textual description, is replaced by the "Sub-Conv" section. The "Sub-Conv" section specifies the name of the sub-conversation. When control reaches a sub-conversation, control is transferred to the start node (a different diagram) of this sub-conversation. Again, when control reaches the end node of a sub-conversation, control returns to the successive node in the original state diagram, from where control has been transferred before. Sub-conversations are also accompanied by return values. The return value is indicated as a transition value followed by a "#" sign, on the final transition of the sub-conversation.

6. In addition to handling the syntactic user input, the transitions between nodes are extended to handle the semantic values too. This is necessary because, the direction of a dialog is often dependent upon the result of an action. For example, in a banking system, the user (a teller) would be asked to input a customer account number. A subsequent action would be to look up this account number in the bank's customer account database. The next message presented to the teller would depend on the "result" of the search. So, a return value is associated with the action, and then to branch on that value. This is accomplished by indicating one or more arcs emerging from an action box (Figure 3.4), with arcs labeled with alternate return values.

7. While specifying user interfaces for real time systems, unexpected delay in the user input indicates a problem. Therefore, it is desirable to be able to effect
Actions
1 verify(acct_no)

Diagram main
digit acct_no

Define key 'Press any key to continue.'

node start
   '* Enter your account number.'

node err
   r2,c_,'Invalid account number.', nl
   r+1,c_,'key'

node options
   r2,c_,'W : for withdrawal.', nl
   r+2,c_,'D : for Deposit.', nl
   r+2,c_,'B : for Balance.', nl

node x
   c_ /* exit the system */

Legend:

start node
end node
no user input is required
a single character input is required
indicates a non-standard terminator in the input
subconversation

Figure 3.4: Symbols in Extended State Transition Diagrams
a transition on the expiration of a predefined time limit. In this way, it is possible to branch to another node, from which a reminder or help message can be displayed. Such situations can be handled by using the alarm transition. The alarm transition is denoted by writing the time limit on the appropriate arc (Figure 3.4). The alarm transition is made if no input is received from the user before the time limit expires.

8. The extended state transition diagrams also promote re-usability. This is quite handy because, one might want to display the same message at different times: examples are, online help messages, screen headings, error messages etc. The message, and its name are declared in the Define section (Figure 3.4) of the textual description. Re-usability is then provided by referring the message name, whenever we want to display that message.

3.5.2 Interaction Diagrams

The interaction diagrams proposed in this thesis model the communication between different Interaction Objects (IOBs) and Computational Objects (COBs). An interaction diagram is a graph consisting of nodes and directed arcs. To differentiate the interface objects from the computational objects, different representations are used. A circular node corresponds to a COB and the solid line rectangle represents an IOB. The user forming an external object (EOB) is shown by a broken line rectangle. An interaction is shown by a solid line with an uni-directional arrow. The different symbols used to build an interaction diagram are shown in Figure 3.5. Each interaction
represents a control (associated with data values) communication between two objects. The object which initiates the communication by sending an event with the data values associated to it is referred to as the from object, and the object which receives the event is referred to as the to object. The data values associated with an event are optional.

In response to an event from the user, an IOB may interact with more than one COB. For every interaction \( I_U \), from the user to an IOB, there is an interaction \( I_I \), coming from an IOB to the user. All those interactions that happen between the IOB and COB, from the time \( I_U \) is sent to the time \( I_I \) is given are referred to as the set of interactions. All the IOBs together are referred to as the set of interface objects and all the COBs together are referred to as the set of computational objects. The to object for the interaction \( I_U \) is referred to as the start object. Conversely, the from object corresponding to the interaction \( I_I \) is referred to as the end object. Those objects (excluding the start object and the end object) that communicate from the time \( I_U \) is sent to the time \( I_I \) is given are referred to as the intermediate objects. For every set of interactions there is a start object, end object, and intermediate objects. Start object and end object always belong to the set of interface objects. Intermediate objects are objects either from the computational objects set, or from the interface objects set. There can be more than one end object in a set of interactions, or some times both the start object and the end object may correspond to the same IOB.
In the CUIM methodology, an interface object does not perform any computations. An interface object, say \( IOB_1 \) always interacts with a computational object, say \( COB_1 \) to request for any required service. So, there is no necessity for an interface object \( IOB_1 \) to interact with another interface object \( IOB_2 \). Therefore, in CUIM an interaction between two IOBs is always interleaved by an interaction with a COB. This feature makes the design of the system simple, by reducing the number of objects with which an object has to interact. Considering the example of the banking system, the interactions between the IOBs and COBs, when the user enters the account number, represented as an interaction diagram is shown in Figure 3.6. The interface object, \( Start\ Interface \) after receiving the “acct_no” from the user, interacts with the computational object \( list \), by sending the event “verify” with the account number associated to it. The \( list \) object then verifies the validity of the account number given by the user. If the account number is invalid, then the “invalid” event is sent to the interface object \( Start\ Interface \). Otherwise, the “valid” event is sent. In the former situation, the IOB, \( Start\ Interface \) sends an “invoke” event to \( list \), requesting it to invoke the IOB object \( Err\ Interface \). The \( list \) object then invokes the IOB \( Err\ Interface \), by sending the “invalid” event with the associated error message.

Even though there is more than one output arrow emerging from the object \( list \), in the interaction diagram(Figure 3.6), it is important to understand that at any instant, only one of all the output events will happen. The next chapter shows how the interactions between the various IOBs and COBs (for the example system CAS) are specified using the interaction diagrams proposed here.

### 3.6 Summary

When the user interface and the program design activities are performed concurrently, interaction between the user interface designers and the program designers is necessary because, the program designers may overlook the modeling of some tasks required by the user interface designers and vice-versa. It is important to ensure consistency in the system being designed by these two groups of people. The USE methodology proposed by Wasserman et. al.[Wass85] does not model these interactions during the
analysis and design phases. This is a major drawback of the USE methodology. The concurrent methodology _CUIM_ developed, and which is the essence of this dissertation, overcomes such limitations and provides an effective environment for _concurrent engineering_ of user interfaces.
Chapter 4

User Interface Analysis in CUIM

The first phase in CUIM is the requirements analysis phase. It concentrates on understanding and modeling what the user wants. The result of the analysis is a clear understanding of the problems and issues which serve as a preparation for the design of the user interface. In section 4.1, the basic determinants of user behavior which serve in understanding the user are discussed. As the user interface provides a language for communicating with the user, section 4.2 concentrates in specifying the dialog between the user and the computer. The user interface acts as an intermediary between the user and the computational part of the software, therefore specifying the interactions between the interface objects and the computational objects is necessary. Section 4.3 specifies these interactions. Section 4.4 deals with verifying the extended state transition diagrams with the interaction diagrams. And finally, section 4.5 shows how to ensure that the output of the user interface analysis phase (the IAD document) and the output of the computational analysis phase (the CAD document) are consistent.

4.1 Constructing the User Profile

The performance[Mayh92] of a user can be measured in terms of the amount of time and effort consumed to complete a task. Therefore, improving the user's performance
in his or her job is the main goal for user interface design. Towards this end, the first step during the analysis is to identify the factors that affect or determine the user's performance. A classification strategy serves as an useful tool for knowing the user is described below[Mayh92].

- User Psychology

  It is a well known fact[Mayh92] that the user's motivation plays a significant role in the performance of tasks requiring motor, cognitive, or perceptual skills. It is important to provide incentives if users lack motivation (either in their jobs in general or to use computers in particular). Discretionary users (who choose whether or not to use a computer) need to feel immediately that a system will not take too long to learn. Mandatory users (who must use a computer as part of their job) need to immediately experience some benefit from using a computer. Users who are highly motivated out of fear (for example: of losing their job, or of appearing incompetent) need the reassurance that the system is not overly complex and will not be difficult to learn. Therefore, user interfaces which are consistent, predictable, and simple to understand should be designed to increase motivation.

- Knowledge and Experience

  Instead of considering the user's experience as a simple binary expression (novice, expert), a number of types of knowledge and experience which are listed below, are relatively independent of one another and must all be considered during user interface design.

  1. Task experience corresponds to knowledge of the task domain. Shneiderman [Shne80] refers to task experience as the semantic knowledge of the system. For example, to use an air traffic control system, the user needs to know quite a bit about how the air traffic is controlled.

  2. In contrast to task experience, system experience corresponds to knowledge of a particular language or mode of interaction of a given system. Users may have been performing their job for years effectively by hand, but will be unable to perform the job when it is automated until they
have learned the idiosyncratic language of the new system. In the example of the air traffic controller, system experience pertains to knowledge of the syntax for entering a flight plan, codes for different airports, and the route specifications, the signaling commands which would be received from other controllers etc.. System experience can be called as syntactic knowledge[Shne80].

3. It is important that the user interface designers consider the level of computer literacy of the intended user population. The user interface designers should sketch out whether the users are highly technical and computer literates, or they have no prior experience with computers at all? The user interface designers should know whether the users of the system will be familiar with the use of keys such as tab, return, backspace, and with computer jargon and concepts such as memory, saving etc.?

4. Another important user characteristic that should be considered is the typing skill of the user. With the advent of GUI and other interface techniques, their skills may be viewed generally as “user-computer interaction skills”. It will be clear later on, how the various characteristics listed here affect the decisions made during the design.

The various kinds of knowledge and experience listed above are relatively independent of one another. For example, a given user may have low typing skill, low computer literacy and high task experience. Any combination of different levels of different kinds of knowledge and experience is possible. Users with varying degrees of knowledge and experience have different needs that must be accommodated by the system. For example, users with little task or system experience will need a system with many prompts of both a semantic and syntactic nature, and effective error recovery procedures. Therefore, the users knowledge and experience which affects the performance is an important determinant to be considered during user interface design.

- Job and Task Characteristics
Also, the nature of the user's job (or tasks), the frequency with which it is performed and its importance will affect the user's level of knowledge and experience over time. These factors, in turn determine the relative emphasis to be put on ease of learning versus ease of use, and thus will affect the amount of syntactic and semantic assistance required in the interface. Therefore, the job and task characteristics drive user interface design in many ways. A number of dimensions of the user's job, which have implications for user interface design are discussed below:

1. One of the most important determinants of user performance with an interactive system is the frequency of use of the system. Frequency of use has profound implications for user interface design because it affects learning. Frequency of use affects system design in two ways. First, users who spend a lot of time on the system are usually willing to invest more time in learning, and therefore efficiency of operation takes precedence over ease of learning. When the user interface is mainly used by frequent users, its design may give more weightage to the efficiency of operation, and the adaptability of the interface to individual users or user groups. Low frequency users will not be able to learn and remember an interface unless it is designed for ease of learning.

2. Another important determinant is primary training. The amount of available training determines in part how easy to learn the interface must be.

3. Another determinant that guides user interface designers in the ease-of-use/ease-of-learning trade offs is system use. When the system usage is discretionary, first impressions are more important to create motivation, so promoting ease of learning is important. But if the system use is mandatory, users usually get training, and ease of use will provide them with a sense of power and control, thus keeping up their motivation.

4. The importance of the task automated by a new system will influence how much of an investment in learning the users (especially discretionary users) are willing to make. According to the amount of time the users are willing to invest for training/learning, promoting ease of learning would be more or less important relative to other design goals such as power, and ease of use. When the task is perceived to be important, motivation is high, then
<table>
<thead>
<tr>
<th>User Psychology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{Low, High}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Knowledge and Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Literacy</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{Low, Moderate, High}</td>
</tr>
<tr>
<td>System Experience</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{Low, Moderate, High}</td>
</tr>
<tr>
<td>Task Experience</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{Low, Moderate, High}</td>
</tr>
<tr>
<td>Typing Skill</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{Moderate, High}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job and Task Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency of Use</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{Low}</td>
</tr>
<tr>
<td>Primary Training</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{None}</td>
</tr>
<tr>
<td>System Use</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{Discretionary}</td>
</tr>
<tr>
<td>Task Importance</td>
</tr>
<tr>
<td>∈</td>
</tr>
<tr>
<td>{High}</td>
</tr>
</tbody>
</table>

Table 4.1: User Profile

the system should promote ease of use. The task importance and frequency of use are not necessarily the same. The task may have a high importance but be executed with a low frequency.

To understand the user population and to improve user's performance, the various determinants described above are used to construct the user profile. The scale for each of the determinant could be: Low, Moderate, and High. As mentioned in Chapter 3, the course advising system (CAS) will be used as an example, while discussing the various activities in CUM. The user profile constructed for the intended population of CAS is listed in Table 4.1. The intended user population for CAS includes students with low motivation, either due to fear of using a computer or they don't like to be controlled by a machine. Also, there will be students who are highly motivated out of interest or due to the fear of appearing to be incompetent. Since, not all the students are computer literates\(^1\), the range for computer literacy varies from low to high. CAS is not a highly sophisticated application, therefore the syntactic knowledge required to use the system is not high. Since there are students with low computer literacy,

\(^1\) A computer literate for our purpose is expected to be confident in the use of a mouse, and the associated window operations.
the system experience varies from low to high. The first year students might not have enough knowledge of how the courses should be chosen. Therefore, the task experience for the users of CAS, also varies from low to high. We assume that every user is at least a moderate (10 words per minute) typist. Since, the advising system will be used only during the registration period, we say that the frequency of use of the system is low. The task importance for this application is high because it is important for the students to get advise before registering their courses. The characteristics primary training and system use are set by the management.

After establishing the user profile the next step in CUIM is to specify the dialog between the user and the interface.

4.2 Specifying the Dialog between the User and the Interface

In order to specify the dialog between the user and the interface, the user goals need to be identified. A goal(task) is something that the user tries to accomplish. By using the informal requirements document (IRD), the list of goals the users will have are identified. Sometimes, accomplishing a goal might require accomplishing one or more sub-goals. Table 4.2 shows the goals and sub-goals if any, to be accomplished for the example system CAS. It shows that goal1 requires at least one of the sub-goals 1.1, 1.2, 1.3 and 1.4.

After the goals and sub-goals are identified, the next step in CUIM is to define the specific actions that are to be performed by the user and the interface in order to accomplish these goals. This is done by specifying the dialog between the user and the interface, where the required inputs and outputs are identified. This dialog specification also identifies the major functions which provide linkage to the computational objects. During this stage, decisions are also made about, Task Ordering, Task Anticipation, and Assistance, as explained below:

- Task Ordering, which refers to making decisions about how much freedom must
<table>
<thead>
<tr>
<th>Goal</th>
<th>Description</th>
</tr>
</thead>
</table>
| Goal 1 | Specify the preferences  
|        | 1.1 Specify the preferred courses &/or  
|        | 1.2 Specify the preferred time &/or  
|        | 1.3 Specify the preferred campus &/or  
|        | 1.4 Specify the preferred workload |
| Goal 2 | Specify the constraints  
|        | 2.1 Specify the unpreferred courses &/or  
|        | 2.2 Specify the unpreferred time &/or  
|        | 2.3 Specify the unpreferred campus &/or  
|        | 2.4 Specify the unpreferred workload |
| Goal 3 | Request for advice |

Table 4.2: User Goals

be given to the user to switch between tasks.

• Task Anticipation, which refers to how much information must be provided about the next tasks allowed, once a particular task has been specified.

• Assistance, which corresponds to how much information must be suggested for error repair. Since, users make errors[38] either due to non-intent actions or due to inappropriate intentions (lack of semantic knowledge), error situations are also identified during this stage. The user actions are analyzed for possible error conditions that might occur during the execution of a task.

The dialog diagrams therefore capture the dynamic behavior of the interface. As described in Chapter 3, the extended state transition diagrams are used for specifying the dialog between the user and the interface.

An overview of the different dialogs between the user and the interface, which are specified towards accomplishing the goals listed above, is given in Table 4.3. Figure
<table>
<thead>
<tr>
<th>Dialogs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.1</td>
<td>This diagram specifies the initialization of the dialog between the user and the interface. The different tasks that can be carried out next, by the user are also specified here.</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>The dialog between the user and the interface when the user wanted to list his preferences is specified in this diagram.</td>
</tr>
<tr>
<td>Figure 4.3 through 4.6</td>
<td>These diagrams specify the dialog between the user and the interface when the user enters his preferred courses, preferred time, preferred campus, and preferred work load.</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>The dialog between the user and the interface when the user wanted to list his constraints is specified in this diagram.</td>
</tr>
<tr>
<td>Figure 4.8 through 4.11</td>
<td>These diagrams specify the dialog between the user and the interface when the user enters his unpreferred courses, unpreferred time, unpreferred campus, and unpreferred work load.</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>This diagram specifies the dialog between the user and the interface when the user requested the system to give advice. The tasks that can be carried out next, by the user are also specified here.</td>
</tr>
</tbody>
</table>

Table 4.3: Overview of the Dialog Specifications
Figure 4.1: Top level state transition diagram of the Course Advising System
4.1 shows the top-level state transition diagram for CAS.

The state transition diagram in Figure 4.1 begins at the node *start*. The action box numbered "1" is a call to "start_up". If this action returns "success", then control flows to node *key*; otherwise, control flows to node *nodb* and the message specified in the textual description for "node nodb" is displayed to the user. The system remains in this state waiting for user input. Once the user input is received, the control flows to node *x* and the program terminates.

If the control flows to node *key*, then the message given in the textual description for "node key" is displayed to the user. The system will remain in this state awaiting user input. If the input received from the user is "Exit", then a call to "shut_down" is made and the program terminates. Otherwise, the input given by the user is assigned to the variable "id", and a call to "verify_id" is made. The value of the variable "id" is passed as the parameter during the call. If this action returns "success" then control flows to the node *main*. Otherwise, control flows to node *inv_id* and an error message is displayed to the user. When the user hits any key, then control returns to node *key* and the system behaves as explained before.

When the control is at node *main*, the system provides a menu-like interface by providing different options to the user. Depending on the user input, the system may enter the sub-conversations "pref choice", "un pref choice", or "advise". The *main* node also provides for terminating the program, or going to the previous option of entering an id number.

The sub-conversation "pref choice" is shown in Figure 4.2, and has much the same structure as does the top-level diagram. The start node for this sub-conversation is *preferences*. This node displays a list of options the user can choose to specify his preferences. The "pref choice" sub-conversation invokes "pref course" when the user wishes to list his preferred course(s). The sub-conversation "pref course" is shown in Figure 4.3. When the user enters the course number, the user input is assigned to the variable "course_no" and a call to "check_course_number" is made. The value of "course_no" is passed as parameter during the call. If this action returns "valid" then control flows to node *start*; otherwise, control flows to node *msg1* and an error
Figure 4.2: The ‘pref choi’ sub-conversation

message is displayed to the user. As soon as the user hits any key, control returns to
node start. The user can now enter the next course number (if any).

The user either chooses “Accept” or “Cancel” after listing his preferred course(s).
If the user input is “Cancel”, then a call to “clear_input” is made in “pref course” and
then control flows to node x, at which point control is returned to node preferences
in sub-conversation “pref choi”. If the return value in “pref course” is “Accept” then
control returns to node preferences via node x, without any invocation of a function
call. If the user chooses other preferred choices, then depending on the user input, the
“pref choi” sub-conversation invokes either “pref-time”, or “pref campus”, or “pref
load”. These sub-conversations are shown in Figures 4.4 through 4.6. Control returns
back to node main when the preferences node receives “Close” as input from the user.

From node main control may flow to sub-conversations “pref choi”, “un pref choi”,
or “advise” depending on the user input. Figures 4.7 through 4.11 model the interac-
tions between the user and the interface, when the user wishes to list his un-preferred
choices.

When the “advise” sub-conversation is invoked control flows to node start in Figure-
Figure 4.3: The 'pref course' sub-conversation

Figure 4.4: The 'pref time' sub-conversation
Figure 4.5: The 'pref campus' sub-conversation

<table>
<thead>
<tr>
<th>Transition Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advise</td>
<td>give a list of suggested courses</td>
</tr>
<tr>
<td>cancel</td>
<td>cancel the input given during the accomplishment of the current goal/sub-goal</td>
</tr>
<tr>
<td>course_no</td>
<td>verify the validity of the given course number</td>
</tr>
<tr>
<td>done</td>
<td>requested operation has been completed successfully</td>
</tr>
<tr>
<td>list</td>
<td>requested operation has been completed successfully and a list of values are returned</td>
</tr>
<tr>
<td>Exit</td>
<td>terminate the program</td>
</tr>
<tr>
<td>failure_ansg</td>
<td>an error is encountered while doing the requested operation and the type of error is reported</td>
</tr>
<tr>
<td>id</td>
<td>verify the validity of the given id number</td>
</tr>
<tr>
<td>valid</td>
<td>data value given is acceptable</td>
</tr>
</tbody>
</table>

Table 4.4: Data Dictionary for transition values to & from a function call
Figure 4.6: The 'pref load' sub-conversation
Figure 4.7: The 'unpref choi' sub-conversation

Figure 4.8: The 'unpref course' sub-conversation
Figure 4.9: The 'unpref time' sub-conversation

Figure 4.10: The 'unpref campus' sub-conversation
Figure 4.11: The 'unpref load' sub-conversation

Figure 4.12: The 'advise' sub-conversation
4.12. In this state, the system displays the suggested course list to the user. At this point, the user can view either the preferred choices, or un-preferred choices, or accept the course list suggested by CAS, or even terminate the program. If the user chooses preferred choices or un-preferred choices, then either the sub-conversation "pref choi" or "un pref choi" respectively is invoked. Once the user finishes listing his preferences/constraints control is returned to the start node in Figure 4.12. It is obvious from this Figure that a call to "do_advise" is made after returning from either "pref choi" or "un pref choi" sub-conversation. Therefore, the start node always displays the up-to-date suggested course list to the user.

After specifying the dialog between the user and the interface, all those transition values to and from the function calls in the various state transition diagrams are tabulated, from which data dictionaries are generated. The term transition values is used to refer to values on the transitions going to or coming from a function call in the state diagrams. The data dictionary corresponding to function calls in the state transitions diagrams for CAS is shown in Table 4.4.

4.3 Specifying the Interactions between the IOBs and COBs

After specifying the dialog between the user and the interface, the next step in CUIM is to specify the interactions between the IOBs and the COBs. The interaction diagrams explained in the previous Chapter, are used to specify the communication between different IOBs and COBs.

The dialogs specified between the user and the interface are used to identify the different IOBs and COBs. The different tasks performed by the interface depicted in the textual description of the state transition diagram, are mapped to IOBs, and the function calls in the state transition diagrams correspond to COBs in the interaction diagrams. In the state transition diagrams the text inside a node represents the name of the state, whereas in the interaction diagrams, the text inside a node represents to the name of an object. The different IOBs required for CAS and a description of each
<table>
<thead>
<tr>
<th>IOB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>an object which initiates the dialog with the user</td>
</tr>
<tr>
<td>Main Interface</td>
<td>an object which provides the user with different goals which can be accomplished next</td>
</tr>
<tr>
<td>Prefs Interface</td>
<td>an object which allows the user to specify his preferences for courses, time schedule, campus, work load, and confirm his specifications</td>
</tr>
<tr>
<td>Constraints Interface</td>
<td>an object which allows the user to specify his unpreferred courses, time schedule, campus, work load, and confirm his specifications</td>
</tr>
<tr>
<td>Advisor Interface</td>
<td>an object which gives the list of suggested courses to the user; also provides the user with different goals which can be accomplished next</td>
</tr>
<tr>
<td>Messenger</td>
<td>an object which gives messages (error) to the user</td>
</tr>
</tbody>
</table>

Table 4.5: Data Dictionary for IOBs in CAS

IOB is given in Table 4.5. Table 4.6 lists the different COBs and the data dictionary for each COB. The COBs modeled by the interface group during this phase serves to establish a link between the interface process and the computational process. The transition values that are generated in response to each input from the user, which are specified at an abstract level using function calls in the state transition diagrams are refined in the interaction diagrams.

The interactions between IOBs and COBs when the user enters the id number is shown in Figure 4.13(a). The interface object, Initiator after receiving the id input from the user, interacts with the computational object list by sending the event “data” with the id value associated to it. The list object then verifies the validity of the id value given. If the id value is valid then the “valid” event is sent to the interface object Initiator. Otherwise, the “invalid” event is sent to the IOB object Initiator. In the former situation, the Initiator sends an “invoke” event to list, requesting it to invoke the IOB object Main Interface. The list object then invokes the IOB Main Interface by sending the “valid” event. For the later case (if id is invalid), the Initiator sends an “invoke” event to list, requesting it to invoke the IOB object Messenger. The
<table>
<thead>
<tr>
<th>COB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>advisor</td>
<td>a person who is qualified to suggest the list of courses the student should take during the current academic year, by considering the student’s preferences/constraints if any, the pre-requisites the student has completed, the degree requirements set by the university, and the course schedule given by the department</td>
</tr>
<tr>
<td>approver</td>
<td>an object that prepares an official listing of the suggested courses</td>
</tr>
<tr>
<td>controller</td>
<td>an object which controls the system by initializing all the databases when the system is started, and shuts down all the databases while exiting the system</td>
</tr>
<tr>
<td>list</td>
<td>a repository for all the values entered by the user; this takes care of checking the validity of the value given by the user. Upon request, this object clears all the data values which are stored during the accomplishment of the current goal/sub-goal.</td>
</tr>
</tbody>
</table>

Table 4.6: Data Dictionary for COEs in CAS
<table>
<thead>
<tr>
<th>Internal Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>do_advise</td>
<td>give a list of suggested courses</td>
</tr>
<tr>
<td>clear_input</td>
<td>cancel the input given during the accomplishment of the current goal/sub-goal</td>
</tr>
<tr>
<td>data(course.num)</td>
<td>verify the validity of the given course number</td>
</tr>
<tr>
<td>done</td>
<td>requested operation has been completed successfully</td>
</tr>
<tr>
<td>list</td>
<td>requested operation has been completed successfully and a list of values are returned</td>
</tr>
<tr>
<td>do_exit</td>
<td>terminate the program</td>
</tr>
<tr>
<td>invalid(err.type)</td>
<td>an error is encountered while doing the requested operation and the type of error is reported</td>
</tr>
<tr>
<td>data(id)</td>
<td>verify the validity of the given id number</td>
</tr>
<tr>
<td>valid</td>
<td>data value given is acceptable</td>
</tr>
</tbody>
</table>

Table 4.7: Data Dictionary for Internal Events in Interaction Diagrams

list object then invokes the IOB Messenger by sending the “invalid” event and the corresponding error type as the parameter. Even though there is more than one output arrow emerging from the object list, in the interaction diagram 4.13(a), it is important to understand that at any instant, only one of all the output events will happen. Also, from this diagram, one can easily see that no two solid line rectangular nodes interact.

Figure 4.13(b) shows the interaction diagram when the user enters the preferred course number. Figures 4.14 through 4.16 show the interactions between different IOBs and COBs for different user inputs. The solid rectangular node shown in Figure 4.16(b) means, that the user sends an “Exit” event either to the Initiator or to the Main Interface or to the Advisor Interface. Depending on which IOB receives the user input, that IOB interacts with the computational object controller to achieve the desired functionality.

Once the interactions between the IOBs and the COBs are specified, the next step is to prepare a data dictionary for the internal events in the interaction diagrams. All those events that flow between an IOB and a COB are termed as internal events, and
Figure 4.13: Interaction Diagrams: Set One
Figure 4.14: Interaction Diagrams: Set Two

Figure 4.15: Interaction Diagrams: Set Three
those events that flow between the user and an IOB are termed as *external events*. The data dictionary for the internal events modeled in the interaction diagrams for CAS is given in Table 4.7.

### 4.4 Verifying the Extended State Diagrams with the Interaction Diagrams

The *extended state transition diagrams* and the *interaction diagrams* tackle different aspects of the same problem. It is clear from Sections 4.2 and 4.3 that, the state diagrams concentrate on the dialog between the user and the interface by abstracting the interactions between the IOBs and the COBs, whereas the interaction diagrams concentrate on events between IOBs and COBs by abstracting the external events. Since, humans lack the ability to perform with perfection, verification at various stages of the development process is necessary. The term *verification* in our context can be defined as an activity which assures that the results of each successive step in an user interface development cycle correctly realizes the intentions of the previous step. In *CUIM*, the process of verification is carried out at the end of each phase to ensure a more reliable process of user interface development. At the end of the analysis phase, we verify that the state transition diagrams and the interaction diagrams are consistent. The verification process is as follows:
1. Ensure that a circular node, say $S_1$ (explained in the textual description of the nodes) in the state transition diagram corresponds to a solid line rectangular node, say $IOB_1$ (described in the data dictionary for IOBS) in the interaction diagram.

To ensure that (1) is true: Construct a state-IOB association table, which contains three columns. The first column specifies the dialog diagram, the second column lists the state node in that diagram, and the corresponding IOB object is specified in the last column. After constructing the state-IOB association table, the textual description of the nodes and the data dictionary for IOBs are used to check that the state node corresponds to the IOB object.

2. Ensure that a function call, say $FC_1$ (explained in the textual description of 'Actions') in the state diagram corresponds to a circular node, say $COB_1$ (described in the data dictionary for COBs) in an interaction diagram.

To ensure that (2) is true: Construct a function-call-COB association table, which contains three columns. The first column specifies the dialog diagram, the second column lists the function call number in that diagram, and the corresponding COB object is specified in the last column. After constructing the function-call-COB association table, the textual description of the actions and the data dictionary for COBs are used to check if the function call corresponds to the COB object.

3. If (1) and (2) are true, then the transition value between $S_1$ and $FC_1$ should correspond to the internal event between $IOB_1$ and $COB_1$ (Figure 4.17). If this correspondence between the transition value and the internal event is established, we conclude that the state transition diagrams and the interaction diagrams are consistent.

To prove (3), the correspondence between the internal value and the transition value is established by constructing the event verification table. The event verification table contains two columns, where the first column lists the transition value between the nodes $S_1$ and $FC_1$ and the second column lists the internal event between $IOB_1$ and $COB_1$. Similarly, all the transition values and internal events are listed in the event verification table. Once, the event verification table is constructed, we can check that each transition value has an associated
internal event. The data dictionary for transition values and the data dictionary for internal events is used to check that the transition value corresponds to the internal event.

Since the event verification table groups the transition values in the state diagram with the internal events in the interaction diagram, one can easily identify:

1. if there are any transition values that are not modeled in the interaction diagrams and
2. if there are any function calls which are missing in the state diagrams

The verification process for the example system CAS is given below:

- **Step 1:** The state-IOB association table for CAS is shown in Table 4.8. By using the data dictionary for IOBs (Table 4.5) and the textual description of the nodes (given in Section 4.2), we say that the state nodes in column 2 of Table 4.8 correspond to the IOB objects in column 3.

- **Step 2:** The function_call-COB association table for CAS is shown in Table 4.9. By using the data dictionary for COBs (Table 4.6) and the textual description of the actions (given in Section 4.2), we say that the function calls in column 2 of Table 4.9 correspond to the COB objects in column 3.
<table>
<thead>
<tr>
<th>Dialog</th>
<th>State</th>
<th>IOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.1</td>
<td>key,x; main,x</td>
<td>Initiator</td>
</tr>
<tr>
<td>Figure 4.2, Figure 4.3 through 4.6, Figure 4.6</td>
<td>start; start</td>
<td>Main Interface</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>maco, mico, macr, micr</td>
<td>Prefs Interface</td>
</tr>
<tr>
<td>Figure 4.8 through 4.11, Figure 4.10</td>
<td>start; start</td>
<td>Unprefs Interface</td>
</tr>
<tr>
<td>Figure 4.1, Figure 4.3, 4.8</td>
<td>inv.id, nodb; msg</td>
<td>Messenger</td>
</tr>
<tr>
<td>Figure 4.1, Figure 4.12</td>
<td>x, start,x</td>
<td>Advisor Interface</td>
</tr>
</tbody>
</table>

Table 4.8: State-IOB Association Table

<table>
<thead>
<tr>
<th>Dialog</th>
<th>FunctionCall</th>
<th>COB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.1, Figure 4.12</td>
<td>1, 2; 1</td>
<td>controller</td>
</tr>
<tr>
<td>Figure 4.1, Figure 4.3 through 4.6, Figure 4.8 through 4.11, Figure 4.3, 4.8, 4.12</td>
<td>3; 1; 1; 2</td>
<td>list</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>3</td>
<td>approver</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>4</td>
<td>advisor</td>
</tr>
</tbody>
</table>

Table 4.9: FunctionCall-COB Association Table
<table>
<thead>
<tr>
<th>Transition Values</th>
<th>Internal Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advise</td>
<td>do_advise</td>
</tr>
<tr>
<td>cancel</td>
<td>clear_input</td>
</tr>
<tr>
<td>course_no</td>
<td>data(course_num)</td>
</tr>
<tr>
<td>done</td>
<td>done</td>
</tr>
<tr>
<td>list</td>
<td>done(list)</td>
</tr>
<tr>
<td>Exit</td>
<td>do_exit</td>
</tr>
<tr>
<td>failure_msg</td>
<td>invalid(err_type)</td>
</tr>
<tr>
<td>id</td>
<td>data(id)</td>
</tr>
<tr>
<td>valid</td>
<td>valid</td>
</tr>
</tbody>
</table>

Table 4.10: Event Verification Table

- **Step 3:** After ensuring step 1 and step 2, we should now ensure that the transition values in the state diagrams corresponds to the internal events in the interaction diagrams. The *event verification table* for CAS shown in Table 4.10 associates transition values to internal events. By using the data dictionary for internal events (Table 4.7) and the data dictionary for transition values (Table 4.4), we say that the transition values in the state diagrams correspond to the internal events in the interaction diagrams. Therefore, we conclude that the state transition diagrams and the interaction diagrams are consistent.

### 4.5 Ensuring Consistency between IAD & CAD

All the above activities (constructing the user profile, modeling the dialog between the user and the interface, and modeling the interactions between the IOBs and the COBs, verifying the extended state transition diagrams with the interaction diagrams) that are carried out during the user interface analysis phase comprise the *Interface Analysis Document* (IAD). The *Computational Analysis Document* (CAD) comprises all those activities carried out during the analysis phase of the computational process. Once the IAD document and the CAD document are produced, the analysis walk-through
is conducted. An analysis walk-through is an informal analysis of IAD and CAD documents, as a cooperative and organized activity by the two groups of people (user interface engineers, and the software engineers). Software engineers from either group (interface group and the computational group) meet to review the output of the analysis phase of both the interface process and the computational process. This meeting focuses on "discovering the errors and inconsistencies", but not fixing them.

Key people (say, the group leaders) in either group walk through the IAD and CAD documents to present and explain the rationale of their work. The software engineers check that, for every internal event sent by an IOB in the IAD, there exists a COB in the CAD document, which accepts that event. Similarly, for every internal event sent by a COB in the CAD document, there should exist an IOB in the IAD document, which accepts that event. During this process, the user interface engineer takes notes on the changes that are to be made to the IOBs/COBs in the IAD document. This applies to the software engineer too. Therefore, this walk-through serves the interface group to ensure that the CAD document does not miss any of the tasks that are modeled in the IAD document and vice-versa.

CAS is used as a running example in this thesis. At the end of the user interface analysis phase, we have the IAD document as the output of this phase. Since the various activities in CUIM are independent of the activities involved in the computational process, this thesis does not concentrate on the computational analysis phase. In following the advanced evolutionary prototype model, ensuring consistency between the IAD document and the CAD document is necessary. This requires the CAD document for CAS, which was prepared by Kim[Duon95] in her project work. We conducted analysis walk-throughs to review the IAD and CAD documents. By reviewing the IAD and CAD documents, we ensured that the CAD document does not miss any of the tasks modeled in the IAD document and vice-versa.

The analysis walk-through served us to identify the changes that are to be made to the IAD and CAD documents to ensure consistency between them. The changes that are to be made to the IAD document are in the interaction diagrams. The object names given to the COBs in the interaction diagrams are to be changed in accordance to the names specified in the CAD document. Table 4.11 shows the changes that will
<table>
<thead>
<tr>
<th>Existing COBs</th>
<th>New COBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>advisor approver</td>
<td>Suggested Courses</td>
</tr>
<tr>
<td>list</td>
<td>Transcript</td>
</tr>
<tr>
<td></td>
<td>User Constraint</td>
</tr>
<tr>
<td>controller</td>
<td>CAS</td>
</tr>
</tbody>
</table>

Table 4.11: Necessary Changes to the COBs

be made to the COBs which are modeled during the user interface analysis phase.

The time taken to produce the IAD document, may not be the same as the length of the computational analysis stage. The length may vary from application to application. In such cases, the interface group postpones the *analysis walk-through* until the CAD document is available, or may enter the *user interface design stage* in anticipation of consistency. The next Chapter deals with the design issues in *CUIM.*
Chapter 5

User Interface Design in CUIM

In Chapter 4, we considered the user interface analysis aspects following CUIM, and in this Chapter we consider the interface design aspects. The user interface part of a software system is something which has look and feel characteristics. Since the feel of the interface cannot be achieved until it is implemented, specifying how the interface looks is tackled during the design phase. Section 5.1 discusses how the look of the interface (user view) can be constructed. The dialog between the user and the interface specified during the analysis focussed on the user view of the interface, and summarized in the IAD document. This is to be refined further in a way that it corresponds to the decisions made during the dialog design. The issues of how this refinement can be done is discussed in Section 5.2. Section 5.3 discusses how the various design activities such as listing the class charts, depicting the spatial organization and specifying the class descriptions help the user interface designers to construct the static structure (designer's view) of the interface. By designing system interactions which realize the behavior of the various classes and the interaction relationship among them, the user interface designer moves closer to implementation. Section 5.4 discusses how this can be done.

Since the design phase in CUIM includes various activities to be carried out, ensuring consistency among the outputs of these activities is important. Section 5.5 shows how the design can be reviewed in order to ensure consistency. As shown in the
advanced evolutionary prototype model, the outcome of the design phase is the IDD
document. Since the user interface analysis focuses on what needs to be done (IAD),
and the user interface design provides a solution to the problem analyzed during the
analysis (IDD), Section 5.6 verifies that the IAD and IDD documents are consistent
with each other. And finally, Section 5.7 ensures that the outcome of the design phase
of the user interface process and the computational process are consistent.

5.1 Dialog Design

Since the user interface consists of different dialog styles, in order to communicate
with the user, identifying appropriate dialog styles which satisfy the user needs is
important. Therefore, considering the User Profile specified during the analysis, the
first step in dialog design is to identify the appropriate dialog styles. The second step
is to decide how to integrate the different dialog styles identified during step one in
order to maximize the overall usability. Subsection 5.1.1 discusses the first step and
subsection 5.1.2 discusses the second step.

5.1.1 Identifying Appropriate Dialog Styles - Cell Matrix
Method

A cell matrix is a rectangular array of cells set out in rows and columns. The Cell
Matrix Method [Mayh92] proposes a strategy for selecting an appropriate set of dialog
styles for an application. In the cell matrix, the top-most row lists the different
dialog styles from left to right, and the left-most column lists the user characteristics
from top to bottom. Figure 5.1 shows the cell matrix which lists the different dialog
styles and the user profile. Each cell in the matrix holds a particular value of the
user characteristic in that row, for which the dialog style in that column would be
appropriate. For example, in Figure 5.1, going from top to bottom under the column
"Menu", it can be seen that menus might be an appropriate dialog style for users
with low motivation, low typing skill, low system experience, low task experience, low
computer literacy, low frequency of use and so on. In contrast, going down the column
<table>
<thead>
<tr>
<th><strong>Motivation</strong></th>
<th><strong>Fill-in forms</strong></th>
<th><strong>Question and answers</strong></th>
<th><strong>Command language</strong></th>
<th><strong>Function keys</strong></th>
<th><strong>Direct manipulation</strong></th>
<th><strong>Natural Language</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low Moderate</td>
<td>Low High</td>
<td>Low High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Typing skill</td>
<td>Low Moderate</td>
<td>Moderate High</td>
<td>Moderate High</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>System experience</td>
<td>Low Low Moderate</td>
<td>Low Moderate High</td>
<td>High Low</td>
<td>Low</td>
<td>Moderate</td>
<td>Low</td>
</tr>
<tr>
<td>Task experience</td>
<td>Low Moderate</td>
<td>Low High</td>
<td>Moderate High</td>
<td>Moderate High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Computer literacy</td>
<td>Low Moderate</td>
<td>Low High</td>
<td>Moderate High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Frequency of use</td>
<td>Low Moderate</td>
<td>Low High</td>
<td>High Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Primary Training</td>
<td>Little or none</td>
<td>Little or none Formal</td>
<td>Little or none Some</td>
<td>Little or none</td>
<td>Little or none</td>
<td>Discretionary</td>
</tr>
<tr>
<td>System Use</td>
<td>Discretionary</td>
<td>Discretionary</td>
<td>Mandatory Discretionary</td>
<td>Low</td>
<td>Low</td>
<td>Discretionary</td>
</tr>
<tr>
<td>Task Importance</td>
<td>Low Moderate</td>
<td>Low High</td>
<td>Low High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Figure 5.1: Appropriate Dialog Styles - Cell Matrix Method (from [Mayh92])

under "command languages", it can be seen that command languages might be an appropriate dialog style for users with high motivation, moderate to high typing skill, high system experience, high task experience, high computer literacy, high frequency of use and so on.

After determining where the intended user group falls on each user characteristic (constructing the user profile), we read across each row in the matrix by marking (putting a tick mark) every cell that matches the user characteristics. For instance, if the intended users for an application are low in motivation, then all dialog styles in the matrix, except command language would be marked in the row for the user characteristic "motivation". Therefore, an initial marking is prepared by marking the cells which are appropriate for the user domain constructed. Then, the number of tick marks obtained for each dialog style are added. The total for each dialog style is tallied at the bottom of each column.

Once the initial marking has been done, a second pass is made and each "unmarked
cell" is further inspected against the following criteria: Check whether the dialog style for users other than those indicated in the cell for each row or user characteristic, has any serious disadvantages? If the dialog style does not have any serious disadvantages for that user characteristic, then the cell should be marked. For example, if the intended users for an application have users with high typing skill, then the cell under "Menu" for the row "typing skill" will not be marked during the initial marking phase. However, menus do not carry any penalty for good typists. Therefore, menus should get a tick mark on the typing skill characteristic. Even though they provide no special advantage for high-skill typists, they do not introduce any particular disadvantage for this type of users. In brief, we can conclude that, even though the user characteristics noted in each cell do not match the user characteristics in the user profile, they should still be marked if the dialog style does not pose any particular disadvantage for the users with characteristics listed in the user profile.

At the end of the second pass, the number of tick marks for each dialog style across user characteristics are again added. The new total for each dialog style is tallied at the bottom of each column. The dialog style with the highest score is considered as the best match to the user profile constructed. At this point, if more than one dialog style has the same highest score, then the highest score dialog styles are examined in the context of other factors such as:

1. Identify the cost of implementing the dialog style.

2. The accommodation of this dialog style on the available hardware platform; i.e., check if the hardware imposes any constraints for this dialog style.

3. The matrix method described above, does not take into account the "relative importance" of the different user characteristics. The scoring strategy outlined above, gives all user characteristics equal weight in the scoring. Therefore for applications where some of the user characteristics are more important than others, an alternative strategy would be to create a weighted matrix, where, the characteristics are prioritized and a weighting factor is included in the scoring technique.

In any case, it is important to note that, for a given set of users and tasks, the
Figure 5.2: Appropriate Dialog Styles - First Pass

*cell matrix method* provides only a cookbook strategy rather than a strict algorithmic method, to help select an appropriate primary dialog style.

We will now see, how the cell matrix method described above, helps us to design the dialog for the user profile constructed during the user interface analysis phase, for the example system CAS. Since the example system CAS contains users whose motivation is low as well as high, all the dialog styles in the row for the user characteristic *motivation* should be marked. Since the users typing skill ranges from moderate to high, *fill-in-forms, question and answers, command language, and natural language* should be marked in the *typing skill* row. Figure 5.2 shows the cell matrix, resulted from the initial marking of every cell where a simple match with the user characteristic value was found.

For users with moderate/high typing skill, menus are not a disadvantage. So, during the second pass, a tick mark can be made under *menu* for the *typing skill* row. Also, the menu style does not impose any penalty for high task importance. So the
task importance row under menu style can be marked. Since the frequency of use for the example system CAS is low, the chances that the user makes syntactic mistakes would be high. So, the cell for frequency of use under fill-in forms should not be marked. Fill-in forms provides a forward context and due to high task importance, the cell under fill-in forms for the task importance row can be marked. Question and answers make the application tedious by asking the user to enter a lot of input. This would cause typographical errors (typos). So, users need to spend more time correcting typos, rather than doing their job. Therefore, the cell under question and answers for the task importance row should not be marked. Command language loads the users long-term memory due to low frequency and discretionary use of the system. So, the cells for frequency of use and system use under command language should not be marked. Function keys are not a disadvantage for users with moderate/high typing skill. So, the cell under function keys for the typing skill row can be marked. The same reasoning applies for direct manipulation column in this row. When compared to other dialog styles (fill-in-forms, question and answers), direct manipulation interfaces sometimes might take longer time to complete a task. For example, consider the situation where the user needs to give his 'id number' to the system. This can be done in different ways:

1. By giving a list of all the 'id numbers' (direct manipulation dialog style) and asking the user to select one of them.

2. By using fill-in-form dialog style and asking the user to type his 'id number'.

Using direct manipulation dialog style in this situation would take much longer time. For applications with high task importance, efficiency in the task to be executed is important. Therefore the cell under this dialog style, for the row task importance should not be marked. Natural language interfaces hide the enhancements from the user. Therefore, due to high task importance, the cell under natural language for the task importance row should not be marked. Figure 5.3 shows the cell matrix at the end of the second pass. The number at the bottom of each column is the sum of all the matches of that dialog style with the user characteristics. From the Figure, we see that the dialog styles that best match the user characteristics of the intended user population of CAS are menus and function keys. The closest competitors are fill-in
forms, question and answers, and natural language dialog styles. Therefore, the cell matrix method suggests menus and function keys as the suitable dialog style for our user domain. However, this initial suggestion must be examined in the context of other factors. Since the frequency of use is low, the system needs to provide a lot of feedback to the user for function key dialog style. Therefore, comparing menus and function keys, the best choice of dialog style would be "menus".

Different dialog styles lend themselves better to different tasks and users. Generally, a system is used by different users performing a variety of tasks. Thus incorporating multiple dialog styles would be advantageous. For example, while some user inputs are best solicited through menus, others cannot be and are best solicited through fill-in fields. While employing multiple dialog styles, they should be "consistently" assigned to actions, in a manner anticipated by users.

The following are the multiple dialog styles that are to be used for the example system CAS. Even though CAS is primarily menu driven, fill-in forms can be used
Figure 5.4: The Initiator & Messenger Interfaces

* To start the advising process: enter your ID and click on OK.
* To exit the system: click on EXIT.

Figure 5.5: The Select Interface

* To get advice from the system: click on Advise.
* If you have any preferences: click on Preferred Choices.
* If you have any constraints: click on Unpreferred Choices.
* To go to the previous window: click on Previous.
* To quit the system: click on Exit.

Figure 5.6: The Preferences Interface
whenever menus are not best suited. Fill-in forms would be appropriate for entering information like the id number, the course number etc.. Therefore, the interactive course advising system CAS, contains menus and fill-in forms as the dialog styles interacting with the user. Since, the key to usability of an interface incorporating multiple dialog styles would be “smooth integration”, consistency in the actions to be performed by the user, is aimed during integration.

5.1.2 Integrating the Multiple Dialog Styles

The multiple dialog styles identified in the previous Section are integrated in a way such that the interfaces designed, are consistent with the actions that are to be
performed by the user. For example, fill-in-forms are used for all user inputs of specifying the course number, time schedule and so on. Figures 5.4 through 5.8 show the multi-dialog-style interfaces for CAS. The start-up interface which appears as soon as the system is invoked is shown in Figure 5.4(a). This interface uses the fill-in-form dialog style for entering the “student id” and menu dialog style for choosing different options available. Using direct manipulation style for choosing the “student id” would be a bad design compared to fill-in-forms. Entering the “student id” with direct manipulation dialog style will take much longer time because the user need to scan the whole list to find the “student id”.

Figure 5.4(b) shows how an error message is communicated. A response to this does not require any input from the keyboard. So, this interface contains only the menu dialog style. Figure 5.5 allows the user to select one of the goals of specifying the preferred choices, unpreferred choices, and requesting for advice, uses menu as a means of interaction with the user. The preferences and constraints interfaces in Figures 5.6 and 5.7, again combines the menu dialog style with fill-in-forms. The advisor interface in Figure 5.8 uses menus for displaying the various options available, to the user.

5.2 Refining the Dialog between the User & the Interface

By now, we have the various user interfaces as seen by the user (designed in Section 5.1). At this point, the dialog between the user and the interface which is specified using the extended state transition diagrams during the analysis phase needs to be updated to correspond to the decisions made during the dialog design. The various user interfaces designed during the dialog design are used to present the output from the system to a user. Since the textual descriptions of the nodes in the state transition diagrams also specify the system output, these textual descriptions are modified to be in correspondence to the dialog designs.
Figure 5.9: Modified Top level state transition diagram of the Course Advising System

Actions:
1. Start up
2. Enter your ID
3. Verify ID
4. Choose

Diagram:

node start

node verify_id

node choose

node exit

node error

node success

node failure

Actions:
- "Options: Select the course from the list.
- "Enter ID: Enter your unique student ID.
- "Verify ID: Confirm the entered ID is correct.
- "Choose: Select the course from the list.
- "Exit: Exit the system.
- "Error: Display an error message.
- "Success: Display a success message.

States:
- start: Initial state
- verify_id: Verify the entered ID
- choose: Choose a course
- exit: Exit the system
- error: Error state
- success: Success state
- failure: Failure state

Transitions:
- From start to verify_id
- From verify_id to choose
- From choose to success
- From verify_id to error
- From choose to failure
- From success to exit
- From failure to exit

Notes:
- Database files:
  - "Preferred Courses" file
  - "Unpreferred Courses" file
- "Err" button: Error button
- "OK" button: OK button
- "Exit" button: Exit button
Figure 5.10: The Modified 'pref choi' sub-conversation
Figure 5.11: The Modified 'unpref choi' sub-conversation
Figure 5.12: The Modified ‘advise’ sub-conversation

While developing the user interface, specifying the positioning of various components of the user interface on a two-dimensional screen is necessary. Therefore, at this stage, the textual descriptions are changed not only to correspond to the decisions made during the dialog design but also to contain information pertaining to the exact or relative position of the components on the screen. The symbols described in Chapter 3 are used to format the system output.

In Figure 5.9, the line:

r2, c., “Welcome to the Automated Course Advising System”

specifies that the welcome message need to be displayed on row two and should be centered in that row. The next line,

r + 1, c1, eol, draw_horiz

specifies that a horizontal line should be drawn on the succeeding row on which the welcome message is specified. The “c1, eol” indicates that the separator should be drawn starting from column1 and ending at the end of the row. The symbol ‘r$1’ used to specify the node “inv_id” indicates that the acknowledge button should be placed on the last but one row in the interface. Therefore, one can easily specify the exact or relative positioning of the output to be displayed. The definition of inputbox
in Figure 5.9, represents a textField widget whose height equals 1 row and width equals 7 columns. Also, one can see how the message name “draw_horiz” has been used at several places, for drawing the separator widget. Figures 5.9 through 5.12 show the modified dialog specifications of the Figures 4.1 through 4.12.

5.3 Static Structure of the Interface

The user interface design phase provides a solution to constructing the interface. Therefore, after designing the appropriate dialog styles, the next step in the design phase of CUIM is to construct the static structure which depicts the designers’ view of the interface. For this purpose, we use the class charts, which depict the spatial organization of the interface classes, and we provide the class descriptions. Jean Marc[Ners90] discusses an object oriented notation BON (Better Object Oriented Notation) for object oriented design of software systems. We have adapted BON for our purposes and called it Simplified Object Notation (SON). The various design activities such as: listing the class charts, depicting the spatial organization, and specifying the class descriptions are carried out using the SON notation. The the SON notation is explained by giving examples in the following subsections.

5.3.1 Listing the Class Charts

All the interface classes identified during the analysis phase are examined for any changes that are necessary. The reasons that would necessitate the changes could be:

1. Extra classes\(^1\) need to be added as a result of the dialogs designed in the previous step.

2. Some of the classes identified during the analysis phase might be redundant.

3. The result of the analysis walk through might demand more interface classes to be added, if the interface group did not model the tasks that have been specified

\(^1\)The term “class” is used as in the case of Object Oriented Modeling.
by the computational group.

After identifying the new list of classes, the interface designer draws a class chart for each class to give a textual description of the class. The different rows in a class chart are:

- **Name**: The first row of a class chart contains the name of the class.

- **Definition**: The row below the class name, defines the class and contains two columns:
  1. **Type Of Object**: This column gives an informal description of the class.
  2. **Behaves Like**: The Behaves Like column states that this object behaves like (inheritance relationship) another type of object.

- **Miscellaneous**: The last row in a class chart lists three basic types of information:
  1. **Questions**: What information can the class ask from the user.
  2. **Commands**: What services can the user ask the class to provide.
  3. **Constraints**: What knowledge must the class maintain.

The values for the columns Behaves Like, Questions, Commands, and Constraints are optional. Considering the example system CAS, it has been found that some of the classes (Prefs Interface, Constraints Interface) identified during the analysis phase are redundant. Both the Prefs Interface and the Constraints Interface which allow the user to specify his preferred/unpreferred courses, time schedule, campus, and workload, can be thought of as different instances of a single class. So these two classes in the analysis phase are now changed to a single class called the *Chooser*. Figure 5.13 shows the new list of classes, by drawing the class charts. The classes *Main Interface* and the *Advisor Interface* (specified in the IAD document) have been renamed as *Selector* and *Advisor* respectively. From Figure 5.13, one can easily understand that the *Initiator* class asks the user to enter his id#, in order to start the advising session. The class chart for the *Selector* class shows that this class allows the user to request
Figure 5.13: Class Charts for the Interface Classes
for advise, request for specifying his preferences/constraints, or to exit the system. The Messenger class needs to have knowledge of the interface class which invoked it via a COB. This is necessary because once the user acknowledges the message, the Messenger should return control(via a COB) to the IOB which invoked it before. From the interfaces designed in Section 5.1.2, we see that the user can specify his choices by invoking the Choicer object either from the Selector interface object or from the Advisor interface object. Depending on from where the Choicer object was invoked, at the end of the user input activity, the control will return to that IOB.

The knowledge of which IOB should be invoked next, must be maintained either by an IOB or a COB. During the analysis phase, it is the user interface group who specifies the order of the tasks that are to be carried out. If the knowledge specified in the “Constraints” column is maintained by a COB, then any changes made by the interface designers to the ordering of the tasks would necessitate these changes to be propagated to the computational group. Therefore, it is logical to maintain this knowledge in an IOB.

5.3.2 Depicting the Spatial Organization of the Interface

The spatial organization captures the physical placement relationship among the various widgets in the interface. A widget is a user interface object which the user sees as a picture on the screen, and the user interface designer sees it as a set of resources and callbacks. The resources lets the designer to control the appearance and behavior of the widget as suited to the user needs. For example, every widget has a “width” and a “height” resource, which determines the width and height of the widget on the screen. A change in the resource during program execution lets the user see a change in the appearance of the widget. The callbacks let the widget communicate with the program as the user performs actions. For example, if the user changes the value of a scale’s slider, the widget recognizes the change, and generates a “valueChanged” callback. This causes the widget to call the specified function.

The SON notation is used in depicting the spatial organization of the user interface. In SON, the spatial organization of the user interface is shown as a graph containing
nodes connected by solid lines. A solid line in the graph represents the relationship between the different nodes in the graph. This relationship can be either "spatial" or "functional". The aggregation and inheritance relationships between any two nodes correspond to the spatial relationship. In SON, aggregation is indicated by a small square which is drawn at the assembly end of the relationship. The node attached to the non-assembly end of the relationship is called the component.

The nodes in the spatial layout are categorized as:

1. Non-Terminal Nodes, and
2. Terminal Nodes

A non-terminal node is represented as an ellipse, if the node corresponds to the name of a class. Otherwise, it is represented as a broken line rectangle. A terminal node is represented as a solid line rectangle. We define a terminal node as: "the node which is not an aggregate node". The component can be either a non-terminal node, or a terminal node.

The inheritance relationship that can exist between a class and one or more refined versions of it is shown by a solid triangle connecting the parent to its descendants. The parent is connected by a line to the apex of the triangle. The descendants are connected to base of the triangle. Deferred classes (abstract classes)[Meye88] are topped with a star sign. Figure 5.14 gives an overview of the SON notations used for depicting the spatial organization of the interface. In SON, the "functional" relationship arc is shown as a directed solid line emanating from a terminal node to a non-terminal node.

The modeling of COBs is not the concern of an interface designer. However, if the COBs do not exist, the IOBs stay apart. In order to connect the different IOBs, the modeling of a COB should be done hand in hand or by simulating the COBs. In CUI/M, simulation of the COBs is done while constructing the spatial organization. This simulation is achieved by labeling the functional relationship arc with the functionality that is to be achieved. The spatial organization does not show any COBs. Only the desired functionality is shown abstractly, irrespective of how
or by whom it is achieved. The desired functionality is indicated as a label on the functional relationship arc.

The spatial organization of the interface is constructed by using the dialog interfaces designed in Section 5.1.2. The interface components designed in Section 5.1.2 correspond to the components in the spatial organization. By using the $SON$ notation described above, Figure 5.15 shows the toplevel spatial view of the course advising system CAS. All GUI applications which are developed on the X window system [Sche92] contains a “toplevel shell” which holds the application. The toplevel shell created in X/MOTIF [Brai92] can hold only one widget. But, most of the applications need to display a number of widgets simultaneously. This problem can be solved by using “manager” widgets [Brai92]. The manager widgets handle the placement of multiple widgets in a single window. Therefore, the toplevel shell widget indirectly holds multiple widgets by handling a single manager widget. Both the toplevel shell and the manager widgets are invisible to the user.

The $Initiator$ class (Figure 5.15) contains the “toplevel_shell”, which in turn contains the “form1” as the manager widget. The widgets hdr_label1, msg_label2, ok_button1, exit_button2, and id_enter_area1 are also the components of the $Initiator(form1)$. An instance of a class communicates with the user through its components. The “invokes” relationship in Figure 5.15 corresponds to the functional relationship arc emanating from a terminal node to a non-terminal node. Figure 5.15 shows the component
Figure 5.15: Interface Classes Spatial Layout - Toplevel View
Figure 5.16: Spatial Layout - of the Choicer, Advisor & Messenger Classes
through which the Selector is invoked. Figure 5.15 also shows the functional relationship between the components of the Initiator class with the Selector and the Messenger. Figures 5.15 and 5.16 show the various components of the different interface classes, and their relationship (if any) with other classes.

5.3.3 Specifying the Class Descriptions

The “user interface system” contains interface components and callback functions. The various classes and their components constructed in the spatial organization will be dummy objects if the user actions are not communicated to the user interface system. Therefore after constructing the interface layout, the next step is to notify the user interface system about the user actions, which is achieved through class descriptions.

The class descriptions in CUIM, describes the attributes, callback functions and invariants if any. The class charts used in the development of class descriptions include the following mapping:

- Commands in class charts are mapped into callbacks.

- Constraints in class charts are mapped into class invariants.

Sometimes the components in the spatial organization need to be changed during the run-time. For example: consider the component which displays the context sensitive help messages. This component need to be changed (during the run-time) as the context changes. Updating the property value of a component is possible if the component is specified as a class attribute. Therefore all the class components in the spatial organization become attributes to the class.

The class descriptions are described using the SON notation. In SON each class is described with its header and a body. The header contains the class name. The body contains attributes (properties of the objects in the class), callback functions and class invariants.
Figure 5.17: Class Descriptions: set one
Figure 5.18: Class Descriptions: set two
* Attributes are described according to the syntax:

\[ \text{attribute : TYPE} \]

* Callbacks are described following the syntax:

\[
\text{callback name( arg1: TYPE, arg2: TYPE, ..., argn: TYPE )}
\{
\}
\text{-- callback ends}
\]

* Abstract functions are preceded by a "**" sign.

* The invariant appears in a clause titled by the keyword \text{invariant}.

* Comments are written by beginning with "*/" and ending with "*/".

Figures 5.17 and 5.18 show the class descriptions for each of the classes listed in the \text{class charts}.

## 5.4 System Interactions

The interaction diagrams during the analysis phase specify abstractly the interactions between the user, the IOBs and the COBs. These interaction diagrams are further refined during the design phase by specifying the internal behavior (Section 5.4.1) of the different interface objects, and by specifying the interaction relationship (Section 5.4.2) among the various objects in the system.

### 5.4.1 Behavioral Specification

The class descriptions specified in Section 5.3.3 give a template of the class by listing the various callback functions that the class contains. The behavior of the class when its callbacks are triggered needs to be specified. Therefore, the next step during
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>corresponds to the name of the class</td>
</tr>
<tr>
<td>@</td>
<td>represents the ports at which the interaction with other classes can happen</td>
</tr>
<tr>
<td>event ?</td>
<td>represents an input event</td>
</tr>
<tr>
<td>event !</td>
<td>represents an output event</td>
</tr>
<tr>
<td>{attr1, attr2}</td>
<td>represents the attributes that are active, in the state to which they are associated</td>
</tr>
<tr>
<td>![state name]</td>
<td>represents the initial state</td>
</tr>
<tr>
<td>![state name]</td>
<td>represents an intermediate/end state</td>
</tr>
</tbody>
</table>

Figure 5.19: Notations used in TROM

the design phase in CUMI, is to specify the behavior of the class when its callbacks are triggered. In an event driven application, a callback is triggered in response to an event from the user. Therefore, the behavior of the class with respect to the event received is specified using the TROM model (Timed Reactive Object Model), developed in the doctoral research of [Achu94].

A TROM is a finite state machine augmented with attributes, time constraints and logical assertions. In TROM, the transitions are labeled by events, and, these events form the fundamental message components of an interaction of an object with its environment. The attributes model the data computations associated with the transitions. Each transition is associated with three assertions:

1) Pre-condition: The conditions under which the transition will be initiated are specified in the pre-condition.
2) Post-condition: The data computation that is associated with the transition is specified in the post-condition.
3) Port-condition: The port at which an interaction can happen is specified in the port-condition.

Figure 5.19 gives a description of the notations used in the TROM model.
Communication between any two TROMs is based on synchronous message passing. The port associated to a TROM specifies the interaction ports of that TROM with its environment. A port is a bi-directional communication channel between a TROM and its environment. There can be multiple ports associated to a TROM. Each port has a unique port-type and multiple port-types can be associated with a TROM. The port-type determines all the set of messages and the possible sequences that are allowed to communicate with that port.

The TROM model in [Achu94] is based on a three-tiered approach for the specification of system design. The top-most tier describes the interaction relationship that can exist between the objects in a system. The middle tier constituting the detailed specification of the objects used in the system architecture, is given using the TROM model. The bottom most tier specifying the data abstractions used in the class definitions of the middle tier by means of the Larch Shared Language (LSL) [Gutt91].

During the user interface design in CULM, the data abstractions used in the class definitions in TROM, are already specified in [Visa93]. The user interface designer, therefore specifies the behavior of the class for each of the user event the class receives. The behavior of the Initiator class for different events the class receives from the user is shown in Figure 5.20. Initially the Initiator will be in state wait. The Initiator goes from state wait to state active, when it receives the invoke_I event from cas (a COB). After receiving the ok event from the User, the Initiator goes into state check to check the student id given by the user. At this instance, the Initiator sends the id value entered by the user to the transcript object (which is a COB) and requests it to verify the student id. The Initiator then comes to state wait from state check. The Initiator comes to state invoke by receiving either valid or invalid event from the transcript. If the event received is valid, then the Initiator sends the event invoke_S to cas (which is a COB) and comes to wait state. Otherwise, the Initiator sends the event invoke_M to the transcript and comes to wait state. The Initiator comes back to active state only when it receives the invoke_I event from cas. When the user gives the exit event, the Initiator notifies cas that the system needs to be terminated and comes to end state. The Initiator also comes to end state, when it receives the exit event from cas. The end state marks the state in which the object is killed. In the notation used in TROM, the reserved word trashed indicates that the object is killed.
The TROMs for the Messenger, Advisor, Selector, and Choicer are shown in Figures 5.21 through 5.24. The behavioral specification of the User is shown in Figure 5.25.

5.4.2 User Interface Configuration

As a next step in the design phase of CUIM, the user interface designer constructs the User Interface Configuration (UIC) [AARV95]. The UIC specifies the possible interaction relationship that can exist between the user, the IOBs and the COBs. Since CUIM does not support the interaction of any two IOBs, no interaction relationship exists between any two IOBs in the UIC. The UIC shown in Figure 5.26 defines the user interface system by composing objects instantiated from the classes described before. The different Sections in the UIC specification are:

1. The Include section lists the system/subsystem definitions imported from other user interface configurations. This section is optional.

2. The Instantiate section specifies the instantiation relationship between the objects and their classes. The cardinality of each type of port associated to the class and the values for the attributes if any, which needs to be initialized are also specified when an object is instantiated. For instance in Figure 5.26, i1 is an instantiation of the Initiator and has one port of type A1, one port of type X and so on. There are no initialization attributes for Initiator. Considering the Choicer class, we see that the Choicer has n instances c1, ..., cn and has one port of type C1, one port of type P, and n ports of type M. The initialization attributes for Choicer are win and valid. The other classes and their object instances together with their initialization attributes if any are shown in Figure 5.26.

3. The Configure section defines the user interface system by using objects specified in the Instantiate section. The operator "→" is used to link the respective ports of each class. The basic relationships that can exist between objects of two classes are one-to-one, one-to-many, and many-to-many. For example, to specify a one-to-many relationship which exists between the objects of the Selector
Class Initiator [A1,Ax,G1]

Events: ok(id.val)?, valid?, invalid?, invoke_I?,
verify(id), invoke_S!, invoke_M!, exit?, exit!

State: *wait, check, active, invoke, end, exit

Attributes: flag : boolean, id : integer

Attribute-function:
  check -> id; wait -> flag;
  end, exit -> {};
  active, invoke -> flag, toplevel_shell;

Transition Spec:
R1 : {active, check}; ok(id.val)?(pid : A1);
  true -> id' = id.val;
R2 : {check, wait}; verify(id)!
  (pid : X); true -> flag' = flag;
R3 : {wait, invoke}; valid?(pid : X);
  true -> flag' = flag∧
  UxPopdownInterface(toplevel_shell);
/* The behavior of UxPopdownInterface is
given in [Visa93] */
R4 : {wait, invoke}; invalid?(pid : X);
  true -> toplevel_shell' = toplevel_shell∧
  flag' = FALSE;
R5 : {invoke, wait}; invoke_S!(pid : G1);
  flag = TRUE -> flag' = flag;
R6 : {invoke, wait}; invoke_M!(pid : G1);
  flag = FALSE -> flag' = flag;
R7 : {wait, active}; invoke_I?(pid : G1);
  flag = TRUE -> flag' = flag∧
  UxPopupInterface(toplevel_shell, no_grab)
/* The behavior of UxPopupInterface is
given in [Visa93] */
R8 : {wait, active}; invoke_I?(pid : G1);
  flag = FALSE -> flag' = TRUE∧
  toplevel_shell' = toplevel_shell;
R9 : {active, exit}; exit?(pid : A1); --
R10 : {exit, end}; exit!(pid : G1);
  true -> trashed;
R11 : {wait, end}; exit?(pid : G1);
  true -> trashed;

end

Figure 5.20: Initiator Behavior Specification
Class \textit{Messenger} [@D1, @Q, @L]

Events: ok?, invoke_M(type.val)?, invoke_Ch!, invoke_A!, invoke_I!, exit?

State: *wait, active, invoke, end

Attributes: intr_prt1 : @L, msg_type : integer

Attribute-function:
\begin{itemize}
  \item \texttt{wait, end} $\mapsto \emptyset$; \texttt{invoke} $\mapsto \text{form4}$;
  \item \texttt{active} $\mapsto \text{msg.type, intr_prt1, form4}$;
\end{itemize}

Transition Spec:
\begin{itemize}
  \item $R_1 : \langle \text{active, invoke} \rangle$; ok?(pid : @D1);
  \item $\text{true} \rightarrow \text{UserPopdownInterface(form4)}$;
  \item $R_2 : \langle \text{invoke, wait} \rangle$; invoke_I!(pid : @Q);
  \item $\text{msg.type} = 1 \rightarrow \text{true}$;
  \item $R_3 : \langle \text{invoke, wait} \rangle$; invoke_Ch!
  \item (pid = intr_prt1); \text{msg.type} = 2 \rightarrow \text{true}$;
  \item $R_4 : \langle \text{invoke, wait} \rangle$; invoke_A!(pid : @Q);
  \item $\text{msg.type} = 3 \rightarrow \text{true}$;
  \item $R_5 : \langle \text{wait, active} \rangle$; invoke_M(type.val)?
  \item (pid : @Q); \text{true} $\rightarrow \text{msg.type'} = \text{type.val} \wedge$
  \item intr_prt1' = intr_prt1$
  \item $\text{UserPopdownInterface(form4, no_grab)}$;
  \item $R_6 : \langle \text{wait, active} \rangle$; invoke_M(type.val)?
  \item (pid : @L); \text{true} $\rightarrow \text{msg.type'} = \text{type.val} \wedge$
  \item intr_prt1' = pid$
  \item $\text{UserPopdownInterface(form4, no_grab)}$;
  \item $R_7 : \langle \text{wait, end} \rangle$; exit?(pid : @Q);
  \item \text{true} $\rightarrow \text{trashed}$;
\end{itemize}

Figure 5.21: \textit{Messenger} Behavior Specification
Class *Advisor* [E1, O]

Events: `print?`, `unprefs?`, `prefs?`,
`exit?`, `done?`, `print!`, `invoke_C!`,
`invoke_A[list]?`, `invoke_M!`, `exit!`

State: `*wait, active, invoke, approval, end, exit`

Attributes: `win, type : integer`

Attribute-function:

- `end, approval, wait, exit -> {}`
- `invoke -> win, type, form5`
- `active -> form5`

Transition Spec:

- **R1**: `(active, invoke); prefs?(pid : E1);
  \[\text{true} \rightarrow \text{win}' = 1 \land \text{type}' = 1\land
  \text{form5}' = \text{form5};\]
- **R2**: `(active, invoke); unprefs?(pid : E1);
  \[\text{true} \rightarrow \text{win}' = 1 \land \text{type}' = 2\land
  \text{form5}' = \text{form5};\]
- **R3**: `(active, exit); exit?(pid : E1); --
- **R4**: `(active, approval); print?(pid : E1); --
- **R5**: `(approval, wait); print!(pid : O); --
- **R6**: `(wait, invoke); done?(pid : O);
  \[\text{true} \rightarrow \text{win}' = 2 \land \text{type}' = \text{type}
  \land \text{form5}' = \text{form5};\]
- **R7**: `(invoke, wait); invoke_M!(pid : O);
  \[\text{win} = 2 \rightarrow \text{true};\]
- **R8**: `(invoke, wait); invoke_C[type]!(pid : O);
  \[\text{win} = 1 \rightarrow \text{true};\]
- **R9**: `(exit, end); exit!(pid : O);
  \[\text{true} \rightarrow \text{trashed};\]
- **R10**: `(wait, active); invoke_A[list]?(pid : O);
  \[\text{true} \rightarrow \text{InvokeInterface(form5, no_grab);}\]
- **R11**: `(wait, end); exit?(pid : O);
  \[\text{true} \rightarrow \text{trashed};\]

end

Figure 5.22: Advisor Behavior Specification
Class Selector [@B1, @T, @N]

Events: previous?, prefs?, unprefs?,
advise?, invoke_S?, exit?, invoke.C(type!),
invoke.A!, invoke.J!, exit!

State: *wait, invoke, active, end, exit

Attributes: win, type: integer,
intr.prt1 : @N

Attribute-function:
end, exit => \{\}; wait => intr.prt1;
invoke => win, type, form2; active => form2;

Transition Spec:
R1 : (active, invoke); previous?(pid : @B1);
true -> win' = 1 \land type' = type \land
UxPopdownInterface(form2);
R2 : (active, invoke); prefs?(pid : @B1);
true -> win' = 2 \land type' = 1 \land
UxPopdownInterface(form2);
R3 : (active, invoke); unprefs?(pid : @B1);
true -> win' = 2 \land type' = 2 \land
UxPopdownInterface(form2);
R4 : (active, invoke); advise?(pid : @B1);
true -> win' = 3 \land type' = type \land
UxPopdownInterface(form2);
R5 : (active, exit); exit?(pid : @B1); --
R6 : (invoke, wait); invoke.J!(pid : @T);
win = 1 -> intr.prt1' = intr.prt1;
R7 : (invoke, wait); invoke.C(type)!(pid : @N);
win = 2 -> intr.prt1' = pid;
R8 : (invoke, wait); invoke.A!(pid : @T);
win = 3 -> intr.prt1' = intr.prt1;
R9 : (wait, active); invoke.S?
(pid : @T) ->
UxPopdownInterface(form2, no_grab);
R10 : (wait, active); invoke.S?
(pid = intr.prt1) ->
UxPopdownInterface(form2, no_grab);
R11 : (exit, end); exit!(pid : @T);
true -> trashed;
R12 : (wait, end); exit?(pid : @T);
true -> trashed;

end
Class Choice [[@C1, @M, @P]]

Events: accept?, cancel?, course.val?,
valid?, invalid?, verify(course)?,
t.val?, macro.val?, macro.val?, mcr.val?,
mcr.val?, cmp1.val?, cmp2.val?, invoke_C?
invoke_S(t.list, course, macr, micr, macro, mico, cmp1, cmp2)!
invoke_A(t.list, course, macr, micr, macro, mico, cmp1, cmp2)!
invoke_M!, exit?

State: *wait, active, invoke, verify, end

Attributes: win: integer; valid: boolean;
macr, micr, macro, mico: string; cmp1, cmp2: boolean;
course, t.list: string;

Attribute-function:
invoke → valid, cmdn.box1, cmdn.box2,
macro.enter.area2, macro.enter.area3,
macro.enter.area4, micr.enter.area5,
campus.togglebutton1, campus.togglebutton2, form3;
active → win, valid, form3;
verify → course; wait, end → {};

Transition Spec:
R1: (active, invoke); accept?(pid @C1);
true → cmdn.box1' = cmdn.box1 ∧
cmdn.box2' = cmdn.box2 ∧
macro.enter.area2' = macro.enter.area2 ∧
macro.enter.area3' = macro.enter.area3 ∧
macro.enter.area4' = macro.enter.area4 ∧
micr.enter.area5' = micr.enter.area5 ∧
campus.togglebutton1' = campus.togglebutton1 ∧
campus.togglebutton2' = campus.togglebutton2 ∧
valid' = valid ∧ UxPopdownInterface(form3);
R2: (invoke, wait); invoke_S(t.list, course,
macr, micr, macro, mico, cmp1, cmp2)!
(pid @M); valid = TRUE ∧ win = 1 → true;
R3: (wait, active); invoke.C?(pid @M);
valid = TRUE → valid' = valid ∧ win' = 1 ∧
UxPopupInterface(form3, no.grab);
R4: (invoke, wait); invoke_A(t.list, course,
macr, micr, macro, mico, cmp1, cmp2)!
(pid @P); valid = TRUE ∧ win = 2 → true;
R5: (wait, active); invoke_C?(pid @P);
true → win' = 2 ∧ valid' = valid ∧
UxPopupInterface(form3, no.grab);
\[ R_6 : (\text{active}, \text{invoke}); \text{cancel?}(\text{pid}: @C1); \text{true} \rightarrow \text{valid}' = \text{valid} \wedge \]
\[ \text{UxPutListItems(cmnd.box2, t.list)} \wedge \]
\[ / * \text{The behavior of UxPutListItemsis} \]
\[ \text{given in [Visa93]} * / \]
\[ \text{UxPutListItems(cmnd.box1, course)} \wedge \]
\[ \text{UxPutListItems(cmnd.box2, t.list)} \wedge \]
\[ \text{UxPutText(maco.enter.area2, maco)} \wedge \]
\[ / * \text{The behavior of UxPutTextis} \]
\[ \text{given in [Visa93]} * / \]
\[ \text{UxPutText(mico.enter.area3, mico)} \wedge \]
\[ \text{UxPutText(macro.enter.area4, macr)} \wedge \]
\[ \text{UxPutText(micr.enter.area5, micr)} \wedge \]
\[ \text{UxPutSet(campus.togglebutton1, cmp1)} \wedge \]
\[ / * \text{The behavior of UxPutSetis} \]
\[ \text{given in [Visa93]} * / \]
\[ \text{UxPutSet(campus.togglebutton2, cmp2)} \wedge \]
\[ \text{\& UxPopdownInterface(form3)}; \]
\[ R_7 : (\text{active}, \text{verify}); \text{course.val?}(\text{pid}: @C1); \text{true} \rightarrow \text{course}' = \text{course.val} ; \]
\[ R_8 : (\text{verify}, \text{wait}); \text{verify(course)}! \]
\[ (\text{pid}: @M) ; -- \]
\[ R_9 : (\text{wait}, \text{active}); \text{valid?}(\text{pid}: @M); \text{true} \rightarrow \text{valid}' = \text{valid} \wedge \text{win}' = \text{win} \wedge \]
\[ \text{form3}' = \text{form3}; \]
\[ R_{10} : (\text{wait}, \text{invoke}); \text{invalid?}(\text{pid}: @M); \text{true} \rightarrow \text{valid}' = \text{FALSE}; \]
\[ R_{11} : (\text{invoke}, \text{wait}); \text{invoke.M}!(\text{pid}: @M); \text{valid} = \text{FALSE} \rightarrow \text{true}; \]
\[ R_{12} : (\text{wait}, \text{active}); \text{invoke.C?}(\text{pid}: @M); \text{valid} = \text{FALSE} \rightarrow \text{valid}' = \text{TRUE} \wedge \]
\[ \text{win}' = \text{win} \wedge \text{form3}' = \text{form3}; \]
\[ R_{13} : (\text{active}, \text{active}); \text{t.val?}(\text{pid}: @C1); \text{true} \rightarrow \text{win}' = \text{win} \wedge \text{valid}' = \text{valid} \wedge \]
\[ \text{form3}' = \text{form3}; \]
\[ R_{14} : (\text{active}, \text{active}); \text{maco.val?}(\text{pid}: @C1); \text{true} \rightarrow \text{win}' = \text{win} \wedge \text{valid}' = \text{valid} \wedge \]
\[ \text{form3}' = \text{form3}; \]
\[ R_{15} : (\text{active}, \text{active}); \text{mico.val?}(\text{pid}: @C1); \text{true} \rightarrow \text{win}' = \text{win} \wedge \text{valid}' = \text{valid} \wedge \]
\[ \text{form3}' = \text{form3}; \]
\[ R_{16} : (active, active); macr._val?(pid:@C1); true \rightarrow win' = win \land valid' = valid \land form3' = form3; \]
\[ R_{17} : (active, active); micr._val?(pid:@C1); true \rightarrow win' = win \land valid' = valid \land form3' = form3; \]
\[ R_{18} : (active, active); cmp1._val?(pid:@C1); true \rightarrow win' = win \land valid' = valid \land form3' = form3; \]
\[ R_{19} : (active, active); cmp2._val.(pid:@C1); true \rightarrow win' = win \land valid' = valid \land form3' = form3; \]
\[ R_{20} : (wait, end); exit?(pid:@P); true \rightarrow \text{trashed}; \]

end

Figure 5.24: Choicer Behavior Specification

and the user constraint classes, the class \textit{Selector}[@N] is instantiated with the cardinality for the port N as \( n \) and the class \textit{user constraint}[@J] is instantiated with the cardinality for the port J as \( one \). Then the ports N and J are linked as follows:

\[ s_{1}.@N, \leftrightarrow u_{s_{1}}.@J_{1} \]

Figure 5.26 shows how the respective ports of interaction, between objects of two classes are linked.

To achieve clarity in depicting the interactions between IOBs and COBs, the same instance of a particular COB is redrawn. That is, the \textit{cas} object in Figure 5.26, interacting with the \textit{Initiator} is the same \textit{cas} object that is interacting with the \textit{Messenger}, although it is drawn many times. Similarly, the \textit{user} object even though drawn many times, represents a single instance of the \textit{User} class. Multiple instances if any, of an interface class are shown as a double square. And, multiple instances if any, of a computational class are shown as a double circle. Therefore, from Figure 5.26, we understand that only the \textit{Choicer} and the \textit{user constraint} classes can have more than one instance created.

121
Class User [$\oplus U, \oplus V, \oplus B, \oplus Z, \oplus C$]

Events: $\text{ok}(\text{id\_val})!, \text{previous}!, \text{prefs}!, \text{unprefs}!, 
\text{advise}!, \text{ok}!, \text{course\_val}!, \text{time}!, \text{macal}, \text{mico}, \text{macr}!, \text{micr}!, 
\text{cmp1}!, \text{cmp2}!, \text{accept}!, \text{cancel}!, \text{print}!, \text{display}S?, \text{display}I?, 
\text{display}C?, \text{display}A?, \text{display}M?, \text{exit}!

State: active, *wait, end

Attributes: $\text{win : integer}; \text{intr\_prt3 : @B};$

Attribute-function:

\[
\text{wait, end} \mapsto \{\}; \text{active} \mapsto \text{win}, \text{intr\_prt3};
\]

Transition Spec:

\[
\begin{align*}
R_1 & : (\text{active}, \text{wait}) ; \text{ok}(\text{id\_val})!(\text{pid : @U}); \\
\text{win} & = 1 \rightarrow \text{true} \\
R_2 & : (\text{active}, \text{wait}) ; \text{previous}!(\text{pid : @V}); \\
\text{win} & = 2 \rightarrow \text{true}; \\
R_3 & : (\text{active}, \text{wait}) ; \text{prefs}!(\text{pid : @V}); \\
\text{win} & = 2 \rightarrow \text{true}; \\
R_4 & : (\text{active}, \text{wait}) ; \text{unprefs}!(\text{pid : @V}); \\
\text{win} & = 2 \rightarrow \text{true}; \\
R_5 & : (\text{active}, \text{wait}) ; \text{advise}!(\text{pid : @V}); \\
\text{win} & = 2 \rightarrow \text{true};
\end{align*}
\]

\[
\begin{align*}
R_6 & : (\text{active}, \text{wait}) ; \text{ok}!(\text{pid : @Z}); \\
\text{win} & = 5 \rightarrow \text{true}; \\
R_7 & : (\text{active}, \text{wait}) ; \text{course\_val}!(\text{pid = intr\_prt3}); \\
\text{win} & = 3 \rightarrow \text{true}; \\
R_8 & : (\text{active}, \text{active}) ; \text{time}!(\text{pid = intr\_prt3}); \\
\text{win} & = 3 \rightarrow \text{true}; \\
R_9 & : (\text{active}, \text{active}) ; (\text{macal} \lor \text{mico} \lor \text{macr}! \lor \text{micr}!)!(\text{pid = intr\_prt3}); \text{win} = 3 \rightarrow \text{true}; \\
R_{10} & : (\text{active}, \text{active}) ; (\text{cmp1}! \lor \text{cmp2}!)(\text{pid = intr\_prt3}); \text{win} = 3 \rightarrow \text{true}; \\
R_{11} & : (\text{active}, \text{wait}) ; \text{accept}!(\text{pid = intr\_prt3}); \\
\text{win} & = 3 \rightarrow \text{true}; \\
R_{12} & : (\text{active}, \text{wait}) ; \text{cancel}!(\text{pid = intr\_prt3}); \\
\text{win} & = 3 \rightarrow \text{true}; \\
R_{13} & : (\text{active}, \text{wait}) ; \text{print}!(\text{pid : @C}); \\
\text{win} & = 4 \rightarrow \text{true}; \\
R_{14} & : (\text{active}, \text{wait}) ; \text{prefs}!(\text{pid : @C}); \\
\text{win} & = 4 \rightarrow \text{true}; \\
R_{15} & : (\text{active}, \text{wait}) ; \text{unprefs}!(\text{pid : @C}); \\
\text{win} & = 4 \rightarrow \text{true};
\end{align*}
\]

122
\[ R_{16} : \langle \text{wait}, \text{active} \rangle; \text{displayS?(pid:@V)}; \\
\text{true} \rightarrow \text{win'} = 2 \land \text{intr.prt3'} = \text{intr.prt3}; \\
R_{17} : \langle \text{wait}, \text{active} \rangle; \text{displayI?(pid:@V)}; \\
\text{true} \rightarrow \text{win'} = 1 \land \text{intr.prt3'} = \text{intr.prt3}; \\
R_{18} : \langle \text{wait}, \text{active} \rangle; \text{displayA?(pid:@C)}; \\
\text{true} \rightarrow \text{win'} = 4 \land \text{intr.prt3'} = \text{intr.prt3}; \\
R_{19} : \langle \text{wait}, \text{active} \rangle; \text{displayM?(pid:@Z)}; \\
\text{true} \rightarrow \text{win'} = 5 \land \text{intr.prt3'} = \text{intr.prt3}; \\
R_{20} : \langle \text{wait}, \text{active} \rangle; \text{displayC?(pid:@B)}; \\
\text{true} \rightarrow \text{win'} = 3 \land \text{intr.prt3'} = \text{pid} \\
R_{21} : \langle \text{active}, \text{end} \rangle; \text{exit!(pid:@U\lor pid:@V \lor pid:@C)}; \text{win} = 1 \lor \text{win} = 2 \\
\forall \text{win} = 4 \rightarrow \text{true}; \]

end

Figure 5.25: User Behaviour with different IOBs

Since TROM model helps us to formally specify: (1) the various states the class undergoes and (2) the communication between the various objects in the system via ports, CUIM suggests that during the design phase both the Interface Group and the Computational Group should use the TROM model to specify the behavior of the class.

5.5 Reviewing the Design

By reviewing the design we ensure that the user's view of the interface (Dialog Design) is translated properly to the system view (Spatial Organization). The system view is then refined such that, each user action is notified to the system (Class Descriptions) and the behavior of the system in response to each user action is clearly specified (Behavioral Specifications). We also ensure, that the output events of the User Class (Behavioral Specifications) are input events to another interface class (Behavioral Specifications). These translations are achieved as a four step process:

• **Step 1:** Ensure that the class components in the spatial organization "corresponds" to the interface components specified during the dialog design. To
UICS System

Include:

Instantiate:

\[ u_1 :: User[@U : 1, @V : 1, @B : n, @Z : 1, @C : 1]; \]
\[ i_1 :: Initiator[@A1 : 1, @X : 1, @G1 : 1]; (flag = TRUE); \]
\[ s_1 :: Selector[@B1 : 1, @T : 1, @N : n]; \]
\[ m_1 :: Messenger[@D1 : 1, @Q : 1, @L : n]; \]
\[ c_1, \ldots, c_n :: Choice[@C1 : 1, @P : 1, @M : n]; (valid' = TRUE); \]
\[ a_1 :: Advisor[@E1 : 1, @O : 1]; \]
\[ tr_1 :: transcript[@Y : 1]; \]
\[ ca_1 :: cas[@K : 1, @I : n, @F : 1, @D : 1, @H : 1]; \]
\[ us_1, \ldots, us_n :: userconstraint[@J : 1, @H : n, @E : 1]; \]

Configure:

\[ \forall i \in 1 \ldots n \]
\[ u_1.@U_i \leftrightarrow i_1.@A1_i \]
\[ u_1.@Z_i \leftrightarrow m_1.@D1_i \]
\[ u_1.@V_i \leftrightarrow s_1.@B1_i \]
\[ u_1.@B_i \leftrightarrow c_i.@C1_i \]
\[ u_1.@C_i \leftrightarrow a_i.@E1_i \]
\[ i_1.@X_i \leftrightarrow tr_1.@Y_i \]
\[ i_1.@G1_i \leftrightarrow ca_1.@H1_i \]
\[ m_1.@L_i \leftrightarrow us_i.@E_i \]
\[ m_1.@Q_i \leftrightarrow ca_1.@F_i \]
\[ s_1.@N_i \leftrightarrow us_i.@J_i \]
\[ s_1.@T_i \leftrightarrow ca_1.@K_i \]
\[ c_i.@P_i \leftrightarrow ca_1.@I_i \]
\[ c_i.@M_i \leftrightarrow us_i.@H_i \]
\[ a_i.@O_i \leftrightarrow ca_1.@D_i \]

end

Figure 5.26: User Interface Configuration Specification - Course Advising System
<table>
<thead>
<tr>
<th>User View</th>
<th>Designer View</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Welcome to the Automated Course Advising System;</td>
<td>hdr_label1</td>
<td>Initiator</td>
</tr>
<tr>
<td>Enter your ID;</td>
<td>msg_label2</td>
<td></td>
</tr>
<tr>
<td>OK;</td>
<td>ok_button1</td>
<td></td>
</tr>
<tr>
<td>EXIT;</td>
<td>exit_button2</td>
<td></td>
</tr>
<tr>
<td>user types the id here;</td>
<td>id_enter_area1</td>
<td></td>
</tr>
<tr>
<td>online help messages;</td>
<td>msg_label34</td>
<td></td>
</tr>
<tr>
<td>a line separating the help message display area from the push buttons on</td>
<td>hseparator6</td>
<td></td>
</tr>
<tr>
<td>the top;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Previous;</td>
<td>prev_button3</td>
<td>Selector</td>
</tr>
<tr>
<td>Advise;</td>
<td>advise_button4</td>
<td></td>
</tr>
<tr>
<td>Preferred Choices;</td>
<td>prefs_button5</td>
<td></td>
</tr>
<tr>
<td>Unpreferred Choices;</td>
<td>unprefs_button6</td>
<td></td>
</tr>
<tr>
<td>Exit;</td>
<td>exit_button7</td>
<td></td>
</tr>
<tr>
<td>online help messages;</td>
<td>msg_label3</td>
<td></td>
</tr>
<tr>
<td>a horizontal line separating the help messages area from the buttons on</td>
<td>hseparator1</td>
<td></td>
</tr>
<tr>
<td>the buttons above it;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system generated message;</td>
<td>err_msg_label33</td>
<td>Messenger</td>
</tr>
<tr>
<td>OK;</td>
<td>ok_button19</td>
<td></td>
</tr>
<tr>
<td>advisor message area;</td>
<td>msg_label26</td>
<td>Advisor</td>
</tr>
<tr>
<td>Course#;</td>
<td>course_label27</td>
<td></td>
</tr>
<tr>
<td>Course Name;</td>
<td>course_name_label28</td>
<td></td>
</tr>
<tr>
<td>Instructor;</td>
<td>instructor_label29</td>
<td></td>
</tr>
<tr>
<td>Time;</td>
<td>time_label30</td>
<td></td>
</tr>
<tr>
<td>Term;</td>
<td>term_label31</td>
<td></td>
</tr>
<tr>
<td>area which displays the suggested course list;</td>
<td>suggested_list_label32</td>
<td></td>
</tr>
<tr>
<td>Print;</td>
<td>print_button10</td>
<td></td>
</tr>
<tr>
<td>Preferred Choices;</td>
<td>preferences_button11</td>
<td></td>
</tr>
<tr>
<td>Unpreferred Choices;</td>
<td>constraints_button13</td>
<td></td>
</tr>
<tr>
<td>Exit;</td>
<td>exit_button12</td>
<td></td>
</tr>
<tr>
<td>a line separating the advisor message area from the suggested course</td>
<td>hseparator5</td>
<td></td>
</tr>
<tr>
<td>list;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

contd.
<table>
<thead>
<tr>
<th>User View</th>
<th>Designer View</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preferred Course List;</td>
<td>course_label4</td>
<td>Choicer</td>
</tr>
<tr>
<td>Enter Course Number;;</td>
<td>cmnd_box1</td>
<td></td>
</tr>
<tr>
<td>Preferred Time List;</td>
<td>time_label5</td>
<td></td>
</tr>
<tr>
<td>Enter Time:</td>
<td>cmnd_box2</td>
<td></td>
</tr>
<tr>
<td>Select Preferred Campus;</td>
<td>campus_label9</td>
<td></td>
</tr>
<tr>
<td>toggle button for SGW;</td>
<td>campus_togbutton1</td>
<td></td>
</tr>
<tr>
<td>toggle button for Loyola;</td>
<td>campus_togbutton2</td>
<td></td>
</tr>
<tr>
<td>a horizontal line separating the course list from the time list and campus</td>
<td>hseparator2</td>
<td></td>
</tr>
<tr>
<td>specification area;</td>
<td>vseparator1</td>
<td></td>
</tr>
<tr>
<td>a vertical line separating the time list and the campus specification area;</td>
<td>workload_label10</td>
<td></td>
</tr>
<tr>
<td>Enter Prefered Workload;</td>
<td>max_course_label11</td>
<td></td>
</tr>
<tr>
<td>Maximum # of courses;;</td>
<td>maco_enter_area2</td>
<td></td>
</tr>
<tr>
<td>user types the maximum number of courses he prefers;</td>
<td>min_course_label12</td>
<td></td>
</tr>
<tr>
<td>Minimum # of courses;;</td>
<td>mico_enter_area3</td>
<td></td>
</tr>
<tr>
<td>user types the minimum number of courses he prefers;</td>
<td>max_credit_label13</td>
<td></td>
</tr>
<tr>
<td>Maximum # of credits;;</td>
<td>macr_enter_area4</td>
<td></td>
</tr>
<tr>
<td>user types the maximum number of credits he prefers;</td>
<td>min_credit_label14</td>
<td></td>
</tr>
<tr>
<td>Minimum # of credits;;</td>
<td>micr_enter_area5</td>
<td></td>
</tr>
<tr>
<td>user types the minimum number of credits he prefers;</td>
<td>hseparator3</td>
<td></td>
</tr>
<tr>
<td>a horizontal line separating the workload specification from the time list</td>
<td>accept_button8;</td>
<td></td>
</tr>
<tr>
<td>and campus specification area;</td>
<td>cancel_button9;</td>
<td></td>
</tr>
<tr>
<td>Accept</td>
<td>hseparator4</td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a horizontal line separating the push buttons area from the choices specific area;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Dual view Association Table
<table>
<thead>
<tr>
<th>Class</th>
<th>Questions</th>
<th>Class Component</th>
<th>Class Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>student id</td>
<td>hdr_label1</td>
<td>hdr_label1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg_label2</td>
<td>msg_label2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ok_button1</td>
<td>ok_button1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exit_button2</td>
<td>exit_button2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>id_enter_area1</td>
<td>id_enter_area1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg_label34</td>
<td>msg_label34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hseparator6</td>
<td>hseparator6</td>
</tr>
<tr>
<td>Selector</td>
<td></td>
<td>prev_button3</td>
<td>prev_button3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>advise_button4</td>
<td>advise_button4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>prefs_button5</td>
<td>prefs_button5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unprefs_button6</td>
<td>unprefs_button6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exit_button7</td>
<td>exit_button7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>msg_label3</td>
<td>msg_label3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hseparator1</td>
<td>hseparator1</td>
</tr>
<tr>
<td>Messenger</td>
<td></td>
<td>err_msg_label33</td>
<td>err_msg_label33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ok_button19</td>
<td>ok_button19</td>
</tr>
<tr>
<td>Advisor</td>
<td></td>
<td>msg_label26</td>
<td>msg_label26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>course_label27</td>
<td>course_label27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>course_name_label28</td>
<td>course_name_label28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instructor_label29</td>
<td>instructor_label29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time_label30</td>
<td>time_label30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>term_label31</td>
<td>term_label31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>suggested_list_label32</td>
<td>suggested_list_label32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print_button10</td>
<td>print_button10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>preferences_button11</td>
<td>preferences_button11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>constraints_button13</td>
<td>constraints_button13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exit_button12</td>
<td>exit_button12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hseparator5</td>
<td>hseparator5</td>
</tr>
</tbody>
</table>

cont'd.
<table>
<thead>
<tr>
<th>Class</th>
<th>Questions</th>
<th>Class Component</th>
<th>Class Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choicer</td>
<td></td>
<td>course_label4</td>
<td>course_label4</td>
</tr>
<tr>
<td></td>
<td>courses</td>
<td>cmdnd_box1</td>
<td>cmdnd_box1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time_label5</td>
<td>time_label5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cmdnd_box2</td>
<td>cmdnd_box2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>campus_label9</td>
<td>campus_label9</td>
</tr>
<tr>
<td></td>
<td>campus</td>
<td>campus_toglbutton1</td>
<td>campus_toglbutton1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>campus_toglbutton2</td>
<td>campus_toglbutton2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hseparator2</td>
<td>hseparator2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vseparator1</td>
<td>vseparator1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>workload_label10</td>
<td>workload_label10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>max_course_label11</td>
<td>max_course_label11</td>
</tr>
<tr>
<td></td>
<td>workload</td>
<td>maco_enter_area2</td>
<td>maco_enter_area2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>min_course_label12</td>
<td>min_course_label12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mico_enter_area3</td>
<td>mico_enter_area3</td>
</tr>
<tr>
<td></td>
<td>workload</td>
<td>max_creadit_label13</td>
<td>max_creadit_label13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>macr_enter_area4</td>
<td>macr_enter_area4</td>
</tr>
<tr>
<td></td>
<td>workload</td>
<td>min_creadit_label14</td>
<td>min_creadit_label14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>micr_enter_area5</td>
<td>micr_enter_area5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hseparator3</td>
<td>hseparator3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>accept_button8</td>
<td>accept_button8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cancel_button9</td>
<td>cancel_button9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hseparator4</td>
<td>hseparator4</td>
</tr>
</tbody>
</table>

Table 5.2: Class Attribute Table
ensure this, the user interface group prepares the Dual View Association Table. The Dual View Association Table contains three columns: The first column lists the components of the user view specified during the dialog design. Each user view component in this column is separated by a semi-colon. The second column lists the corresponding components in the designer view which are specified in the spatial organization, and the third column lists the class to which these components belong to. Since the toplevel shell, and the manager widgets in the Spatial Organization are invisible to the user, the user’s view does not contain these components. Therefore the Dual View Association Table, does not list the toplevel shell and the manager widgets. Table 5.1 shows the Dual View Association Table constructed for the example system CAS. By doing this, the designers can check if for each component in the user’s view there is a corresponding class component in the designer’s view. Therefore we can assure that all the user views of the interface are translated to the system view.

- **Step 2:** In step 2 we verify, that the spatial organization, the class charts and the class descriptions are consistent. This consistency can be achieved by:

1. Ensuring that the class components in the spatial organization became class attributes in the class descriptions, and “Questions” in class charts are mapped to attributes in the class descriptions. To do this, the user interface designers construct the Class Attribute Table which lists the class name in the first column and the “Questions” in the class charts are listed in the second column. The third column lists the class components. And in the last column, the attribute corresponding to the class component is listed. The Class Attribute Table for the example system CAS is shown in Table 5.2. By doing this, the designer verifies that for each Question in the class chart there is a class component which accepts the user input, and that class component became class attribute when the design moved closer to implementation.

2. Ensure that commands in the class chart are mapped to callbacks in the class description. This is ensured by constructing the Command-Callback Table, which has three columns: the first column specifies the class name. The second column lists the commands in the class chart for that class. And the third column specifies, the callback corresponding to the command
<table>
<thead>
<tr>
<th>Class</th>
<th>Command</th>
<th>Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>start advising session</td>
<td>ok_button1_callback</td>
</tr>
<tr>
<td></td>
<td>request exit</td>
<td>exit_button2_callback</td>
</tr>
<tr>
<td>Selector</td>
<td>request advise</td>
<td>advise_button4_callback</td>
</tr>
<tr>
<td></td>
<td>request preferences</td>
<td>prefs_button5_callback</td>
</tr>
<tr>
<td></td>
<td>request constraints</td>
<td>unpref_button6_callback</td>
</tr>
<tr>
<td></td>
<td>request previous window</td>
<td>previous_button3_callback</td>
</tr>
<tr>
<td></td>
<td>request exit</td>
<td>exit_button7_callback</td>
</tr>
<tr>
<td>Messenger</td>
<td>acknowledge error</td>
<td>ok_button14_callback</td>
</tr>
<tr>
<td>Choicer</td>
<td>accept preferences</td>
<td>accept_button8_callback</td>
</tr>
<tr>
<td></td>
<td>cancel preferences</td>
<td>cancel_button9_callback</td>
</tr>
<tr>
<td>Advisor</td>
<td>request preferences</td>
<td>preferences_button11_callback</td>
</tr>
<tr>
<td></td>
<td>request constraints</td>
<td>constraints_button13_callback</td>
</tr>
<tr>
<td></td>
<td>request approval</td>
<td>print_button10_callback</td>
</tr>
<tr>
<td></td>
<td>request exit</td>
<td>exit_button12_callback</td>
</tr>
</tbody>
</table>

Table 5.3: Command-Callback Table

in the second column. The Command Callback Table for CAS is shown in Table 5.3. By doing this the designers ensure that the class is notified of each command/request given by the user.

3. Ensure that the constraints specified in the class chart are mapped to class invariants in the class description. To do this, the user interface designers will construct the Constraint-Invariant Association Table. This table has three columns: The class name is specified in the first column. The constraints specified in the class chart are listed in the second column and the class invariants in the class description are listed in the third column. The Constraint-Invariant Association Table for CAS is shown in Table 5.4. The interface designers thus ensure that the knowledge the class should maintain, which is specified in the class charts is also specified explicitly in the class description.

By ensuring 1, 2, and 3, the designer can convince that the various activities involved in constructing the static structure of the interface are consistent.
<table>
<thead>
<tr>
<th>Class</th>
<th>Constraint</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Selector</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Messenger</td>
<td>caller_class_name</td>
<td>caller != &quot;&quot;</td>
</tr>
<tr>
<td>Choicer</td>
<td>caller_class_name</td>
<td>caller != &quot;&quot;</td>
</tr>
<tr>
<td>Advisor</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 5.4: Constraint-Invariant Association Table

- **Step 3:** Ensure that the set of behavioral specifications cover the behavior for each of the callbacks specified in the class description. Since, the callbacks in class descriptions become user events in the behavioral specification, we check that for each callback in the class description, there is an user event in the behavioral specification. To do this, the Event-Callback Association table is constructed. This table contains three columns: the first column lists the name of the class. The second column lists the callback function (specified in the class description), and the corresponding user event (specified in the behavioral specification) is listed in the third column. This helps the interface designers to verify that the behavioral specifications specify the behavior for each of the callbacks specified in the class description. The Event-Callback Association Table for CAS is shown in Table 5.5.

- **Step 4:** In CUIM, there does not exist any interactions between the interface objects. The interactions of the User object with other IOBs, specified in the Behavior Specifications, are verified using the Event Correspondance Table. The Event Correspondance Table contains two columns: The first column lists the class(EOB) and the output event generated by it. The second column lists the corresponding class(IOB) which receives this event. The input event corresponding to the output event is also specified in this column. Table 5.6 shows the Event Correspondance Table constructed for the example system CAS. By constructing this table, the designers check that, each event sent by the User object is received by an IOB.
<table>
<thead>
<tr>
<th>Class</th>
<th>Callback</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>ok.button1.callback, exit.button2.callback</td>
<td>ok, exit</td>
</tr>
<tr>
<td>Selector</td>
<td>advise.button4.callback, prefs.button5.callback, unpref.button6.callback, previous.button3.callback, exit.button7.callback</td>
<td>advise, prefs, unprefs, previous, exit</td>
</tr>
<tr>
<td>Messenger</td>
<td>ok.button14.callback</td>
<td>ok</td>
</tr>
<tr>
<td>Choicer</td>
<td>accept.button8.callback, cancel.button9.callback</td>
<td>accept, cancel</td>
</tr>
<tr>
<td>Advisor</td>
<td>preferences.button11.callback, constraints.button13.callback, print.button10.callback, exit.button12.callback</td>
<td>prefs, unprefs, accept, exit</td>
</tr>
</tbody>
</table>

Table 5.5: Event-Callback Table

5.6 Ensuring Consistency between IAD & IDD

1. The Dual View Association table constructed while reviewing the design, shows the system output in the User View column. Since the textual description of the nodes in the state transition diagrams also specify the system output, we need to ensure that the task of displaying the system output which is described by a node in the IAD document is handled by a class in the IDD document. To do this, the interface designer constructs the Node-Class Association table. This table contains two columns. The first column lists the different nodes in the state transition diagrams. The corresponding class name is listed in the second column. The Node-Class Association Table for CAS is shown in table 5.7. By doing this, the designers ensure that for every node stated in the state transition diagrams of the IAD document there exists an interface class in the IDD document which handles the required system output.

2. Since the decisions about how much freedom must be given to the user to switch between tasks are made during the analysis stage, it is important for the
<table>
<thead>
<tr>
<th>From</th>
<th>Output Event</th>
<th>Input Event</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>ok(id.val)!@U</td>
<td>ok(id.val)?</td>
<td>Initiator</td>
</tr>
<tr>
<td>User</td>
<td>previous!@V</td>
<td>previous?</td>
<td>Selector</td>
</tr>
<tr>
<td>User</td>
<td>prefs!@V</td>
<td>prefs?</td>
<td>Selector</td>
</tr>
<tr>
<td>User</td>
<td>unprefs!@V</td>
<td>unprefs?</td>
<td>Selector</td>
</tr>
<tr>
<td>User</td>
<td>advise!@V</td>
<td>advise?</td>
<td>Selector</td>
</tr>
<tr>
<td>User</td>
<td>ok!@Z</td>
<td>ok?</td>
<td>Messenger</td>
</tr>
<tr>
<td>User</td>
<td>course_val!@B</td>
<td>course_val?</td>
<td>Choicer</td>
</tr>
<tr>
<td>User</td>
<td>time!@B</td>
<td>t_val?</td>
<td>Choicer</td>
</tr>
<tr>
<td>User</td>
<td>macol!, micol!,</td>
<td>maco.val?,</td>
<td>Choicer</td>
</tr>
<tr>
<td></td>
<td>macr!, micr!@B</td>
<td>mico.val?,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>macr_val?,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>micer_val?</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>cmp1!, cmp2!@B</td>
<td>cmp1?, cmp?</td>
<td>Choicer</td>
</tr>
<tr>
<td>User</td>
<td>accept!@B</td>
<td>accept?</td>
<td>Choicer</td>
</tr>
<tr>
<td>User</td>
<td>cancel!@B</td>
<td>cancel?</td>
<td>Choicer</td>
</tr>
<tr>
<td>User</td>
<td>print!@C</td>
<td>print?</td>
<td>Advisor</td>
</tr>
<tr>
<td>User</td>
<td>prefs!@C</td>
<td>prefs?</td>
<td>Advisor</td>
</tr>
<tr>
<td>User</td>
<td>unprefs!@C</td>
<td>unprefs?</td>
<td>Advisor</td>
</tr>
<tr>
<td>User</td>
<td>exit!@U, @V, @C</td>
<td>exit?</td>
<td>Initiator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exit?</td>
<td>Selector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exit?</td>
<td>Advisor</td>
</tr>
</tbody>
</table>

Table 5.6: Event Correspondance Table
<table>
<thead>
<tr>
<th>Node</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodb</td>
<td>Messenger</td>
</tr>
<tr>
<td>inv_id</td>
<td></td>
</tr>
<tr>
<td>msg1</td>
<td></td>
</tr>
<tr>
<td>msg2</td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>Initiator</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>Selector</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>cas</td>
</tr>
<tr>
<td>preferences</td>
<td>Choicer</td>
</tr>
<tr>
<td>constraints</td>
<td></td>
</tr>
<tr>
<td>start</td>
<td>Advisor</td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: Node-Class Association Table

interface designers to ensure that the ordering of tasks in the IDD document is consistent with the ordering of tasks described in the IAD document. This can be achieved by constructing the Action Response table. The Action Response Table contains three columns: the first column lists the actions performed by the user. The second column which specifies the node in the state transition diagrams contains two sub-columns: Current, Next. The sub-column "Current" lists the node which shows the current state of the system. The sub-column "Next" lists the next possible node(s) that can be reached in the state transition diagrams. The last column in the Action Response Table specifies the interface class in the IDD document which corresponds to the node(s) in the second column. The sub-column "Current" in this column specifies the class that is currently active, and the class name(s) specified in the sub-column "Next" lists the possible class that can be invoked next.

From the Node-Class Association Table (table 5.7) constructed earlier, one can understand which nodes correspond to which interface classes. Therefore, by comparing the columns "Node" and "Class" in table 5.8, the designers check that the system responses listed in the third column are consistent with the
<table>
<thead>
<tr>
<th>User Action</th>
<th>Node Current</th>
<th>Node Next</th>
<th>Class Current</th>
<th>Class Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>key</td>
<td>main</td>
<td>Initiator</td>
<td>Selector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inv_id</td>
<td></td>
<td>Messenger</td>
</tr>
<tr>
<td>Previous</td>
<td>main</td>
<td>key</td>
<td>Selector</td>
<td>Initiator</td>
</tr>
<tr>
<td>Preferred Choices</td>
<td>main</td>
<td>preferences</td>
<td>Selector</td>
<td>Choicer</td>
</tr>
<tr>
<td>Unprefered Choices</td>
<td>main</td>
<td>constraints</td>
<td>Selector</td>
<td>Choicer</td>
</tr>
<tr>
<td>Advise</td>
<td>main</td>
<td>start</td>
<td>Selector</td>
<td>Advisor</td>
</tr>
<tr>
<td>OK</td>
<td>inv_id</td>
<td>key</td>
<td>Messenger</td>
<td>Initiator</td>
</tr>
<tr>
<td></td>
<td>msg1</td>
<td>preferences</td>
<td>Messenger</td>
<td>Choicer</td>
</tr>
<tr>
<td></td>
<td>msg2</td>
<td>constraints</td>
<td>Messenger</td>
<td>Advisor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>start</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefered Choices</td>
<td>start</td>
<td>preferences</td>
<td>Advisor</td>
<td>Choicer</td>
</tr>
<tr>
<td>Unprefered Choices</td>
<td>start</td>
<td>constraints</td>
<td>Advisor</td>
<td>Choicer</td>
</tr>
<tr>
<td>Accept</td>
<td>start</td>
<td>msg2</td>
<td>Advisor</td>
<td>Messenger</td>
</tr>
<tr>
<td>Exit</td>
<td>key</td>
<td>-</td>
<td>Initiator</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>main</td>
<td>-</td>
<td>Selector</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>start</td>
<td>-</td>
<td>Advisor</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.8: Action Response Table
system responses listed in the second column.

5.7 Ensuring Consistency between IDD & CDD

All those activities (dialog design, constructing the static structure, behavioral specification, and so on) that are carried out during the user interface design phase comprise the Interface Design Document (IDD). The Computational Design Document (CDD) comprises all those activities carried out during the design phase of the computational process. Once the IDD document and the CDD document are produced, the design walk-through is conducted. A design walk-through is an informal review of the IDD and CDD documents, as a cooperative and organized activity by several participants (user interface engineers, and the computational engineers). Software engineers from both groups (interface group and the computational group) meet to review the output of the design phase of both the interface process and the computational process. This meeting focuses on “discovering the errors and inconsistencies”, but not fixing them.

Key people (say, the group leaders) in either group walk through the IDD and CDD documents to present and explain the rationale of their work. The software engineers check that, for every output event sent by an IOB in the IDD, there exists a COB in the CDD document, which accepts that event. Similarly, for every output event sent by a COB in the CDD document, there should exist an IOB in the IDD document, which accepts that event. This process of verification becomes much easier if the computational group also uses the TROM model to specify the behavior of the various classes. During the design walk through, the user interface engineer takes notes on the changes that are to be made to the IOBs/COBs in the IDD document. This applies to the computational engineer too. Therefore, this walk-through serves the interface group to ensure that the CDD document does not miss any of the tasks that are modeled in the IDD document and vice-versa. An overview of all the tables created during the User Interface Design phase in CUIM, is given in Table 5.9.

Since the various activities in CUIM are quite independent of the activities involved
<table>
<thead>
<tr>
<th>Table</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Dual View Association Table</td>
<td>Ensures that the components in the <em>spatial organization</em> corresponds to the interface components specified during the <em>dialog design</em></td>
</tr>
<tr>
<td>5.2</td>
<td>Class Attribute Table</td>
<td>Verifies that for each Question in the <em>class chart</em> there is a class component which accepts the user input, and that class component is mapped to attributes in the <em>class descriptions</em></td>
</tr>
<tr>
<td>5.3</td>
<td>Command-Callback Table</td>
<td>Ensures that the commands in the <em>class chart</em> are mapped to callbacks in the <em>class description</em></td>
</tr>
<tr>
<td>5.4</td>
<td>Constraint-Invariant Association Table</td>
<td>Ensures that the constraints in the <em>class chart</em> are mapped to invariants in the <em>class description</em></td>
</tr>
<tr>
<td>5.5</td>
<td>Event-Callback Table</td>
<td>Ensures that the set of <em>behavioral specifications</em> cover the behavior for each of the callbacks specified in the <em>class descriptions</em></td>
</tr>
<tr>
<td>5.6</td>
<td>Event Correspondance Table</td>
<td>Ensures that the output event sent by the User object is received as an input event, by an IOB</td>
</tr>
<tr>
<td>5.7</td>
<td>Node-Class Association Table</td>
<td>Ensures that the task of displaying the system output which is described by a <em>node</em> in the IAD document is handled by a <em>class</em> in the IDD document</td>
</tr>
<tr>
<td>5.8</td>
<td>Action Response Table</td>
<td>Ensures that the ordering of tasks in the IDD document is consistent with the ordering of tasks described in the IAD document</td>
</tr>
</tbody>
</table>

Table 5.9: Overview Table
in the computational process, this thesis does not concentrate on the computational design phase. But, following the advanced evolutionary prototype model, ensuring consistency between the IDD document and the CDD document is necessary. This is achieved by using the CDD document for CAS, which is prepared by Kim [Duon95]. We conducted design walk-through to review the IDD and CDD documents. By reviewing the IDD and CDD documents, we ensured that the CDD document does not miss any of the tasks modeled in the IDD document and vice-versa.

Once again, it is important to note that the length of the interface design stage may not be the same as the length of the computational design stage. The length may vary from application to application. There will be situations where the IDD document is available, and the CDD document is not available yet. Then the interface group postpones the design walk-through until the CDD document is available, and enters the user interface implementation and testing stage. The implementation and testing phase in CUIM is discussed in Chapter 6.
Chapter 6

Implementation and Conclusions

Following the design phase in the advanced evolutionary prototype model are the implementation and testing phases. Section 6.1 describes how the various design activities assist the implementer in constructing the "Interface Subsystem". The testing phase in CUIM is discussed in Section 6.2. Section 6.3 introduces a proposed Hyper-media Design Tool to assist designers in following the CUIM methodology. The conclusions of this thesis are presented in Section 6.4. Finally, Section 6.5 points out the future work that can be carried out.

6.1 Implementing the User Interface

As pointed out in Chapter 5, the user interface possesses the look and feel characteristics. Since the look of the interface is already achieved during the design, the implementation phase achieves the feel characteristic by implementing the user interface, and letting the user try it. The output of the design phase, which is the IDD document serves as the input to this phase. The user interface developers build the look of the user interface by using the class charts, the spatial organization constructed during the design phase and the dialog specifications given in the IAD document. While implementing the user interface, it is possible that the placement(row number, column number, and so on) of the various components of the user interface might slightly

139
vary from that given in the dialog specifications.

Once the look of the interface is implemented, the user interface developers make use of the class descriptions to specify the callback functions required for each class. The system interactions specified during the design phase are then used in writing the code for the callbacks. The behavior specified in the system interactions is rewritten in C++[Berr92]. The port identifiers in the behavioral specification, correspond to object instances during the implementation. An event in the behavioral specification, corresponds to either a method of a class or a return value from a method, during the implementation. For example, considering the Initiator TROM (Chapter 5), the event “invoke_1” corresponds to the method “Invoke_1” and the event “valid” corresponds to the return value from the method of a computational object. It is up to the implementer to make these decisions of whether an event in TROM should correspond to a method or to a return value from a method, during the implementation. Since the design phase in CUI/M suggests for simulating the functionality of the COBs, in order to connect the different interface objects implemented, the functionality of the COBs is simulated during the implementation phase. Thus, the output of the implementation phase is the “Interface Subsystem”, which comprises the executable file, and the code written by the user interface implementers.

The existing GUI tools such as UIM/X[Visu93], and OSF/MOTIF[Bra92] can be used for implementing the user interface on the X Window System[Sche92]. The various user interface classes of CAS, designed during the design phase are implemented in MOTIF, using the UIM/X toolkit. Figures 6.1 through 6.6 show the X Window dumps of CAS user interfaces implemented. UIM/X assisted us to interactively choose the MOTIF widgets (such as, pushbuttons, scrollbars, and so on) to construct, modify, test, and generate code for the look of the user interface. This resulted in developing the user interface to CAS, in shorter amount of time. The time taken to develop the user interface to CAS was 1 week.

The next step after implementing the user interface is to verify, that the Interface Subsystem corresponds to the decisions made during the design. Code walk-throughs

---

1This does not include the time taken to learn UIM/X; Knowledge of OSF/MOTIF is essential to learn UIM/X
Figure 6.1: The initiator Interface

Figure 6.2: The messenger Interface
Figure 6.3: The selector Interface

Figure 6.4: The preferences Interface
Figure 6.5: The constraints Interface

Figure 6.6: The advisor Interface
are performed to verify the program against the design. Since the executable file produces the user's view of the interface, the different interfaces produced by the executable file are compared with the user view (Design Stage) to see if they resemble the interfaces produced during the dialog design. Comparing the CAS user interfaces implemented, with those designed during the dialog design, we see that the interfaces implemented resemble the interfaces in the dialog design. Once this is verified, the callback functions for each class specified during the implementation phase are compared with the callback functions listed in the class descriptions. This can be achieved by constructing a callback table. The callback table contains three columns: with the first column listing the class name, the second column listing the callbacks from the class descriptions, and the third column listing the corresponding callbacks specified during the implementation. The callback table for CAS is shown in Table 6.1. From this table we see that for each callback specified during the design there is a corresponding callback realized during the implementation and vice-versa. The interface designers and the interface implementers walk through the code to verify that the behavior specified during the design phase has been implemented during the implementation phase, for each of these callbacks. When this verification is satisfactorily done, we can say that the implementation phase in CUID, models the decisions made during the design.

6.2 Testing the User Interface

The user interface development process is completed, when the outcome of the testing phase is satisfactory. Generally the testing phase concentrates on validating the user interface developed against the customer requirements. Testing of the user interface is categorized into two types: 1. Customer Testing, 2. User Testing.

- Customer Testing:

Customer testing is the one that is carried out by the interface developers and the customer. A set of tasks representing all the User Goals listed during the analysis, are prepared prior to conducting the test. During the demo (customer testing phase), the system is tested to see if it satisfies all the user goals when
<table>
<thead>
<tr>
<th>Class</th>
<th>Design Phase</th>
<th>Implementation Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>ok.button1.callback exit.button2.callback</td>
<td>activateCB.pushButton1 activateCB.pushButton2</td>
</tr>
<tr>
<td>Messenger</td>
<td>ok.button14.callback</td>
<td>activateCB.pushButton7</td>
</tr>
<tr>
<td>Selector</td>
<td>advise.button4.callback prefs.button5.callback unprefs.button6.callback prev.button3.callback exit.button7.callback</td>
<td>activateCB.pushButton3 activateCB.pushButton4 activateCB.pushButton5 activateCB.pushButton6 activateCB.pushButton8</td>
</tr>
<tr>
<td>Chioicer</td>
<td>accept.button8.callback cancel.button9.callback</td>
<td>activateCB.pushButton9 activateCB.pushButton10</td>
</tr>
<tr>
<td>Advisor</td>
<td>approve.button10.callback preferences.button11.callback exit.button12.callback constraints.button13.callback</td>
<td>activateCB.pushButton15 activateCB.pushButton16 activateCB.pushButton18 activateCB.pushButton17</td>
</tr>
</tbody>
</table>

Table 6.1: Callback Verification Table

the tasks listed are performed. Since the User Goals listed during the analysis represent the different goals the different users of the system will have, the tasks prepared during this phase, represent the different goals the different users of the system will have. Therefore if all the goals are achievable, then we say that the interface developed, satisfies the goals of users with different task experience.

**CUIM** suggests that customer testing should be performed prior to user testing. It is a well known fact [Ghez91] that in most of the cases, the customer himself does not know the complete requirements of the application that is to be developed. New requirements evolve as the system (user interface) is being demonstrated. Since the user testing is costly, **CUIM** suggests that if any requirements evolved during the customer testing phase, the development process should be iterated prior to user testing. In successive iterations, the user interface analysis, and design phases are iterated to match the new (modified) requirements. Customer testing is carried out once again, after implementing the user interface which matches the new requirements.
Considering our example system CAS, the following are the list of tasks that we have selected to execute during the customer testing phase. This task selection will be based on the task analysis performed at the analysis stage.

- **Task Set 1**
  
  1.1 Without specifying your preferences/constraints, *request* the system to give advice.

- **Task Set 2**
  
  2.1 Specify your preferences.
  
  2.2 Specify your constraints.
  
  2.3 Request the system to give advice.

- **Task Set 3**
  
  3.1 Specify your preferences.
  
  3.2 Specify your constraints.
  
  3.3 Request the system to give advice.
  
  3.4 Change your preferences list.
  
  3.5 Request the system to give advice.

A customer is the individual who has commissioned the development of the software. Customer may or may not be the end user. The Informal Requirements Document (IRD) has been developed in close consultation with the customer. He is involved in Customer Testing. At the end of the customer testing phase, we found that: all the specified goals are achievable. However, one omission was discovered and corrected. This is explained below:

- The requirement: Allow the user to delete the course, time which is specified before, in the preferred/unpreferred choices.

We went through one iteration of the user interface development process, to incorporate the requirement evolved. This iteration resulted in adding a “delete” button which allows the user to delete the course, and time specified before.

- **User Testing:**
User testing is carried out by randomly selected people who comprise the user population for the application being developed. The true test of an interface is to conduct *user testing*, which evaluates how satisfactory is the developed user interface. Issues such as error rate, user response time (task response time), ease of use, and ease of learning of the user interface are considered in evaluating the user interface.

The amount of time spent in the completion of a task depends on the *system response time* (SRT) as well as the *user response time* (URT). Therefore, we can define the *task performance time* (TPT) as the sum of SRT and URT. An unacceptably higher TPT would make us conclude that the system is *inefficient*. Since, SRT is taken care by the computational group, the user interface group strives to achieve lower URT.

In *CUIM*, user testing is carried out by allowing users to perform a pre-selected list of tasks on the interface developed. Prior training is also given to the users if the user profile constructed during the analysis states that training is necessary. The user tries to complete these tasks while an evaluation expert is observing. The observer is passive with respect to the task being attempted, but records the errors made by the user in the completion of the tasks. A questionnaire is also prepared to get user feedback about the interface developed.

After conducting the test, the results of the test are examined. If the outcome of the test resulted in unacceptable user response to any of the test parameters, we conclude that the test results are unsatisfactory. At this point, the user interface group examines the following:

1. If the test results and the conclusions derived from them are reliable. If not, should the test be repeated.

2. If the inadequate performance is due to shortcomings in the user interface, what modifications are required to the user interface.

Considering our example system CAS, we conducted user testing on the interface developed. Ten undergraduate students from the department of *Computer Science* performed the testing of the user interface. During the testing phase:
1. The students were first asked to get familiar with the interface system by performing different actions such as entering the id, listing the preferences etc..

2. After they got familiar with the system, the students were asked to perform a given set of tasks prepared during the Customer Testing phase.

3. Finally, a questionnaire was given to the students to get their feedback about the user interface.

Test Results:

1. User Profile:

   From the test conducted it was observed that the users who participated in testing the user interface have the following characteristics:

   - During the test:

     (a) The errors committed by the students were noted down by the observer. Table 6.2 shows the errors committed by the users while using the system for the first time. From this table, we see that student #2 is not confident about how to give input to a mouse driven application.

     (b) The type of errors made by other users (Table 6.2), tells us that the errors made are not due to lack of computer knowledge. Therefore, we say that most of the students are familiar with GUI based interactions.

     (c) Some of the students suggested that more beautiful user interfaces can be created by using Microsoft Windows. At this point, it is important to see the difference between (b) and (c). Point (b) talks about students who have knowledge of using windows and point (c) talks about students who have knowledge of creating windows.

   Therefore from (a), (b), and (c), we conclude that the computer literacy of the users participated in the testing, ranged from low to high.

   - (a) There are students who made errors (Table 6.2) while specifying the preferred time.

   (b) There are students who did not make any errors (Table 6.2).
<table>
<thead>
<tr>
<th>Student</th>
<th>Error committed while using the system for the first time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>specifying the preferred time</td>
</tr>
<tr>
<td>2</td>
<td>is not confident about how to give input to a mouse driven application; specifying the preferred time</td>
</tr>
<tr>
<td>3</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>specifying the preferred time</td>
</tr>
<tr>
<td>5</td>
<td>specifying the preferred time</td>
</tr>
<tr>
<td>6</td>
<td>specifying the preferred time</td>
</tr>
<tr>
<td>7</td>
<td>none</td>
</tr>
<tr>
<td>8</td>
<td>none</td>
</tr>
<tr>
<td>9</td>
<td>none</td>
</tr>
<tr>
<td>10</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 6.2: Error List

Since the user population included people with low computer literacy, we can say that it included people with low syntactic knowledge. From the errors listed in (a), and (b), we observe that the syntactic knowledge for users participated in the testing, ranged from moderate to high.

- Table 6.3 shows that there are variations in the time taken by the students to perform the tasks. One of the possible reasons for this could be the typing skill of the users. Therefore, we can say that the test is performed by people with different typing skills.

- During the customer testing phase, it was found that the user interface satisfied the goals of users with different task experience.

- The characteristics such as Frequency of Use, Primary Training, System Use, and Task Importance are independent of the user population.

From the characteristics mentioned above, we conclude that the characteristics of the user population participated in the test, matched the user profile constructed during the analysis stage.

2. Efficient:

Efficiency of an user interface can be measured in several ways ([Bass90], [Barf93]). In this evaluation experiment, we chose to measure it by means of the time taken
by a user to complete a set of selected tasks. The user interface designer is considered as the expert user. The amount of time spent by an expert user in performing the tasks given to the users is as follows:

(a) Task 1: 30 sec
(b) Task 2: 2 min
(c) Task 3: 3 min

Task #3 when carried out in the “existing manually oriented advising system” (professor-student interaction) would take an average of 17.5 minutes, spread over the interval of 15-20 minutes. We set a goal of getting advice from CAS should be at least two times faster. This gave us an expectation of 8.75 minutes or 9 minutes (when rounded) for completing the task.

From Table 6.3, we see that the average time taken to perform the task is much less than the acceptable average time. Therefore, we consider that the average task performance time is quite low, or the interface is efficient.

3. Easy to Learn:

It was found that the average number of errors performed, when users performed the selected tasks for the first time was 0.6. Also, the users did not repeat the errors which they committed once. Therefore, we conclude that the system is easy to learn.

4. Easy to Use:

(a) From the questionnaire, it was found that 80% of the users marked that the interface is Easy to use and 20% of the users marked that the interface is Very Easy to use. This is an indication of the subjective evaluation.

(b) By noting down the number of errors made by the user during the testing phase, we found that the average number of errors committed while doing the tasks given to them, was zero.

(c) Since, the system response time(SRT) is negligible, we can say that the task performance time(TPT) is practically equal to the user response time(URT). The low TPT resulted in low URT.
<table>
<thead>
<tr>
<th>Task1</th>
<th>Task2</th>
<th>Task3</th>
</tr>
</thead>
<tbody>
<tr>
<td>55 sec</td>
<td>3.40 min</td>
<td>4.59 min</td>
</tr>
<tr>
<td>43 sec</td>
<td>3.13 min</td>
<td>4.12 min</td>
</tr>
<tr>
<td>30 sec</td>
<td>2.09 min</td>
<td>3.10 min</td>
</tr>
<tr>
<td>1.42 min</td>
<td>3.45 min</td>
<td>4.46 min</td>
</tr>
<tr>
<td>43 sec</td>
<td>2.54 min</td>
<td>3.43 min</td>
</tr>
<tr>
<td>33 sec</td>
<td>2.16 min</td>
<td>3.27 min</td>
</tr>
<tr>
<td>31 sec</td>
<td>2.06 min</td>
<td>3.08 min</td>
</tr>
<tr>
<td>32 sec</td>
<td>2.43 min</td>
<td>3.30 min</td>
</tr>
<tr>
<td>32 sec</td>
<td>2.12 min</td>
<td>3.07 min</td>
</tr>
<tr>
<td>30 sec</td>
<td>2.15 min</td>
<td>3.01 min</td>
</tr>
<tr>
<td>Ave.: 43 sec</td>
<td>2.34 min</td>
<td>3.34 min</td>
</tr>
</tbody>
</table>

Table 6.3: Test Results

From (a), (b) and (c), we conclude that the user interface is easy to use.

5. In the questionnaire given, users felt that the interface is easy to use, and commented that it would be better if the OK button in the start interface is replaced by a return key.

Since the test results obtained above, are satisfactory and the changes in user requirements evolved from user testing are minimal, we concluded that another iteration of the development process is not necessary. The process of integration will be carried out in future, when results of the computational process are available.
6.3 The Hyper-media Design Tool

In Chapters 4, 5, and 6, we have shown how to apply the Concurrent User Interface Methodology proposed in this thesis. But we had no good tools to support the various activities in CUIM. We propose the Hyper-media Design Tool (HDT) in this Section. As the name implies, the HDT tool is based on hyper-media concepts [Niel90]. The HDT tool will assist the user interface designers during the various design activities in CUIM by ensuring that the steps in CUIM are faithfully followed. The HDT will support the different specification notations used in CUIM. The tool is abstracted in Figure 6.7, contains different nodes (boxes) corresponding to different design activities in CUIM. The information flow between the different design activities (nodes) is represented as a link (an unidirectional arrow). The HDT tool contains two types of links: 1. Hard Link and 2. Soft Link. A Hard Link is denoted by a directed solid line, and, a Soft Link is denoted by a directed broken line. A Hard or Soft Link emanating from node A to node B denotes that node A serves as input to node B. Multiple inputs to a box, denotes that all those inputs are required to perform the activity corresponding to the incident node.

We define the term component to denote the contents of a node. In HDT, Hyper-links are established between components. No hyper-link can be established between components of the two nodes which are connected by a Soft Link. A hyper-link is established by the user interface designer who is designing an user interface following CUIM methodology. The HDT tool is useful for establishing such hyper-links. Specifying the node components and the hyper-links between the node components is left to the user interface designer. However, the hard link from one node (box in Figure 6.7) to another is automatically established by the HDT. Traversing is possible either through the automatically established links or through hyper-links.

By providing access to relevant information via hyper-links, HDT enables the designer to scan the relevant information to any depth that is needed. This should help in the change and maintenance of the design.

The boxes in the node link diagram (Figure 6.7) are explained below:
Figure 6.7: The Hyper-media Node-Link Diagram
- **The DialogSpecifier:** The Dialog Specifier requires a graph editor, which allows the designer to draw the extended state transition diagrams, while specifying the dialog between the user and the interface. The Dialog Specifier serves as input to the Interaction Specifier and the Dialog Styler.

- **The Interactions Specifier:** The Interactions Specifier requires a graph editor, which allows the designer to draw the interaction diagrams, while specifying the interactions between the IOBs and the COBs. The Interactions Specifier serves as input to the Class Chart Constructor.

- **The Dialog Styler:** The Dialog Styler requires a graph editor, which allows the designer to draw the various dialogs identified during the dialog design. This node serves as input to the Class Chart Constructor, and the Spatial Organizer.

- **The Class Chart Constructor:** The Class Chart Constructor provides a form-like user interface, and asks the designer to enter the information pertaining to a class chart. Then the class charts are constructed automatically, by the Class Chart Constructor. This node serves as input to the Spatial Organizer, and the Class Descriptor.

- **The Spatial Organizer:** The Spatial Organizer requires a graph editor, which allows the designer to draw the various classes, class components, relationship arcs, and the text required for labeling, while constructing the spatial organization of the user interface. This node serves as input to the Class Descriptor.

- **The Class Descriptor:** The Class Descriptor requires a graph editor, which allows the designer to describe the components of the header and the body in a class description. This node serves as input to the Behavior Specifier.

- **The Behavior Specifier:** The Behavior Specifier requires a text editor, which allows the designer to describe the textual description of the behavior of the class. Taking the textual description, the equivalent finite state machine is generated by the Behavior Specifier. This node serves as input to the Configuration Constructor.

- **The Configuration Constructor:** The Configuration Constructor requires a graph editor, which allows the designer to specify the configuration of the user interface subsystem.
With respect to the editor supported by the HDT, we recognize the following design choices:

**Design Choice 1**: A single editor, which facilitates the construction of:

- the extended state transition diagrams
- the interaction diagrams
- the dialog styles
- the class charts
- the spatial layout of the user interface
- the class descriptions and
- the behavior specifications

**Design Choice 2**: Each node (box in Figure 6.7) in the HDT contains an associated editor. In this case, each editor could be a specialized syntax directed editor. We recommend the design choice 2 because,

- A local editor to each node, provides the notations specific to a design activity, by omitting the rest.

- More meaningful messages can be given to the user, when the rules of CUIM are violated. For example, when constructing the interaction diagrams, the designer will be informed if an interaction happens between any two IOBs.

### 6.4 Conclusions

Different user interface models ([Fole82], [Kier88], [Norm84]) proposed in the literature concentrate on developing the user interface, and do not provide any indication of how the user interface developed will interact with the computational part of the software. Also, these models do not provide a systematic procedure for user interface
development. The model proposed by Sutcliffe[1991] is an exception, in the sense that it deals with both the user interface part and the computational part. But there is no clear cut division between them. The Advanced Evolutionary Prototyping Model proposed in this thesis, views the software development to consist of two loosely coupled, possibly concurrent processes. This model allows user interface developers and software developers to work concurrently on the same project, with interactions between them occurring at well defined points in the development process. We believe that the concurrency supported by the Advanced Evolutionary Prototyping Model, would result in shorter software development time.

The Advanced Evolutionary Prototyping Model determines the sequence and interactions between the various phases in the software development, and establishes the transition criteria for progressing from one stage to the next. To put this model into practice, the Concurrent User Interface Methodology (CUIM) has been developed. CUIM focuses on how to navigate through each phase of the development and how to represent the “phase-products”, in the Advanced Evolutionary Prototyping Model.

Based on the Advanced Evolutionary Prototyping Model, we conducted a case study which followed the CUIM methodology. As a part of the case study, the user interface to the Course Advising System (CAS) has been developed, and an evaluation experiment has been conducted to test the developed user interface. This thesis has shown the feasibility of the Advanced Evolutionary Prototyping Model in general, and the CUIM methodology in particular. The experimental evaluation made us believe that the CUIM methodology can lead to better user interfaces; better in the sense of promoting ease of use, and ease of learning of the user interface.

Although the concurrency between the user interface process and the computational process in the Advanced Evolutionary Prototyping Model is logically evident, no rigorous evaluation is done in this thesis. Through this concurrency, that one can reduce the product development time is a subjective claim and no objective testing is done in this thesis.
No methodology can be widely practiced without a good set of tools. The Hypermedia Design Tool (HDT) proposed in this thesis, will assist the user interface designers during the various design activities in CUIM. Use of the HDT tool will ensure that the various steps in CUIM are followed. The design, and development of the HDT tool are left as part of the future work.

6.5 Future Work

The work presented in this thesis, serves as a starting point for developing a systematic procedure for the design and development of user interfaces. Some of the possible directions for further research related to this thesis are as follows:

- The Hypermedia Design Tool described in this thesis needs to be refined, implemented and tested.

- In a typical multi-media application[Phil91], there can be multiple sources of input. For example, consider a voice command as one input, and mouse click or keystrokes as another input. These two modes can be concurrently active. The extended state transition diagrams used for dialog specification, does not support this. Extending the dialog specifications to capture the multi-modal and simultaneous input needs further research.

- Through a case study, the Timed Reactive Object Model (TROM) developed in [Achu94] is demonstrated as a potential tool for user interface design in CUIM. The Behavioral Descriptions developed using TROM are manually verified (that every output event sent by the user object, is received by an IOB) with the help of tables. This verification can be automated. In this context, we note that work is in progress as a continuation of [Achu94], to develop a set of tools for using TROM in specifications.

- The Course Advising System (CAS) is the only example which was used in this thesis for examining the CUIM methodology. Extensive testing of the adequacy of this methodology for all types of user interfaces can be undertaken in future.
Bibliography


