Abstract

Successful development of interactive systems depends on the quality of the requirements engineering process. The past two decades have seen a great deal of human-computer interaction (HCI) research effort in the search for appropriate representations to support user interface and usability requirements. Informal representations, such as statements in natural language, are often supportive in overcoming the difficulty of communication between people with differing backgrounds. However, natural languages suffer from ambiguity and imprecision. Formal representations, on the other hand, provide a solution to the ambiguity problem and facilitate formal proof and analysis of properties of requirements. Such specifications form the basis of formal software development and verification of implementation correctness, however, such specifications are often difficult to understand and develop for newcomers to this area. Semi-formal representations such as scenarios and use cases are promising vehicles for eliciting, specifying and validating user interface requirements.

This thesis proposes a Scenario and Use Case-based for Requirements Engineering (SUCRE) framework. The framework enriched the Use Case Maps (UCM) with new visual notation for representing interface at different abstract levels. We investigate how the enriched UCM for User interface (UCM-UI) models with the complicity of formal software requirements engineering methods can lead to SUCRE the novel comprehensive framework. SUCRE is used to represent and validate user interface requirements while improving and mediating the communication between different parties involved. SUCRE builds operators to validate the UCM-UI model using deign heuristics for constructing a formal specification. Moreover, part of SUCRE framework is a metrics suite to predict usability from scenarios and use cases. This metrics suite includes simple structural measures as well as content-sensitive and task-sensitive metrics. Finally, SUCRE bridges the gap between requirements and design. It is shown that UCM-UI models can form the basis for further developments. As such, they bridge the gap between requirements and detailed design models offered by UML, LOTOS specifications, and XML.

Table of Content

List of Figures

List of Tables

Chapter 1

Introduction

The last 30 years have seen growing interest and efforts towards ameliorating the critical process of engineering higher-quality requirements. Generally software systems requirements engineering (RE) is the process of discovering the purpose for which software systems are implemented by identifying the system users and their needs and documenting these in a form that is amenable to analysis, communication, and subsequent implementation. When applying RE to interactive software systems, there are a number of inherent difficulties in this process. Interactive software systems consist of application functionality and the user interface. Designing user interfaces is challenging due to the necessity of matching concepts understood by users with concepts implemented by engineers. Thus, designing user interfaces often relies too much on trial-and-error techniques. The methods and techniques for user interface design need to be improved in order to systematically develop better systems. Capturing, documenting and validating the user interface requirements at early stages improves the quality of the initial design and reduces the number of iterations needed to get the details right.

Since the late 1980s, researchers in human–computer interaction (HCI) have used scenarios as representation of user interface requirements to improve communication between engineers and users. Engineers look at scenarios as an effective means to discover user needs, to better embed the use of systems in work processes, and to systematically explore system behavior – under both normal and exceptional situations. In the past few years, scenarios have gained enormous popularity through Ivar Jacobsený´s Use Case approach which is now feeding into the efforts to establish a unified modeling language (UML) for systems engineering based on the object-oriented approach . Scenario and use case-based approaches have been

proved to be an effective mechanism to capture requirements and make them available for review, avoiding any implementation bias in the requirement. They offer a significant and unique leverage on some of the most characteristic and vexing challenges of requirements engineering (Pomerol, 1998; Sutcliffe, 2002; Carroll, 2003). In spite of that, the practical application of the scenario and use case-based approaches to real-life interactive systems turned out to be confusing for the following main reasons:

1- the scenario and use case approaches lack proper definitions of the technique's constructs to suit user interfaces,

2- the scenario and use case approaches lack a well defined process for practically applying the technique, and

3- the relation between requirements and scenarios and use cases is not clear enough.

Although there is a considerable number of scenario and use case-based approaches published in the literature as discussed in Chapter 2, either they are not readily suitable for the requirements elicitation and specification of user interfaces. Or these approaches are not well integrated with techniques for modeling the functional application part of the interactive systems. As Collins in (1995) states, "published methods for user interface design largely ignore the fact that it is part of product development".

Considering the inadequacy of scenario and use case-based approaches and their poor integration with techniques for modeling the underlying functional application of the interactive systems, the main aim of this dissertation is to remove this dichotomy using an appropriate approach for modeling the entire interactive system. This thesis describes a scenario and use case-based requirement engineering framework for user interfaces. This framework covers the important aspects from scenario analysis to prototype evaluation. The main idea behind the framework is that a firm understanding of the users' needs is the proper basis for interactive systems development. With a systematic process of building user interfaces to support users' tasks, a higher level of usable and useful systems can be achieved by engineers. Scenarios and use cases are not the only existing approach for handling user interface requirements. However, most approaches cover only specific aspects or activities in

the design process. The framework presented in this thesis builds on existing techniques, but it also adds several original insights and techniques.

1.1 Research Objectives

The ultimate goal of this research is to develop techniques for capturing user interface requirements and to build a framework for documenting and validating the requirements of interactive systems where the user interface designers, the software engineers, and the customers can collaborate to produce a well-integrated product. The outcome of the requirement phase is the foundation for the whole subsequent development. In particular, we will focus on four areas:

– *Developing techniques that are used for gathering the necessary information to build a consistent and complete use case mode*l. Although scenario and use cases are an old approaches to describe requirements, it has not been developed far enough to be of sufficient value to interface requirement. Techniques such as interviews, focus group, essential use cases, use storyboards will be explored in order to highlight their pros and cons and then build on top of their knowledge.

– *Effective employment of use cases in the design of the actual product*. Most scenario and use case approaches are weak on the topic of how the results lead to new or improved systems. Consequently, scenario and use case approaches often do not contain the knowledge that is needed in the later stages of design.

– *Investigate metrics to predict system usability in the requirements phase*. During the requirement process, some kind of evaluation activity can take place. As soon as an initial draft of the requirement document is available, it can already be evaluated using scenarios and use cases. Consequently, when some initial sketches for the new system are known, mockups and prototypes can be used for early evaluation of design concepts. Also, early evaluation can be done by inspecting design specifications or by performing walkthrough sessions with designers and/or users.

– *Reducing the effort of developing a detailed use case*. Partly because of poor techniques, performing an effective use case takes too much effort. This often leads to rejection of the entire activity.

1.2 Thesis Outline

This thesis is structured as follows:

Chapter 2: surveys the scenario and use cases literature identifying issues underlying the scenario and use case-based approaches in use interface requirements engineering and proposing a framework for their classification.

Chapter 3: develop UCM-UI a conservative extension of UCM that provides new notations to model high-level requirements of user interfaces.

Chapter 4: propose a new framework for eliciting and specifying user interface and usability requirements. The goal is to build a complete and consistent user interface requirement framework that is simple, intuitive, unambiguous and verifiable with the help of the extended UCM-UI notations to better suit interactive systems, and by providing step-by-step guidance for the employment of use cases.

Chapter 5: presents number of design heuristics for constructing a formal specification and demonstrates that these heuristics may be used to build operators that validate the UCM-UI model.

Chapter 6: develops a practical suite of metrics; the UCM-UI Metrics Suite; to predict usability from scenarios documented as use case maps.

Chapter 7: Although UCM-UI models are semi-formal semantics, they can be used to guide the generation of more formal models and specifications for interactive systems. This chapter shows that the UCM-UI models can form the basis for further developments and presents how the new framework bridges the gap between requirements and detailed design models offered by UML, LOTOS specifications, and XML.

Chapter 8: concludes the dissertation by discussing the contributions achieved throughout this research. Finally, future work related to this dissertation is presented.

1.3 Publications

The chapters of this dissertation are largely based on previous published work or submitted for publication. This section lists the publications this dissertation is based on:

– van Der Poll, J.; Kotzé, P.; Seffah, A.; Radhakrishnan, T.; & Alsumait, A. (2003). Combining UCMs and Formal Methods for Representing and Checking the Validity of Scenarios as User Requirements. In *Proceedings of SAICSIT 2003*, September 17 – 19, 2003, Gauteng, South Africa. 111 –113.

– Alsumait, A.; Seffah, A.; & Radhakrishnan T. (2003). Use Case Maps: A Visual Notation for Scenario-Based Requirements. In *Proceedings of HCI International 2003*, June 22-27, 2003, Crete, Greece. 3-7.

– Alsumait, A.; Seffah, A.; & Radhakrishnan T. (2002). Use Case Maps: A Roadmap for Usability and Software Integrated Specification. In *Proceedings of IFIP World Computer Conference*, August 25-30, Montreal, Canada. 119-131.

– Alsumait, A.; Radhakrishnan, T.; & Seffah, A. (2001). Enhancing Use Case Maps for User Interfaces Requirements Engineering. Symposium on Human-Computer Interaction Engineering, May 11-13, 2001, Toronto, Canada.

Also two papers are recently submitted:

– Seffah, A.; Alsumait, A.; & Radhakrishnan, T.; van Der Poll, J. & Kotzé, P. (2004). User Requirements via Use Case Maps and Formal Methods: A Mixed Approach. *ACM Transactions on Computer-Human Interaction TOCHI*. (Submitted, February 2003).

– Alsumait, A.; Seffah, A.; & Radhakrishnan T. (2004). Supplementing Scenarios and Use Case Maps with Predictive Metrics. *IEEE Software metrics symposium*. (Submitted, February 2003).

Chapter 2

Literature Review

State of Art in Scenario-Based User Interface Requirements

Abstract

The election, analysis, and documentation of user interface requirements of complex system are crucial and non-trivial task. Well-defined concepts and methods are needed when constructing formal, agreed upon specification that represents the requirements in a *clear*, *consistent*, and *complete* manner. It is also important to have representations of the requirements that are easily understood by all parties involved in the requirement phase. In practice, there are diverse scenario-based approaches and representations that model the user interface. This chapter is an attempt to explore and examine some of the issues underlying scenario-based approaches in use interface requirements engineering and to propose a framework for their classification.

2.1 Introduction

A survey over 8000 projects undertaken by 350 US companies revealed that one third of the projects were never completed and one half succeeded only partially, that is, with partial functionalities, major cost overruns, and significant delays (Standish Group, 1998). The executive managers of those projects identified that poor requirements were the major source of problems; about half of the responses. The lack of user involvement (13%), requirements incompleteness (12%), changing requirements (11%), unrealistic expectations (6%), and unclear objectives (5%) were the major cause of poor requirements; Figure 2.1. Thus, success or failure of software development project greatly depends on the requirement process where user should be actively involved. The requirement process should focus on user characteristics, tasks, work environment as well as usability goals such as effectiveness, efficiency and user satisfaction.

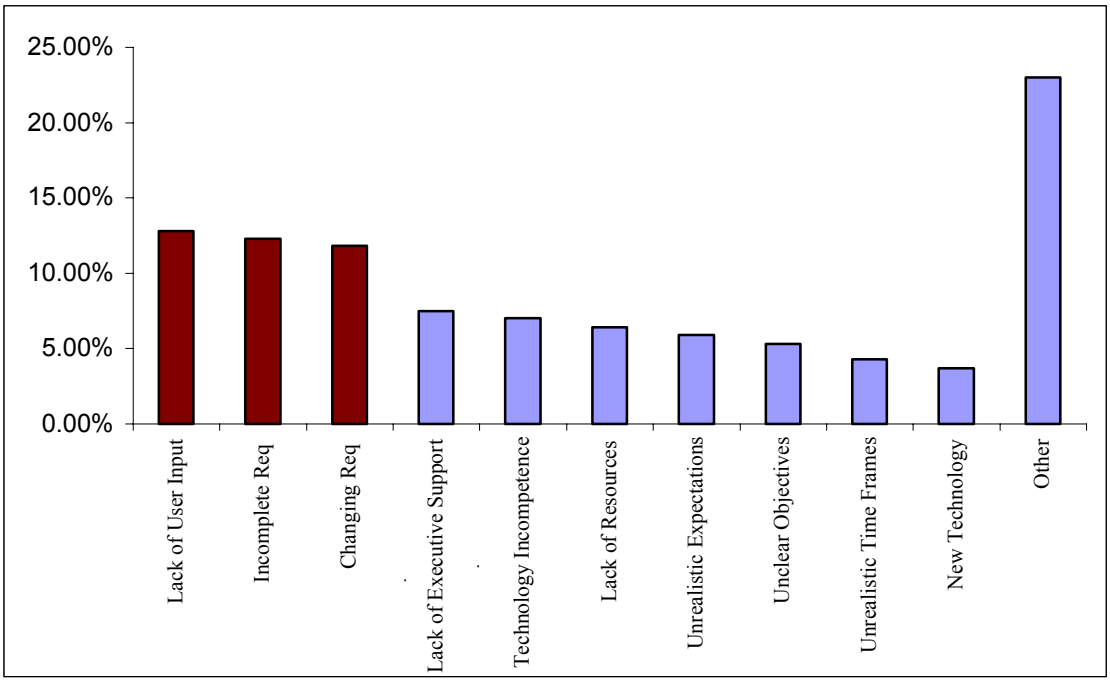


Figure 2.1 Project Failure Factors (Standish Group, 1998).

Requirement engineering has traditionally been concerned with investigating the goals, functions, and constraints of software systems. RE tasks includes elicitation of information related to the problem domain; modeling of the problem; analysis of costs, completeness, and consistency; and verification and validation. These tasks cover the way to generate complete, consistent, and unambiguous specifications of

system behavior that aid in the design and implementation activities. A major motivation for spending time and effort on requirements engineering and its improvement comes from the objective of doing the software development right from the beginning, instead of patching at the end. Experiments show that the cost of detecting and repairing errors increases dramatically as the development process proceeds (Davis, 1993; Nuseibeh and Easterbrook, 2000).

When applying requirement engineering to interactive systems, there is also a need to focus on multiple aspects that captures user interface requirements. People get frustrated with systems that do not adequately support them in their work. Often the software is of *high internal quality*, but when the system does not match the users' tasks and needs, internal software quality becomes almost *irrelevant* (Bevan, 1999). Users will try to avoid using such systems if they can, or may even reject using them at all. Success in different studies reset on the realization that user interface development is not software development.

This chapter is organized as follows: Section 2.2 describes the role of Requirements Engineering (RE) in software and systems engineering, the many disciplines upon which it draws, and the core of the RE activities. We also discuss the essential need of user participation during the RE process. In Section 2.3, we review several representations of scenario-based approaches for user interfaces. Our goal is not to conduct an exhaustive survey, but to show how the different goals of scenario analysts affect the representations they choose. In Section 2.4, we discuss when should scenarios be represented formally, when informally, and the pitfall of current scenario-based approaches for user interfaces. Section 2.5 concludes with a summary of the state of the art in scenario-based approaches, and offers a view of the key challenges for future scenario-based research.

2.2 Requirements Engineering (RE)

Requirements engineering has been recognized during the past 30 years to be a real problem. Bell and Thayer (1976) determined that inadequate, inconsistent, incomplete, or ambiguous requirements have a critical impact on the quality of the resulting software. By investigating different projects, they concluded that *"the requirements for a system do not arise naturally; instead, they need to be engineered and have continuing review and revision"* (Bell and Thayer, 1976). Several studies estimated that the late correction of requirement errors could cost up to 200 times as much as correction during such requirements engineering (Boehm, 1981; Davis, 1993; Nuseibeh and Easterbrook, 2000). Therefore, it is reasonable to believe that efforts spent on improving requirements engineering will pay off. In this section, the definition and dimensions of requirement engineering discussed. After a short discussion on the RE activities we dispute the vital need of user participation during the RE process.

2.2.1 Requirements Engineering Definition

Within software development lifecycle the early phase discipline, which comprises of the process of (a) analyzing and describing the problems a software- and hardware-based system has to solve in a given application domain and then, (b) prescribing the constraints and specifications for a new or changed system by transforming *fuzzy* initial ideas into *precise*, *commonly agreed* specification is called Requirements Engineering. The use of the term " engineering " implies that systematic and repeatable techniques should be used to ensure that system requirement are complete, consistent, relevant, etc.

There is no commonly accepted definition for Requirements Engineering, although there exist several standards. For instance, the IEEE standards (IEEE, 1984; IEEE, 1991) define RE as follows:

"(1) the process of studying user needs to arrive at a definition of system, hardware, or software requirements. (2) the process of studying and refining system, hardware or software requirements."

A requirement is defined as:

"(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or proposed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. (3) A documentation representation of a condition or capability as in (1) or (2)."

The result of the RE process is an unambiguous and complete specification document. This should help: the end-user to accurately describe what they wish to obtain from the system; the stakeholder to understand exactly what the end-user needs. It also helps the software engineers to develop a standard software requirements specification (SRS), define the format and content of their specific software requirements specifications; and develop additional local supporting items such as an SRS quality checklist, or an SRS handbook.

2.2.2 The Three Dimension of Requirements Engineering

At the beginning of the RE process *unclear* personal views of the system exist. Those views are usually recorded using *informal languages*, whereas at the end of the RE process a *complete specification* expressed using *formal languages* on which an *agreement* should be reached. Based on this characterization three dimensions RE are identified in (Pohl, 1994): (1) Specification, (2) Representation and (3) Agreement dimension (See Figure 2.2).

– *The Specification dimension*: As identified by many researchers, the first main goal of RE is to build a requirements specification, according to the standard and guideline used. The degree of the specification (unclear to complete) is captured by this dimension.

– *The Representation Dimension*: During the RE process different representation languages are used. At the beginning of the process the knowledge about the system is expressed using informal representations, while at the end of RE the specification must be formally represented. Thus, the second main goal of the RE process is threefold. First, different representations must be offered. Second, the transformation between the representations (e.g., informal to semi-formal, informal to formal) must be supported. Third, the different representations must be kept consistent.

– *The Agreement Dimension*: this dimension is as important as the *representation* and *specification* dimension. The coexistence of different views has positive effects on the RE process. Allowing different views and supporting the evolution form the personal views to a common agreement on the final specification is the third main goal of RE.

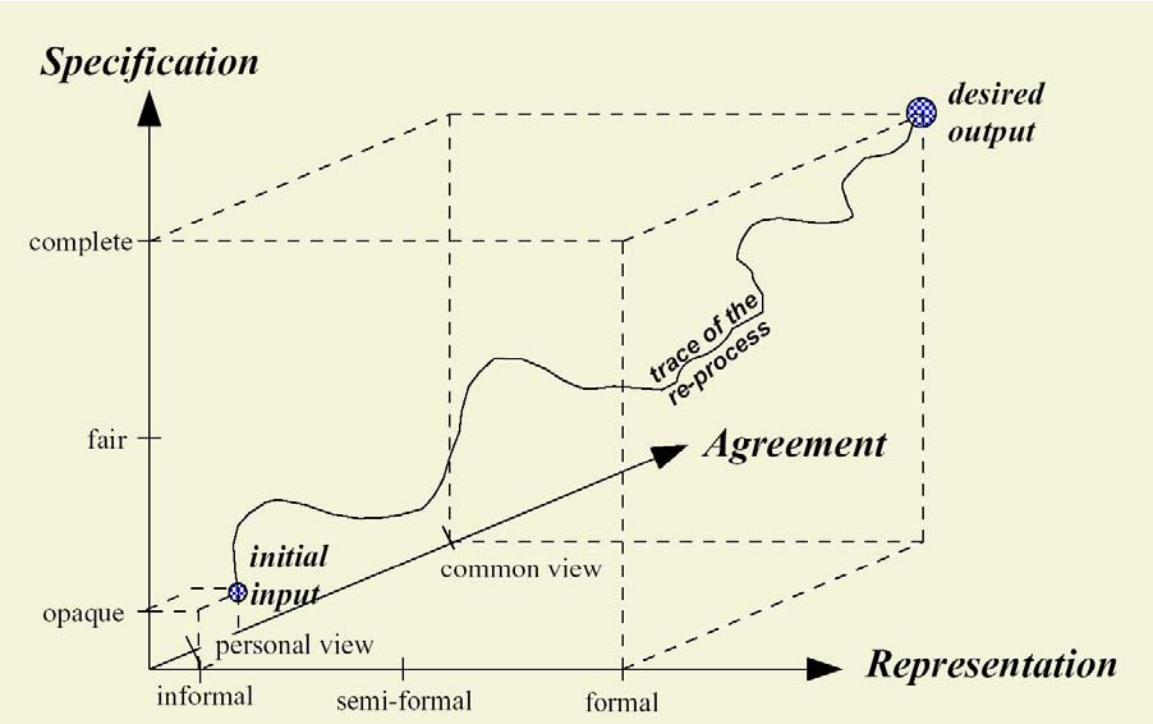


Figure 2.2 The RE process within the three dimensions (Pohl, 1994).

Looking at RE using these three dimensions, the main tasks and goals to be reached within each dimension during the RE process is identified. But RE is not only driven by its goals, it is also influenced by the environment. There are five main factors influencing requirements engineering: (1) methods and methodologies, (2) tools, (3) social aspects, (4) cognitive skills and, (5) economical constraints (Pohl, the 3 dimensions of RE). Accordingly, the first step into getting to the heart of RE is to distinguish between two kinds of problems:

– original requirements engineering problems and

– problems caused by approaches which try to solve the original problems.

Making the original RE problems and the goals to be reached during the process explicit provides the basis for classifying the research of the RE area and for guiding a RE process.

2.2.3 Activities of the RE Processes

According to Pohl (1993), requirements engineering is a *systematic* process of developing requirements through an *iterative co-operative* process of *analyzing* the problem, *documenting* the resulting observations in a variety of representation formats, and *checking* the accuracy of the understanding gained. Improving the quality of requirements is thus crucial; however, it is a difficult objective to achieve. RE is inherently iterative and consists of a number of interrelated sub-processes. A correct, consistent, and complete way to collect, understand, specify, and verify requirements is important. To achieve well-defined document containing the user requirements that satisfies these prerequisites van Lamsweerde (2000a) has tackled the RE process as follows:

- *Domain analysis*: in this process, existing system in which the software should be built is studied. The relevant stakeholders are identified and interviewed. Problems and deficiencies in the existing system are identified; opportunities are investigated; general objectives on the target system are identified there from.
- *Elicitation*: in this process, alternative models for the target system are explored to meet such objectives; requirements and assumptions on components of such models are identified, possibly with the help of hypothetical interaction scenarios. Alternative models generally define different boundaries between the software-to-be and its environment.
- *Negotiation and agreement*: in this process, the alternative requirements/assumptions are evaluated; risks are analyzed; "best" tradeoffs that receive agreement from all parties are selected.
- *Specification*: in this process, the requirements and assumptions are formulated in a precise way.
- *Specification analysis*: in this process, the specifications are checked for deficiencies, such as inadequacy, incompleteness or inconsistency. Moreover, specifications are checked for feasibility in terms of resources required, development costs, and so forth.
- *Documentation*: in this process, the various decisions made during the process are documented together with their underlying rationale and assumptions.

– *Evolution*: in this process, the requirements are modified to accommodate corrections, environmental changes, or new objectives.

Requirements engineering is a systematic process in which requirements engineers use different kinds of techniques or methods to achieve the real requirements of a system, which make it potentially successful. However, requirements derived from the use of traditional methods tends to subordinate interface requirements to the core of system functional requirements. As a result, visual and auditory features of the interface, together with dialog design, are added late in the development process, without proper integration into the system requirements (Parker et al., 1997). Moreover, these traditional methods lack user involvement and offer scant advice on the investigation of user-centered requirements. Their treatment of interface development is inadequate, as is their treatment of non-functional, qualitative requirements relating to system and user performance (Bickerton and Siddiqi, 1993). In fact, from a user's point of view, a requirement stage is necessary because it helps to understand the user's needs and to identify how they can be satisfied. Lutz (1993) studied the software errors in NASA's Voyager and Galileo programs and determined that the primary cause of safety-related faults was errors in *functional* and *interface requirements*.

2.3 User Participation in Requirements Engineering

User participation in developing complex systems has evolved rapidly over the past years. The user's role has changed from a passive one, with little or no involvement in software development lifecycle, to that of an active partner. Designing for product quality requires extracting detailed knowledge of the users. Thus, user participation is advocated in order to discover users' needs and points of view, validate specifications, resolve conflicts, and hence build better systems (Darke and Shanks, 1997; Koh and Heng 1996; Vredenburg et al., 2002).

The lack of user input contributes to incomplete requirements and specifications, because only the system users collectively have the necessary understanding of the needs to be fulfilled (Clavadetscher, 1998; Sack, 1985) and was considered to be at least partially responsible for considerable increases in the costs of developing systems at ITT and IBM (Mumford, 1985). Moreover, the process of developing the AS/400 system was deemed successful, at least in part, due to intensive early user participation (Pine, 1989; Sulack et al., 1989). Consequently, the user participation and influence are expected to increase the likelihood of user acceptance of the solution and of improved system quality (Berry, 1994; Vanlommel and de Brabander, 1989; Torkzade and Doll, 1994). In fact, there is a great deal of literature supportive of user participation or involvement in the requirement process of software developments (Amoroso and Cheney, 1992; Avison and Fitzgerald, 1995; Iivari, and Hirschheim, 1996; Silvestre, 1996; Kujala, 2002). For example, studies on software projects in USA show that user involvement is one of the three major reasons to succeed, beside executive management support and a clear statement of requirements (Standish Group, 1998). RE is not an activity one single person performs, but it is a rather cooperative and interdisciplinary endeavor. One of the most important aspects of this process is to achieve a *shared understanding* of the system to be built.

Nevertheless, well-known problems of communication occur due to the diversity of professional expertise and organizational roles that confer users' different views and expectations of the system to be developed. The users will always play the roles of problem owners, customers and actors, and identifying whether they formulate the *right problem* or formulate the *problem right* is anything

straightforward (Checkland, 1981). Also, interpersonal as well as political problems develop and make this process difficult. In this context, the requirements are regarded as a result of continuous negotiation, a process that goes beyond the one-way transfer of knowledge from the users to engineers, but is about developing an understanding of each other's views and perspectives, about sharing knowledge and about learning as a result of shared experience (Macaulay, 1996; Mumford, 1984; Haumer Thesis, 2000; Kujala, 2003).

2.3.1 The Challenge of User Participations in Requirements

A basic question in requirements engineering is how to find out what users really need (Goguen and Linde, 1993; Haumer et al., 1999). However, the elicitation of requirements information from users is frequently problematic, as discussed in (Palmer, 1987; Kujala, 2003). The problem can be attributed to poor communication between users and engineers. Misinterpretations are evident between them as both sides have different backgrounds, knowledge, vocabulary, and goals. The most frequently mentioned requirements elicitation problems and possible solutions are summarized in Table 2.1.

Problem	Solution
Users do not know what they want, or they cannot articulate it.	Alternative elicitation techniques such as field studies provide a more complete picture without the need for users to articulate their needs. Users are recognized as experts in their tasks; the focus is on their goals, present processes and context of use.
There are too many users to study.	Identify the various kinds of users and sample representative users from all essential groups. It is important to know how to determine who the users should be and how to elicit information from them.
A new product will provide a new way of carrying out the existing tasks.	In order to understand the user needs: The pros and cons of the present way of achieving the user goals must be identified

Problem	Solution
	The future context of use must be identified Users should be allowed to use their skills, the advantages of the current processes should be saved and the problems fixed.
Users have poor understanding of computer capabilities and limitations Users may request a specific feature or technical solution.	The underlying user needs should be discovered. The users may believe this feature solves their problems but it may not be an optimal solution.
Users and requirement engineers speak different languages	The language used to express the requirements back to the user may be too formal or too informal to meet their needs because of the diversity of the communities. Language and terminology understandable by both the users and the requirement engineers should be used.
Users do not have the ability to abstract and structure their knowledge nor understand abstraction made by the requirement engineer	User reacts to concrete instance and examples especially when they come from their own familiar working environment.

Table 2.1 The problems and solutions of requirements elicitation.

2.3.2 Scenarios: Bridging the Gap between User Needs and Requirements

Early user involvement plays a role in understanding user needs, including context of use, in the early stages of product development. Field methods can be recognized as most promising in understanding user needs as users are studied in their own environment. However, field studies are generally considered time consuming and effort intensive and thus they add to product development costs (Bly, 1997).

Scenarios are a possible solution to bridge the gap between user needs and requirements. In general, scenarios represent concrete examples of current and future system usage. A scenario is a partial description of system behavior which, when combined with other scenarios, should provide a more complete system description. This makes scenarios particularly well suited for incremental requirement elicitation.

In addition, the use of scenarios improves the quality of RE process and the descriptions produced, because scenarios usually are less abstract and therefore easier to develop and communicate; especially by those whom uses with less formal training but important domain knowledge (Haumer et al., 1999; Ben Achour et al., 1999; Hertzum, 2003, Sutcliffe, 2003).

Scenarios are used for a variety of different tasks and to accomplish a variety of specific goals, for example: (1) in requirement phase, to represent the needs apparent in current work practice (Jacobsen, 1995); (2) in user-designer communications, as a mutually understood means of illustrating important design issues or possible designs (Kyng, 1995); (3) in software design, as a means to identify the central work domain objects that must be suitably included in the system; (4) in documentation and training, as a means to bridge the gap between the system as an artifact and the tasks that users want to accomplish using it; and (5) in evaluation, as a means of defining the tasks the system has to be evaluated against (Nielsen, 1995).

Accordingly, using scenarios during the requirements discussion phase provide stakeholders and end-users with a deeper understanding of the requirements, provide answers to questions about requirements, and help them notice inconsistencies or missing requirements. Research literature offers an increasing number of scenario related methods, models and notation that are reported in the next section.

2.4 Scenarios in User Interface Requirement Engineering

In the past thirty years, Human-Computer Interaction (HCI) community has been considerably attracted by the use of *scenarios*. The HCI community proposes a large variety of scenario-based approaches emphasizing more on user-oriented perspectives in developing software systems. Scenarios promote shared understanding of the current situation and joint creativity toward the future (Jarke et al., 1998; Weidenhaupt et al., 1998; Robotham and Hertzum, 2000). The main purpose of introducing scenarios is to stimulate thinking, e.g. scenarios are *"a creative tool that facilitates the leap from observation to invention"* (Verplank et al., 1993). This is also apparent in Carroll's definition of the concept: *"The defining property of a scenario is that it projects a concrete description of activity that the user engages in when performing a specific task, a description sufficiently detailed so that design implications can be inferred and reasoned about"* (Carroll, 2002).

Five key properties of scenarios motivate their widespread use in HCI specifically in user interfaced design are address in (Carroll, 1999): (1) scenario evokes reflection in the content of design work, helping developers coordinate design action and reflection and address some of the most difficult properties of design. (2) scenarios are concrete design proposal that a designer can evaluate and develop, but it is also rough in that it can be easily altered and allows many details to be deferred. (3) scenarios afford multiple views of an interaction, diverse kinds and amounts of detailing, helping developers manage the many consequences entailed by any given design move. (4) scenarios can also be abstracted and categorized, helping designers to recognize, capture, and reuse generalizations, and to address the challenge that technical knowledge often lags the needs of technical design. (5) scenarios promote work-oriented communication among stakeholders, helping to make design activities more accessible to the great variety of expertise that can contribute to design, and addressing the challenge that external constraints designers and users often distract attention from the needs and concerns of the people who will use the technology. The five key properties of scenarios are illustrated in Figure 2.3

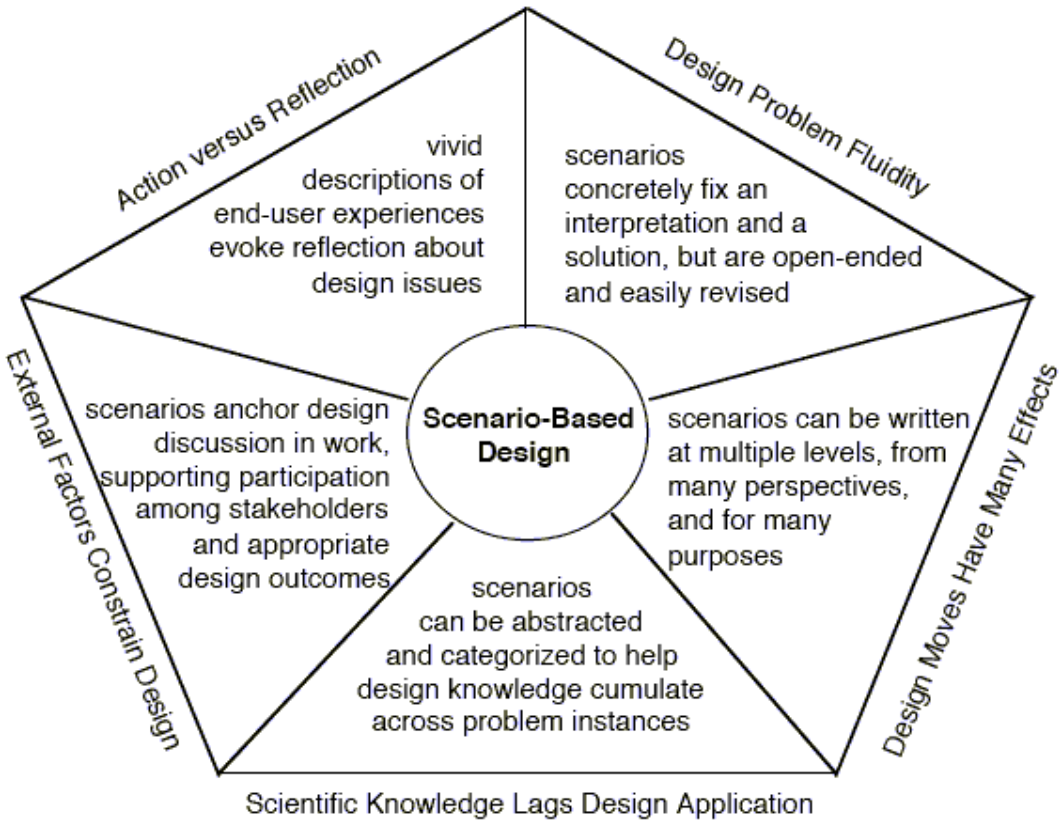


Figure 2.3 Challenges and approaches in scenario-based design (Carroll, 1999).

Scenarios for user interface design may be represented in a variety of media, either natural language text, graphics, images, videos or designed prototypes. Furthermore, it may exist a modeling language providing semi-formal / formal notations. Descriptions may also be informal as they are expressed using concrete terms of the reality. The purpose of this section is not to simply survey scenario-based approaches, but rather to indicate where and how user interface designers use scenarios. Additionally, we examine how their individual goals affect the representations they choose.

2.4.1 Scenarios in Task Analysis for Human Interaction

Scenarios specify how users carry out their *tasks* in a specified context. They provide examples of usage as an input to design, and provide a basis for subsequent usability testing. Thus, much of the HCI work involving the use of scenarios does not refer to them by name, but instead uses the term *task*. A task description can be a scenario by another name, since it describes a single trace of behavior, or a narrow range of alternatives. Also, task description provides an explanation of how the user

interacts with a system to perform a high-level task and incorporates all the envisaged contingencies and alternatives (Antón and Potts, 1998a). Task analysis is the process of gathering data about the tasks people perform and acquiring a deep understanding of it. Several methods can be used to gather the data, including interviewing, observation, talk aloud protocols, and ethnographic work place studies. While these techniques each have their own characteristics and problems, documenting the data and being able to write down the understanding gained is another problem. This section presents some of the task; scenario-like; modeling techniques that are used once data is available.

One of the oldest general-purpose task description techniques and contains many ideas found in later techniques is the Hierarchical Task Analysis (HTA), proposed by Annett and Duncan in 1967. HTA is a process of developing a description of tasks in terms of *operations* and *plans*. Operations are things people do to reach goals and plans are statements of conditions that tell when each operation is to be carried out. The operations can be hierarchically decomposed and with each new sub-task, a new plan exists. In HTA, tasks are defined as activities that people do to reach a goal. A goal is then defined as a desired state of the system under control or supervision. However, in the notation used, only a task hierarchy is modeled and the goals are not explicitly represented.

Another example of a task specification formalism that uses a narrow, scenario-like sense of the term task is *GOMS* (Goals, Operators, Methods, Selection Rules) a method for specifying task-specific user/computer interaction sequences and evaluating user interface designs (Card et al., 1983). GOMS is a representation of the "*how to do it*" knowledge that is required by a system in order to get the intended tasks accomplished. Briefly, a GOMS model consists of descriptions of the *Methods* needed to accomplish specified *Goals* (e.g., moving the mouse and clicking a button for selecting a displayed object). The Methods are a series of steps consisting of *Operators* that the user performs. A Method may call for sub-Goals to be accomplished, so the Methods have a hierarchical structure. If there is more than one Method to accomplish a Goal, then the GOMS model includes *Selection Rules* that choose the appropriate Method depending on the context (e.g., deleting a paragraph by repeatedly deleting text lines or by marking its beginning and end and then cutting the selected text) (John and Kieras, 1996).

Today, there are several variants of the GOMS analysis technique (e.g. Keystroke-Level Model (KLM), Cognitive-Perceptual-Motor (CMN-GOMS), Natural GOMS Language (NGOMSL), etc.), and many applications of the technique in real-world design situations. Most of the GOMS analysis techniques are related to a general task-analysis approach. This general approach emphasizes the importance of the procedures for accomplishing goals that a user must learn and follow in order to perform well with the system. By using descriptions of user procedures, the techniques can provide *quantitative predictions* of procedure learning and execution time and *qualitative insights* into the implications of design features. While other aspects of system design are undoubtedly important, the ability of GOMS models to address this critical aspect makes them not only a key part of the scientific theory of human-computer interaction, but also useful tools for practical design.

GOMS grows out of an engineering psychology approach to the design of user interfaces; the user's behavior is modeled in great detail, down to the quantification of basic parameters and the assignment of probabilities to selection rules or parameter ranges. For example, GOMS tries to estimate learning time, execution time and mental workload solely by counting steps. It is assumed that the steps modeled by GOMS statements normally take 0.1 sec and primitive operators such as mouse movements take between 0.2 and 1.1 sec. Mental operators are estimated to take 1.2 sec in case of lack of any other information. For experienced users this time should sometimes be zero seconds. The assumed values for operators have been heavily criticized in (Nielsen, 1993). Their values have proven to be rather variable and unpredictable which makes the foundation for estimating user performance weak, if not invalid. Other criticisms concern the fact that GOMS assumes error-free behavior and does not distinguish novice and expert behavior. The entry point for GOMS is the user's assumed goal in performing a task. Task descriptions are hierarchical descriptions of the methods (sequences of steps) that can be performed to accomplish the goal. In GOMS, a scenario is referred to as a task instance. Example on GOMS analysis is shown in Figure 2.4.

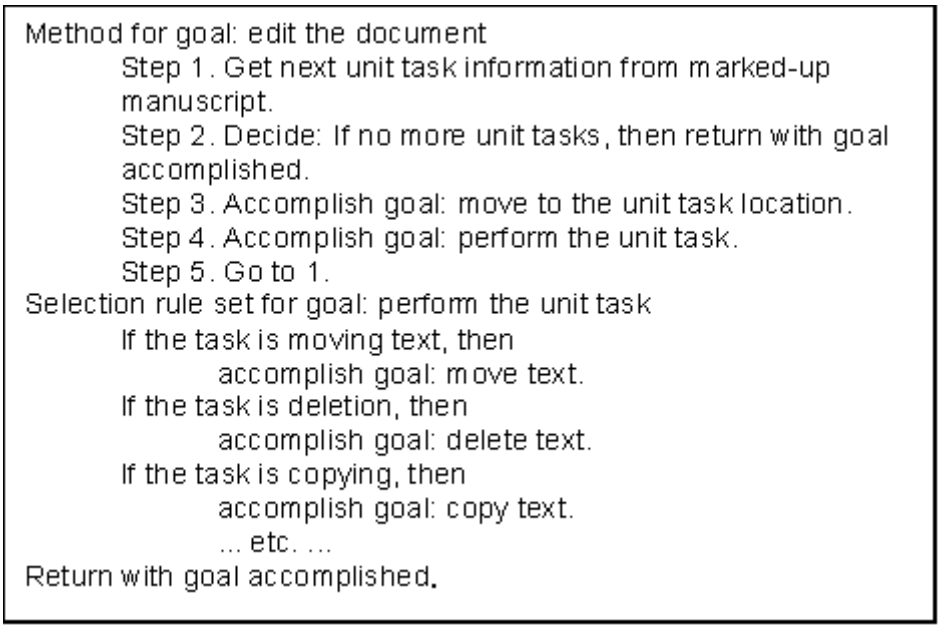


Figure 2.4 Example on GOMS.

Operational Sequence Diagrams (OSD) is another human performance engineering approach to design and evaluate interfaces using task models. It presents the information from the point of view of the user. OSDs are extended forms of Flow Process Charts, which provide a graphic presentation of the flow of information, decisions, and activities in a system, using a set of five basic graphic symbols and an associated grammar (Kurke, 1961; Kirwan and Ainsworth, 1992). The technique is tailored for the representation of the flow of information, with symbols for the transmission, receipt, processing, and use of previously stored information. The diagrams show the sequence of tasks or actions in a vertical sequence: they can be annotated with time information or a time line. The standard operation symbols of OSD are shown in Figure 2.5. It shows the vertical time line, the scenario components, and the allocation of responsibility between user and system.

Figure 2.5 The standard operation symbols of OSD.

M´ethode Analytique de Description des t^aches (MAD) is example on task modeling part of a larger method for designing interactive systems (Scapin and Pierret-Golbreich, 1989). In MAD, task models are similar to HTA models except that the plan has been replaced by *constructors*. A constructor specifies the time dependencies of a task's subtasks i.e. a constructor scopes over all subtasks. The constructors are used to specify the time order in which tasks are executed. Additionally, pre-conditions can be specified for each task in order to "tune" the time ordering.

The most interesting aspect of MAD is the fact that *templates* are used to describe the task details (Figure 2.6). Details include pre- and post-conditions, initial and final states, task types, and priorities. In MAD, a task consists of two parts: a condition part and a *body* part. In most other methods a task is regarded as a ”black box” but MAD has shown that there are many relevant aspects of a task to be described. Moreover, tasks are modeled in great detail but no other concepts are modeled, i.e., no roles or objects.

Task Body attributes	
Identification number	Alphanumeric
Name	Alphanumeric
Goal	Alphanumeric
Comments	Alphanumeric
Degree of Freedom	{Optional, Obligatory}
Interruptability	{True, False}
Upper Task	Identification Number
Priority	Integer
Modality	{Automatic, Interactive}
Type	{Cognitive, sensors-motor}
Frequency	{High, Medium, Low}
Centrality	{Important, Not important}
Experience	{User: novice, occasional, expert}
Mandatory sub-tasks finished	Integer

Figure 2.6 MAD task body attributes (van Welie, 2000).

User Action Notation (UAN) was developed out of the need of communication between implementers and designers (Hix and Hartson, 1998). UAN is a tabular notation in which the user, interface, and underlying computation are treated as three agents involved in a scenario. The user operations are specified in great detail. The interface operations (e.g., changes to display) and its state (e.g., change of mode) are specified informally or by reference to separate pictures, and the underlying computation is described informally also. The technique consists of two types of diagrams; interaction templates and composite templates. Interaction templates are used for describing the actual interaction in detail using four columns

(user action, system feedback, interface state and connection to computation). The composite template is used to describe a hierarchical decomposition of interaction templates. UAN focuses on the interaction between user and the system instead of the system's states as STDs do. van Welie et al. (2000) developed an extension to the UAN, called New User Action Notation (NUAN) to improve and overcome some of the problems in UAN. This made UAN more suitable for task-based design.

A final example of a human performance-engineering notation for tasks is the ConcurTaskTrees (CTT) (Paternò, 1999). The purpose of a task model specified in CTT is to provide a description of how the activities should be performed in order to reach the user's goals. Such activities are described at different abstraction levels in a hierarchical manner, represented graphically in a tree-like format. In contrast to previous approaches, ConcurTaskTrees provides a rich set of operators with a precise semantics to describe the temporal relationships among such tasks. The notation gives also the possibility to use icons or geometrical shapes to indicate how the performance of the tasks is allocated: only to the user, only to the application, interaction between user and application, abstract tasks (which means that they have subtasks allocated differently). For each task it is possible to provide additional information including the objects (for both the user interface and the application) manipulated. Moreover, the notation is supported by the ConcurTaskTrees Environment (CTTE), a set of tools supporting editing and analysis of task models.

All examples- HTA, GOMS, OSD, MAD, UAN, and CTT- seek to achieve highly precise procedural specifications, so that a task analyst knows exactly what steps to perform, how to perform them, and what order to perform them in. This is highly desirable, and especially important if the task analyst is only marginally competent or not very creative. This is where structured methods really prove their value. However, the cost or downside of well-specified method scripts is that they lack flexibility. Also, most task analysis techniques assume error-free, non-interrupted action sequences.

Some examples of tasks analysis techniques such as GOMS models are quite effective because they capture procedural speed and complexity. But other aspects of human performance with an interface are not addressed by the simple cognitive architectures underlying the current GOMS variants. Also, the estimation part of GOMS remains controversial even after almost twenty years of research. However, it

can still be used to compare design alternatives against each other to see which requires the minimal number of steps.

2.4.2 Scenarios in Storyboarding and Prototyping

Scenarios help requirements engineering team handle the complexity of user interfaces by clarifying the actual behavioral penalty of requirements proposals. Another related design strategy that does this is *prototyping*. In fact, one of the principal outputs of scenario development is a road map for prototyping activities. Prototypes are built to support a variety of system life cycle activities including marketing, user requirements definition and validation, user-system interface design, system sizing, usability testing, documentation and training. One of the most productive uses of prototyping to date has been as a tool for iterative user requirements engineering and user interface design (Overmyer, 1999; Weidenhaupt et al., 1998; Elkoutbi et al., 1999; Harel, 2001).

Prototypes can be generally classified into two categories: low-fidelity and high-fidelity. Low-fidelity prototypes or *storyboards* segment behavior into several discrete frames. Each frame uses a mixture of text and pictures to depict the system. They are constructed to depict concepts, design alternatives, and screen layouts, rather than to model the user interaction with a system (Rudd et al., 1996). In contrast, high-fidelity prototypes are fully interactive representing the core functionality of the product's user interface. High-fidelity prototypes are typically built with fourth-generation programming tools such as Smalltalk or Visual Basic, and can be programmed to simulate much of the function in the final product (Rudd et al., 1996).

Low-fidelity prototypes and storyboards are inexpensive and can be build fast, but they fail to show navigation and flow. They are usually demonstrated by the designer rather than tested by a user providing limited usefulness for usability tests and user evaluation (Nielson, 1990; Rudd et al., 1996).

On the other hand, usability testing can be conducted early in the design process with the high-fidelity prototype as a test vehicle. Realistic comparisons with competing products can be made via the prototype to ensure that the program is marketable and usable before committing the resources necessary to develop the product fully. However, high-fidelity prototypes are more expensive and time consuming to construct than low-fidelity prototypes. Since high-fidelity prototypes

represent function that will appear in the final product, prototype development becomes a development effort in itself, sometimes requiring many weeks of programming support. The cost and time saving for high-fidelity prototypes can be achieved by somehow reducing the prototype as compared to the full functional system either by developing vertical prototype; an interactive, high-fidelity prototype of only a subset of the product's available function, or by creating horizontal prototypes; prototypes that contain high-level functionality, but do not contain the lower-level detail of the system.

Virzi and his colleagues in (1996) conducted two experiments to compare the usability problems uncovered using low- and high-fidelity prototypes. In both experiments, substantially the same sets of usability problems were found in the low- and high-fidelity conditions. Moreover, there was a significant correlation between the proportion of subjects detecting particular problems in the low- and high-fidelity groups. Individual problems were detected by a similar proportion of subjects in both the low- and high-fidelity conditions. Finally, they concluded that the use of low-fidelity prototypes can be effective throughout the product development cycle, not just during the initial stages of design.

Actually, Scenarios can be used independently of prototyping, and prototypes can be used in the absence of scenarios. In spite of this, Weidenhaupt and his colleagues in (1998) examined 15 projects in four European countries. In two-thirds of the projects, scenario generation interrelated with prototyping. The initial scenarios served to validate the prototypes and, indirectly, the requirements specification. Moreover, validating prototypes against the initial scenarios allows the engineers to validate the initial scenarios themselves to detect missing functionality, over-specifications, errors, and even unintended side effects. Thus, scenarios and prototypes complement each other in a symbiotic manner. It remains an open issue, however, how to apply scenario representations and prototyping in a controlled and beneficial way.

2.4.3 Scenarios in Use cases

Jacobson (1992, 1995) introduced use cases as a part of object-oriented methodology. The idea was to capture brief narratives describing possible uses of the system under design as distinct cases; these could then be implemented in the design

one at a time. There is no one agreed definition on scenarios and use cases. However, in this thesis we use Cockburn (1999, 2001) definition, "*each use case is a collection of possible scenarios between the system and actors, characterized by the goals the primary actor has, while a scenario is a sequence of interactions happening under certain conditions*". Jacobson (1992) employs a graphical use case model, which includes the system as bounded by a box, each actor represented by a person outside the box, and use cases represented as ellipses inside the box. Rumbaugh (1994) complemented the model by proposing a written description of use case including name, summary, actors, preconditions, description, exceptions, and post conditions.

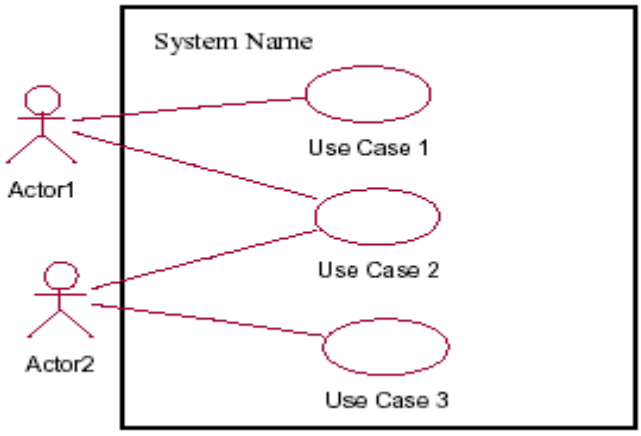


Figure 2.7 Example on use case.

Ever since, Jacobson (1992) introduced use cases, they have been considered to be a good way of capturing the users' needs and requirements (Rumbaugh, 1994; Lee and Xue, 1999) and of modeling functional requirements (Chandrasekaran, 1997). Jacobson's original idea was that by use cases the whole system development starts from what the users wish to be able to do with the system. In this way, the system is built from the users' point of view. However, in practice it is different, use cases are written without any knowledge of user needs and the documents are not read by users as Jacobson recommended. Use cases are written from a technical or interface point of view, but not from the user point of view. The resulting use cases are nearly impossible to understand by users (Lilly, 1999; Lee and Xue, 1999). The rest of this section presents examples on use cases that elicit and represent user interface requirements.

Constantine in (1995) proposed essential use cases, a technology-free, idealized, and abstract descriptions of user interfaces. Essential use cases highlight the purpose, what it is that users are trying to accomplish, and why they are doing it. The resulting design models leave more options open making it easy to accommodate changes in technology. Essential use cases are based on the purpose or intentions of a user, rather than on the concrete steps or mechanisms by which the purpose or intention might be carried out. It has been recommended by Constantine to use essential use cases as a starting point to capture the requirements and then evolve

them into system use cases during the analysis and design efforts. An example on essential use cases for getting cash from an ATM machine is illustrated in Figure 2.8.

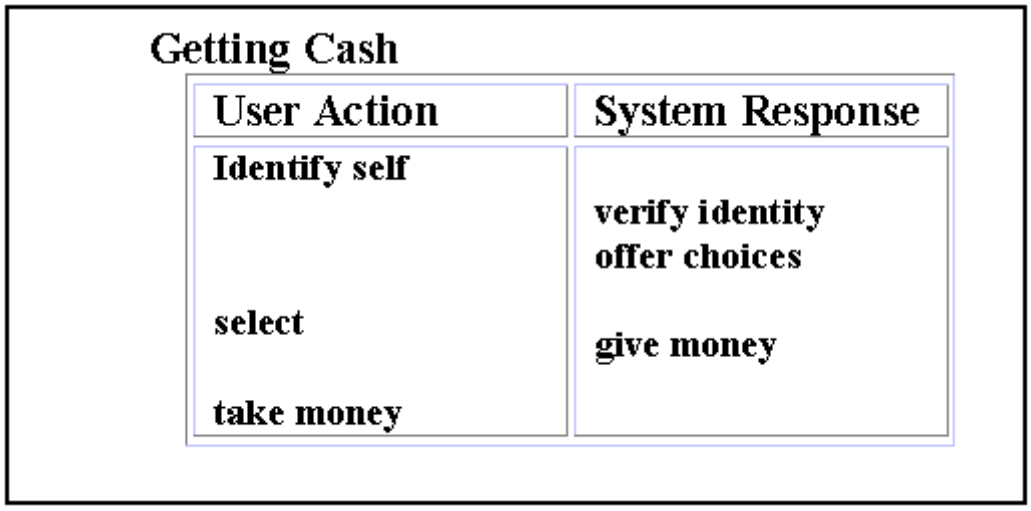


Figure 2.8 Essential use cases for getting cash (Constantine, 1995).

As shown in Figure 2.8, essential use cases are documented in a format representing a dialog between the user and the system. The general format divides the use case into two columns *user intention* and *system responsibilities*. The new labels indicate how essential use case support abstraction by allowing the interaction to be documented without describing the details of user interface. In fact, the abstraction does not really relate to use case as whole, but more to the steps of the use case. In this way, an essential use case does not specify a sequence of interaction, but a sequence with abstract steps.

Conversely, Wiegers in (1997) used dialog maps; a method for modeling user interfaces. Wiegers emphasizes on the goal of having the software developers and their customers achieve a shared vision of the product being created by conducting *use case workshops* to walk through use cases in order to identify the actual functional requirement, exception and decision situations. Dialog map is a modification of the state transition diagram (STD). A user interface resembles a finite state machine in that only one screen (state) is active at any given instant, and a set of defined navigation pathways and triggers (transitions) exists for moving from one screen to another (Wiegers, 1997).

As shown in Figure 2.9, each screen (rectangle) is identified only by name, with no detail at all shown about its fields or layout. The connections between one

screen or window and another are shown as transition lines connecting the states. Dialog maps can be structured hierarchically to further control the degree of detail revealed at any specific level (Wiegers, 1997).

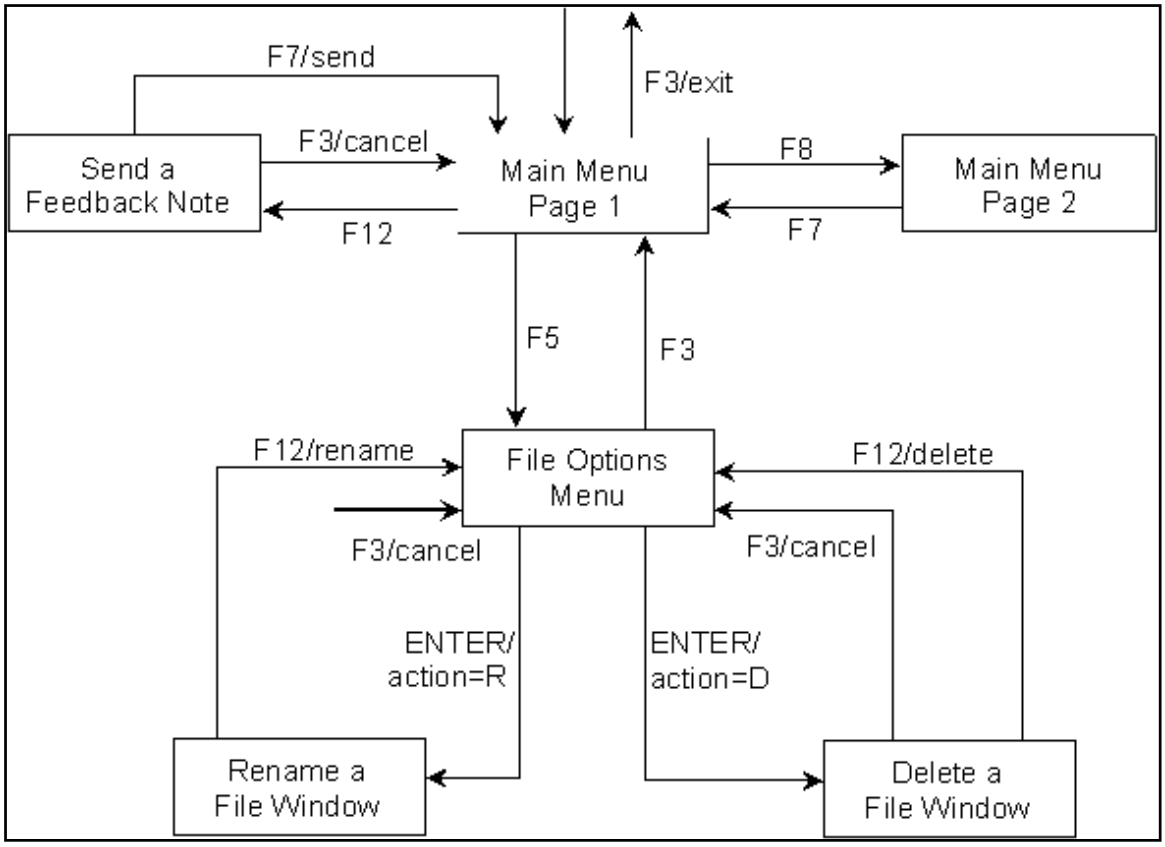


Figure 2.9 Sample dialog Map (Wiegers, 1997).

With the dialog map, common or similar interface features can be identified and built as reusable components. Any duplicated or missing functionality can be detected and corrected well before implementation. Dialog maps can be drawn with CASE tools, or they can be drawn with a pencil and paper.

The Rational Unified Process (RUP) is another use case driven iterative software engineering process. User Interface design within the RUP involves user interface modeling and user interface prototyping. The main input to these activities is the use case model, which describes how the system is used. At the user interface modeling stage, each use case is described via a *use case storyboard,* which is a conceptual description of how a use case is to be supported by the user interface (Krutchen et al., 2001). Phillips and Kemp in (2002) describe two support artifacts – *extended tabular use cases* and *UI element clusters* - which provide bridge between the two central activities within the RUP These artifacts provide support for 'flow of events' storyboarding, the clustering of user interface elements identification of UML boundary classes, and the initial sketching of user interface prototypes.

The *extended tabular use case representation* provides a flow of events storyboard, which is graphical as well as textual. The visual depiction of UI elements makes this easy to understand by all involved in the development process. It also provides the basis for the generation of the set of transactions for testing the implementation. The *UI element cluster* is used by the designer to proceed to user interface sketches and lo-fidelity prototypes that support the activities of the use case. These sketches tie down the "*look and feel*" of the interface, and are concerned with space allocation and layout, the behavior and appearance of widgets, and navigation between contexts. Figures 2.10 and 2.11 illustrate an example of the extended tabular use case and the UI element cluster of a library system.

In SCRAM (SCenario-based Requirements Analysis Method), scenarios are used with early prototypes to elicit requirements in reaction to a preliminary design. The approach is based on the hypothesis that technique integration provides the best avenue for improving RE and that active engagement of users in trying out designs is the best way to get effective feedback for requirements validation (Sutcliffe and Ryan, 1998; Sutcliffe 2003).

SCRAM is based on four techniques for requirements capture and validation: (1) *Use of prototypes or concept demonstrators*: a key concept is providing a designed artifact which users can react to. (2) *Scenario*s: the designed artifact is situated in a context of use, thereby helping users relate the design to their work/task context. (3) *Design rationale*: the designer's reasoning is deliberately exposed to the user to encourage user participation in the decision process. A QOC (Questions, Options,

Criteria) notation is used to illustrate the various trade-offs. (4) *Whiteboard summary*: the designer's requirements are summarized on a whiteboard to identify dependencies and priorities. SCRAM, however, gave only outline guidance for a scenario-based analysis.

Consequently, use cases have proven to be a popular and effective approach for capturing requirements and providing tracability that facilitate testing and validation (Hurlbut, 1997). On the other hand, studies in this area indicate that the current use case modeling techniques lack the level of formalism necessary to adequately support user interface (Constantine and Lockwood, 2000; Chin et al., 1997; Wiegers, 1997).

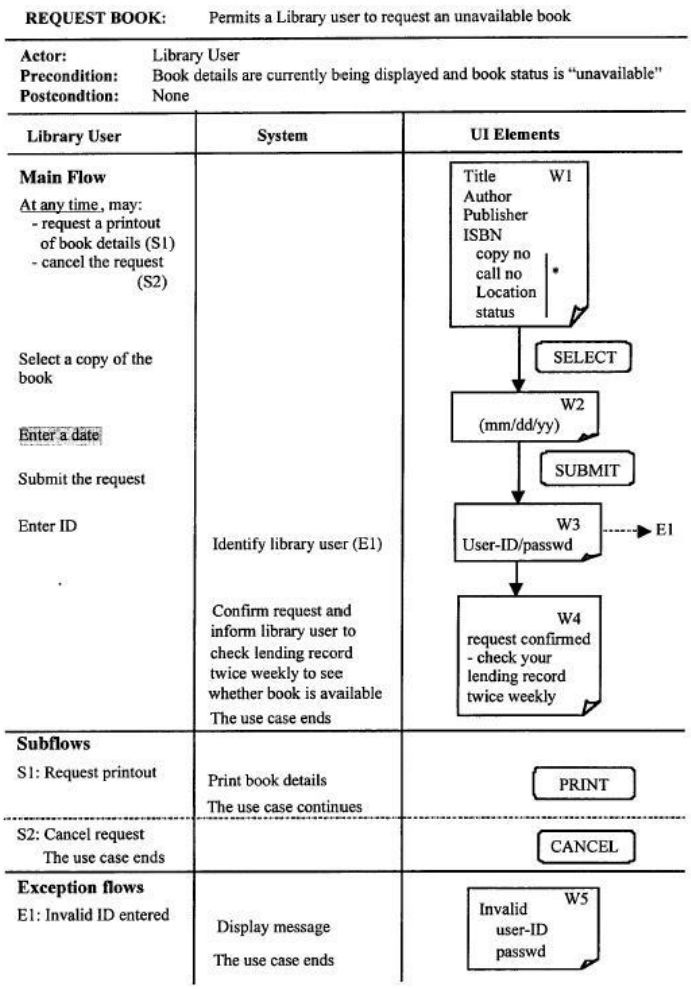


Figure 2.10 Extended tabular representation for the *Request Book* use case (Phillips and Kemp, 2002).

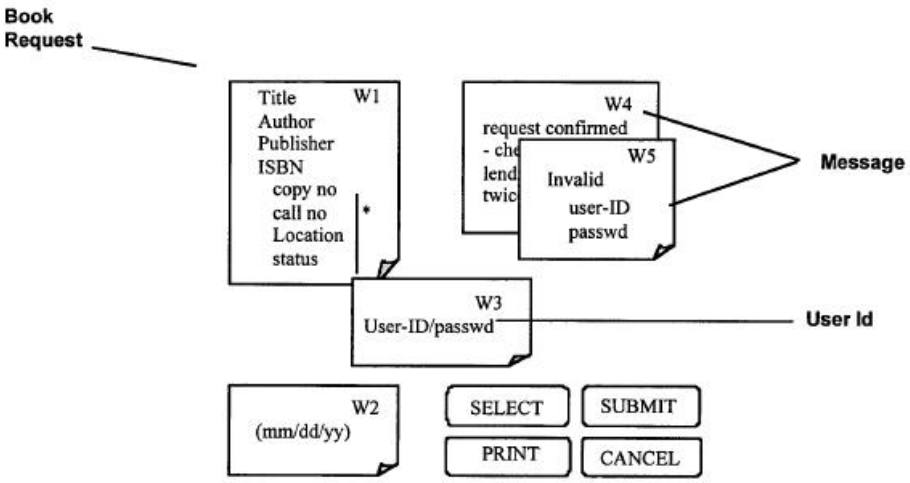


Figure 2.11 Element cluster for the *Request Book* use case, with the boundary classes labeled (Phillips and Kemp, 2002).

2.5 General Discussions

In this section we discuss when should scenarios be represented informally/formally, managing the transition between informal and formal representations, and the appropriate level of abstraction in a scenario, given a certain purpose. We also summarize the general pitfalls of current scenario-based approaches for user interfaces.

2.5.1 Scenario Representations: Informal versus Formal

There is a wide range of scenario representations including (1) raw information, (e.g., video recordings, literal transcripts) (2) free format data (e.g., pictorial descriptions, free form text), (3) structured representation, (e.g., structured texts, templates/ forms), (4) semi-formal syntax with some semantics (e.g., process maps in system analysis, message sequence diagrams, state charts with embedded text, pseudo code), and (5) formal languages with well-defined semantics (e.g., state charts, Petri nets, logic of action) (Jarke et. al., 1998). Figure 2.12 illustrates the different scenario representations.

In practice, scenarios have been described in a variety of media. Narrative text is probably the most common, as many authors associate scenarios to narration of "stories" (Carolls, 1995). Jacobson et al. in (1992) express use cases and event traces with structured English with the claim that narrative texts facilitate the capture of requirements. There are a number of formatted variants such as tables (Potts et al., 1994), structured texts (Regnell et al., 1995) or scripts (Rubin and Goldberg, 1992). Scenarios can also run as simulations to present a future vision of how the system will behave. For example, Benner et al. (1993) make scenarios themselves executable or visualized in order to watch the system as it runs, and detect weather or not the behavior patterns described in scenarios occurs. Also, the underlying language of scenario descriptions proposed by Glinz (1995) is a state-chart-based model. In the same line, Hsia et al. (1994) show that a scenario can be adequately represented by a regular grammar from which a conceptual machine for prototyping may be constructed.

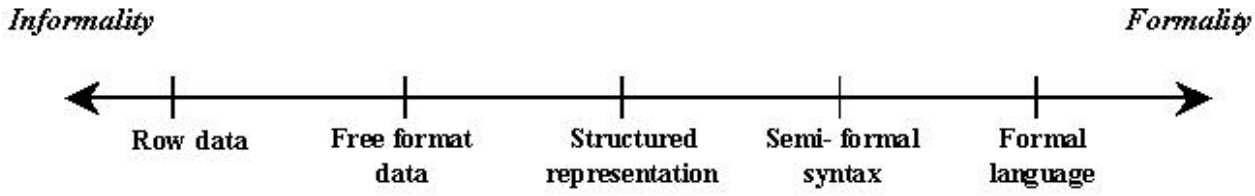


Figure 2.12 Various scenario representations.

Based on the purpose and on the intended users of the scenarios, the degree of scenario formality would change. For example incases where there is a need to represent the result of an agreement process, or a consolidated view is required, formal scenario representations are need. Also, formality is used when there is a strong need for traceability and unambiguity. On the other hand, incases where eliciting a specific view, or setting of particular views sufficiently, informal representations are recommended. Also, informality is used when there is a need for rapid feedback cycles between interface designers and stakeholders.

Effective use of scenarios depends on how the representation of the scenario accords with the purpose of the requirement engineering team. Sometimes there is a need for precision while creativity often asks for informal techniques. Most representations of scenarios in practice have been found to be 'informal' in a computer science sense. However, if these scenarios become too many, too broad or too deep, it is time for more general conceptual models validated against individual scenarios in a more formal manner. Yet, formalization is by no means synonymous with greater coverage or more detail. Accordingly, *semi-formal* representations will be sufficient to adequately represent structural constraints within and between scenarios (Antón and Potts, 1998b; Jarke et. al., 1998; van Lamsweerde, 2000b).

2.5.2 Common Pitfalls in Developing Scenarios-Based RE

Despite its usefulness, scenario-based requirement engineering is not a magic potion. Scenarios are a process, not a product. The process demands significant effort, thought, and creativity of the requirement-engineering team. An important question when applying scenario-based requirement engineering for user interfaces is: What criteria should we have for creating and describing scenarios for user interfaces? From the viewpoint of HCI, dealing with scenarios; narrative, rich, non-formal descriptions; is not considered a choice but forced on research by practice (Jarke et al., 1998). To

help the requirement-engineering team avoid mistakes made by others, common pitfalls are listed below:

- Scenarios do not produce action plans; they help requirement-engineering team imagine what will happen. The methodology is not suited to addressing specific planned issues. It is meant to provide a broad view of the uncertainties facing an intervention. Strategic decisions flow from this understanding, but they are not a direct product of the exercise.
- Deriving scenarios from user interface perspective rather than from the actual user goals and objectives yields too much implementation specific detail which should be addressed during software design, not requirements engineering (Lilly, 1999).
- A weakness of the scenario-based approach in practice is that the gap between a textual/informal description and the more formal representation of a scenario is very wide. Traceability between scenarios at these varying levels of detail becomes a concern.
- Considering the difficulties in comprehending scenarios written in a natural language it may be worthwhile to invest in implementing a prototype that visualizes the system. The prototype may be employed in preliminary usability tests (Zowghi et al., 2001).
- Failure to tell a dynamic, internally consistent story. Scenarios should be movies, not still frames. Each scenario should be a smooth narrative that makes intuitive sense to the reader. The main aspects of the future should be internally consistent; the outcomes assumed for the two key uncertainties should be able to coexist; and the actions of all team members should be compatible with their interests (Maack, 2002).
- Lack of diversity of inputs. If the scenario team members are of homogeneous educational backgrounds and institutional affiliations, they will be much less likely to come up with innovative solutions. To build successful scenarios, the participation of a diverse group of people is essential.
- Large number of scenarios typically required for real-world applications can be particularly difficult to read and understand. A suitable CASE tool would allow for selecting an actor or user role and seeing the associated use cases highlighted, but in paper or static diagrams, they are usually omitted (Nasr et al., 2002).

2.6 Conclusions

As scenario-based approaches attract increasing interest among requirements engineers, the literature on scenario methods, models, and notations expands. Scenarios have become popular notably in HCI field. In this chapter, we discussed how and why scenarios have been advocated as an effective means of communicating between users and requirements engineers and anchoring requirements analysis in real world experience. We also explained how scenarios play a role in task analysis, in storyboards and prototyping, and in use cases. Effective use of scenarios depends on how the representation of the scenario accords with the purpose of the scenario analyst. Thus, the representations chosen for scenarios are crucial in promoting this understanding. In this chapter, the frequently trade-offs to be achieved in practice between the level of formality and precision that a scenario reveals is discussed.

Scenarios have been advocated as a means of improving requirements engineering for user interface design. However, the current challenge facing the HCI community is making scenario usage more effective and efficient. Unfortunately, few concrete recommendations exist about how scenario-based requirements engineering should be practised, and even less tool support is available. In next chapter, we extend a scenario-based notation; the Use Case Maps for user interface requirements.

Chapter 3

Use Case Maps: A Roadmap for Integrated Specifications of Software and its Usability

Abstract

In this Chapter, we explore use case maps and the proposed extension for user interface requirements. Our research responds to major gaps in user interface specifications in HCI, in software development methods for interactive software, and in the communication between user interface developers and software developers. First we introduce the original UCMs notations proposed by Buhr (1998). The Use Case Maps (UCMs) is a scenario-based notation for describing, in an abstract way, how the organizational structure of a complex system and the emergent behavior of the system are intertwined. It provides a first-class design model for the "how it works" aspect of both object-oriented and real time systems. Use case maps give a road-map-like view of the cause-effect paths traced through a system by scenarios or use cases (Buhr, 1998). In this chapter, we emphasize on the great need to enrich the notations to capture user interface requirements in particular for dialog, task and presentation aspects. We illustrate, via a concrete example, the usage of use case maps as an approach for specifying user interface requirements.

3.1 Introduction

In our research, we investigated the possibility of making use of the prospective standard Use Case Maps (UCMs) for the support of user interface requirements. UCMs are scenario-based notations that describe end-to-end causal scenarios at any level of abstraction, allowing the behavior of complex systems to be described with ease (Buhr, 1998). Details of inter-component communication are considered lower level detail that can be determined at a later stage. UCMs are intended to be useful for different software development areas such as requirement specification, design, testing, maintenance, adaptation, and evolution. Already UCMs are used in some of these areas. Buhr (1998) summarize the main properties of UCMs as:

- UCMs describe, in abstract way, how the organizational structure of a complex system and the emergent behavior of the system are intertwined.
- UCMs aim to leverage human understanding of the big picture during all phases of the lifecycle, not just to specify scenario sequence by combining a set of scenarios in a single diagram, which enables designers to express scenarios and scenario interactions in a graphical manner.
- The construction of UCMs reveals problems with use cases, which may be incomplete, incorrect, ambiguous, inconsistent, or at different levels of abstraction.
- UCMs notation is lightweight enough to learn quickly and to be useful for sketching design ideas or explanations quickly in back-of-the-envelope fashion. A novel and powerful feature of the notation is the way it represents highly dynamic situations in a direct and understandable way.
- UCMs are inherently object-oriented because they are concerned with systems of collaborating objects.
- Finally, UCMs notation is gaining popularity and notoriety. Several studies are focusing on how to integrate UCMs to UML and how to formalize UCMs in XML (Amyot, 1999).

In this chapter, we focus on the requirement and analysis phase to specify user interfaces specifications. The requirements concern an *external* view of the interface described in the language of the users/customer, while the analysis concerns an *internal* view of the system described in the language of the developers. In Section

3.2, we study the original UCM notations and consider the strengths and weakness of UCMs. Also in Section 3.2, we discuss the vital to enhance the notation for user interface requirements. In Section 3.3, we present a new proposal notations specifically adapted to develop interactive systems. Section 3.4 outlines the semantics of the UCMs extensions. In Section 3.5 we present a worked example of our approach and, finally, in Section 3.6 we discuss the conclusions and further developments.

3.2 Use Case Maps Original Notations

Use Case Maps (UCMs) notation was invented by Buhr and his co-workers at Carleton University, to capture designer intentions while reasoning about concurrency and partitioning of a system, in the earliest stages of design. A UCM is a visual notation to be used by humans to understand the behavior of a system at a high level of abstraction (Buhr, 1998). It is a *scenario-based* approach intended to explicate cause-effect relationships by traveling over paths through a system. The basic UCM notation is very simple, and consists of three basic elements: responsibilities, paths and components. A simple UCM exemplifying the basic elements is shown in Figure 3.1. *Responsibilities* are generic and denoted by crosses, representing actions, activities, operations, tasks, etc. *Components* are also generic and they represent software entities (objects, processes, databases, servers, etc.) as well as non-software entities (e.g. actors or hardware). The relationships are said to be *causal* because they link causes (filled circles, representing pre-conditions or triggering causes) to effects (bars, representing post-conditions and resulting events) by arranging responsibilities in sequence, as alternatives, or concurrently. Essentially, UCMs show related use cases in a map-like diagram, whereas UCM *paths* show the progression of a scenario along a use case. As shown in Figure 3.1, a UCM is intended to be intuitive, and high-level. Details can be represented, but that is not the purpose.

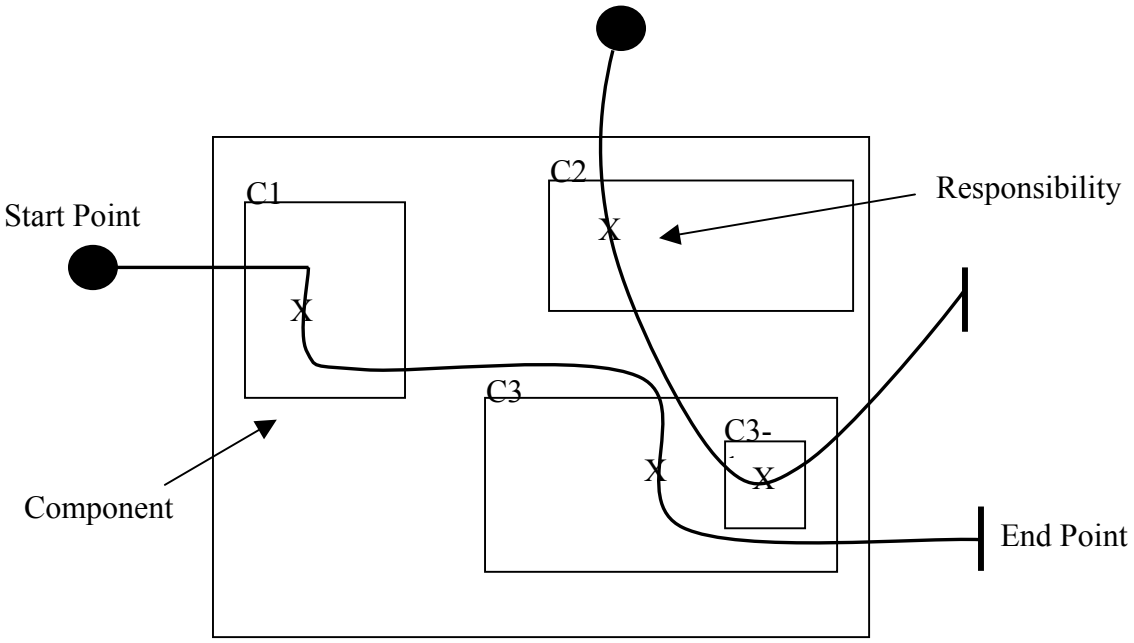


Figure 3.1 Basic UCM notations

UCMs can be hierarchical. Top-level UCMs are called *root maps*. All levels of UCMs can include some containers (called stubs) for sub-maps (called plug-ins) (Amyot, 1999). Plug-ins can be used and reused in appropriate stubs. A map including a stub is called the ***parent map*** of the plug-ins that can be contained in this stub. There are two kinds of stubs (Amyot, 1999).

– **Static stubs**: represented as plain diamonds. They can contain only one plug-in

– **Dynamic stubs**: represented as dashed diamonds. They can contain several plug-ins, whose selection is determined at run-time according to a selection policy.

UCMs can represent systems at different levels of abstraction (Buhr, 1996). A UCM can describe features of a system in general terms at a very early stage even when the architecture of the system is unclear. At such a stage, UCMs are called Unbound UCMs since no components are defined. Unbound UCMs are very useful in the first stage description of service functionalities, which focuses on causality and responsibilities without reference to architecture or components (Amyot, 1999). Basic UCM notation elements, including start point, end point, responsibility and component, have been shown in Figure 3.1. Other UCM path elements are presented in Table 3.1.

Compared to the Unified Modeling Language (UML), UCMs fit in between Use Cases and UML behavioural diagrams. Use cases often provide a *black-box* view where the system is described according to its external behavior. UML Class

Diagrams have a *glass-box* view, they are used to describe how a system is constructed, but do not describe how it works; this task is taken up by UCM's. Thus, UCM view represents a useful piece of the puzzle that helps bridge the gap between requirements and design. UCMs can provide a *gray-box view*, a traceable progression from functional requirements to detailed views based on states, components and interactions, while at the same time combining behavior and structure in an explicit and visual way (Amyot and Mussbacher, 2001).

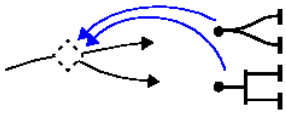
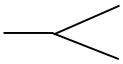
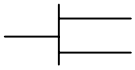
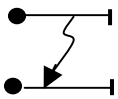
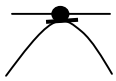
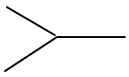
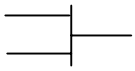
Name	Notation	Description
Static Stub		Static stub contains only one plug-in (sub UCM), hence enabling hierarchical decomposition of complex maps.
Dynamic Stub		Dynamic stubs may contain several plug-ins, whose selection can be determined at run-time according to a *selection policy* (often described with pre-conditions).
OR-Fork		A single path segment splits into several alternative path segments.
And-Fork		A single path segment splits into several concurrent path segments.
Abort		Top path aborts bottom path.
Waiting Place		Top path waits a triggering event at some point for carrying on the path.
OR-Join		Several alternative path segments join in to a single path.
And-Join		Several parallel or concurrent path segments synchronize into a single path.
Timer	 Timeout path	If a timeout occurs, the path goes on the timeout path.
Loop		Part of path can be repeated before carrying on the rest of the part.

Table 3.1 Basic UCM path elements.

3.2.1 Why Consider UCMs for User Interface Requirements?

One important goal of user interface requirements engineering is to ensure that a consistent and feasible requirements specification can be developed, while ensuring that the specification is a valid reflection of user requirements. To achieve this goal, we considered the possibility of making use of the prospective standard Use Case Maps (UCMs) for the support of user interface requirements.

Based on previous studies (Buhr, 1998; Amyot and Mussbacher, 2001), we believe that the UCMs notation is easily understandable by both the user (customer) and the software developer. In fact, this helps user interface designers to handle different users' understanding and expectation of the interface, and bridge the gaps by refining the requirements earlier. If the software developer is using UCMs, this will support the idea of concurrent engineering where a whole overview of the system (functionality and user interface) can be presented using one abstract model. Consequently, this improves the probability that software and the interface will be correct when it is finally put together, since contradictions in the requirement can be captured in early stages, which shortens the overall time required and makes it correct. It will also help to reduce the chances of introducing errors when changes are made. Furthermore, many studies are focusing on how to integrate UCMs to UML and how to formalize UCMs in XML (Amyot and Mussbacher, 2001). Finally, a major strength in the extended UCMs is its ability to capture most of the user interface requirements as shown in the next section.

On the other hand, the support of UCMs for designing user interactive systems is still acknowledged to be insufficient. The current UCMs notation does not support *presentational* aspects of the interface. Also, it is very weak in describing *dialog* modeling between the user and the system. We aim to enrich UCMs for user interface requirements by presenting new notations that assist to detain the complete user interface requirements. The enriched UCMs should focus on three dimensions the *task*, *dialog*, and *presentation*, of the user interface structure.

3.2.2 User Interfaces Requirements- Three Dimensions

User interface requirements is a knowledge base that explicitly answers the following questions: Who are the users of the interface? What tasks do the users perform using the interface? How does user communicate with the interface? How are the interface

components presented to each user? What commands and actions can the user perform on the interface?. In this chapter, we extended the UCMs to accommodate three dimensions of the user interface to answer the above questions.

3.2.2.1 Task Dimension

This dimension describes the tasks that users are able to perform using the application, as well as how the tasks are related to each other. A user task has a goal associated with it, it may include number of subtasks, which are executed according to a particular procedure, and it may have to satisfy a number of conditions before it can be completed by the user. The user task definitions are applied during the interface generation process to derive the interface dialog. UCMs are used as a simple and expressive visual notation that allows describing task scenarios at an abstract level in terms of sequences of responsibilities and tasks over a set of components.

3.2.2.2 Dialog Dimension

In the dialog dimension all communication between the end-user and the system takes place. It specifies the user commands, interaction techniques (e.g., keyboard techniques, mouse-based techniques, pen-based techniques, voice-based techniques), and interface responses and command sequences that the interface allows during user sessions. The dialog specifications can be derived in good part from the user-task.

Consequently, the dialog dimension is intended to explain the style of dialog interaction techniques such as: the dialog type and techniques (e.g., alphanumeric techniques, form filling, menu selection, icons and direct manipulation, generic functions, natural language), explain the navigation and the orientation in dialogs, error management, and illustrate the multi-person dialogs

3.2.2.3 Presentational Dimension

The presentational dimension provides a conceptual description of the structure and behavior of the visual parts of the user interface. There the interface is described in terms abstract objects. The presentation dimension is a view of the static characteristics of an interface, mainly its layout, organization. It identifies the objects comprising the user interface, their grouping and specifies their layout, e.g. by indicating approximate placement or by indicating topological relations between groups. This dimension represents the space within the user interface of a system

where the user interacts with all the functions, containers, and information needed for carrying out some particular task or set of interrelated tasks. Moreover, successive displays of different screens and interactive objects are presented.

3.2.3 UCMs Extensions for UI Requirements (UCM-UI)

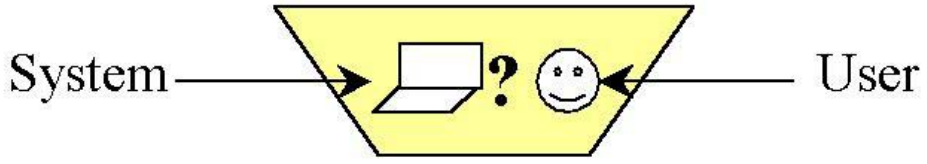
The UCM core notation has been extended over the years to cover different fields of applications such as performance analysis at the requirements level (with timestamps) and agent-oriented design (Amyot, 2000). Due to the inherent complexity and scale of emerging user interface features, special attention has to be brought to the early stages of the design process. The focus should be on understanding the overall behavior of the interface rather than on details belonging to a lower level of abstraction, or to later stages in this process. Many user interface designers and usability experts recognize the need to improve such process in order to cope with the new realities cited above.

In our research, we focused on what notations may be developed to complement UCMs in capturing user interface requirements in the early stages of design where very little design detail is available. In this section, new notations are suggested to support the UCMs for user interface requirement. The extended notations are expressive, compact, understandable and flexible notations representing concurrent and interactive activities by the user and the interface. The extended notations provide UCMs with more expressive power for interactive systems. Even people without formal background easily interpret the new notations.

3.2.3.1 Dialog Notations

UCMs notations are week in representing dialogs, which are a central aspect in user interface requirements. Thus, we introduced a new simple notation for dialog that explains the initiator of interactions between the user and the system. In Figure 3.2 the system (on the left side) is the initiator of a question-and-answer dialog. In reality, richer styles of communications between the user and the interface have now become commonplace. Common dialog styles include: natural language, command language, query language, questions/answers, function keys, menu selection, form filling, direct manipulation, iconic interaction...etc. These types of communications are presented in Figure 3.3.

Figure 3.2 UCM-UI notations for a question-and-answer dialog.



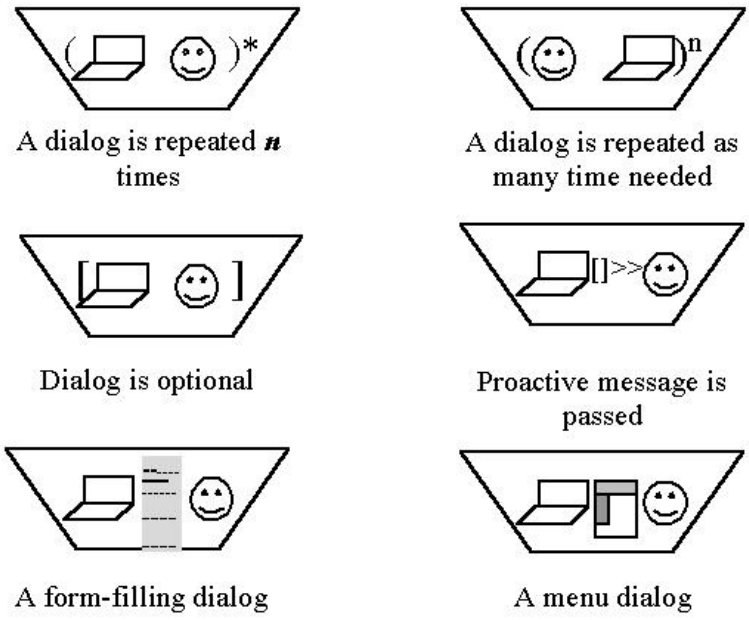


Figure 3.3 UCM-UI dialog notations.

3.2.3.2 Presentational Notations

Once the appropriate dialog style has been selected and the task analysis has been completed, we are ready to build the user interface presentation. The presentation of the user interface is recursively structured into *Presentation Units* (*PU*). By definition, a *PU* consists of an input/display world decomposed into windows (not all necessarily present at the same time) in order to perform a sub-task of the interactive task by satisfying the human requirements of the user. Common presentation units include: menu windows, form filling window, spreadsheets window, multimedia windows...etc. These types of *PU* are presented in Figure 3.4.

3.2.3.3 Tasks Notations

Tasks or actions in UCMs are responsibilities represented by a cross. To express temporal relationships between tasks, we pass a variable with the responsibility. These temporal relations are: undo, iteration, choice, exception, repetition. The temporal relationships in the enriched UCMs for user interfaces are described bellow:

– Iteration R(*) – denotes the responsibility (R) is performed repeatedly until the task is deactivated by another task;

– Finite Iteration(s) R(n) – same as iteration but the responsibility (R) is performed n times;

– Optional responsibility ([R]) – denotes that the performance of a responsibility is optional.

– Undo responsibility (R (↺))- denotes that the responsibility (R) effect may be reversed.

In short, our proposal for the extended UCM notations for user interface (UCM-UI) included a new UCM interaction model to accommodate the three dimensions. Therefore, the user interface model encompasses the task, dialog and presentation dimensions, clearly mapping the conceptual architectural models for interactive systems, while maintaining the desired separation of concerns.

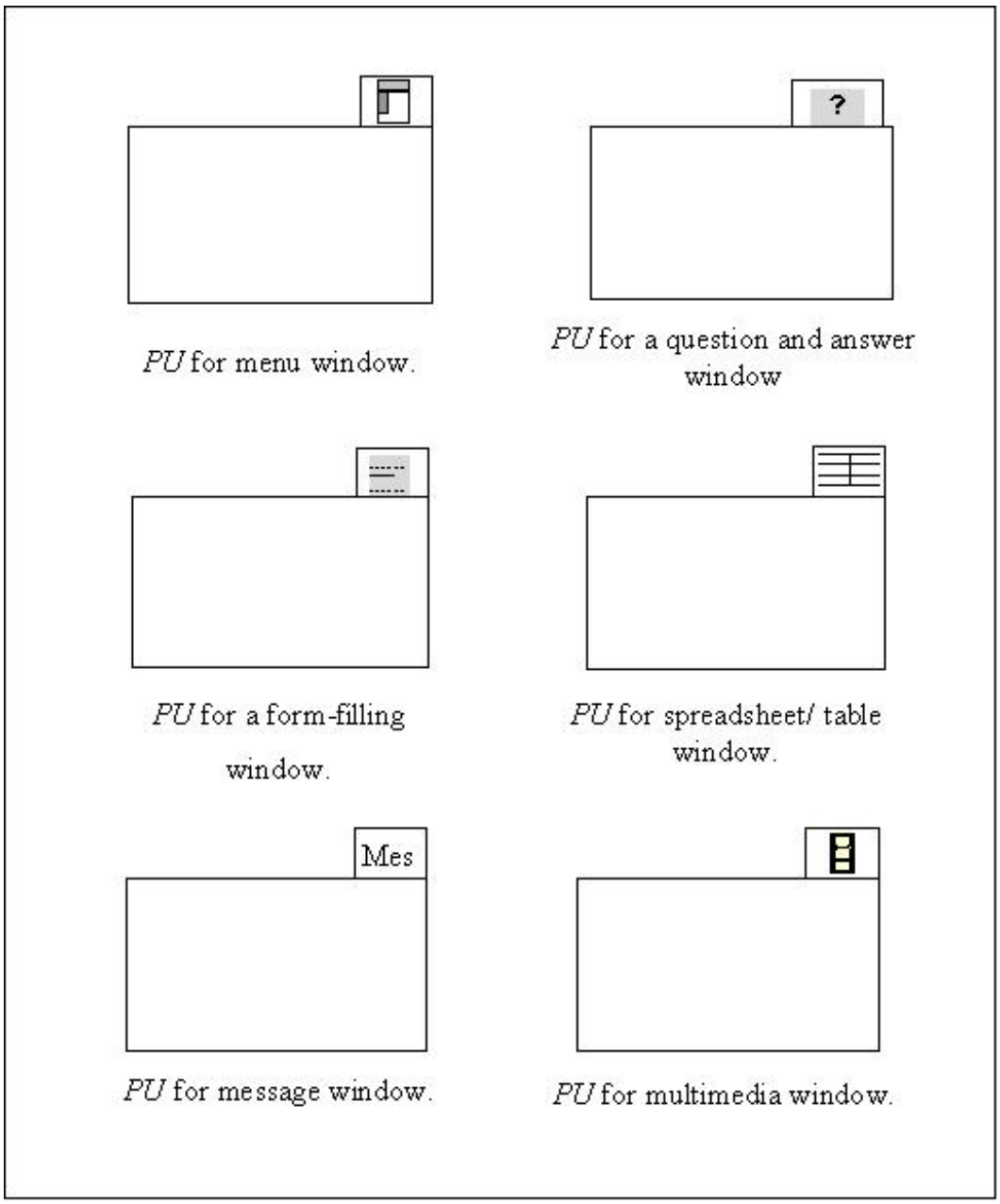


Figure 3.4 UCM-UI notations for the presentation of the user interface.

3.3 AN ILLUSTRATIVE EXAMPLE

Filter Agent (FA) aims to assist the user with email management using a memory based learning technique. The interface agent (FA) learns by continuously "looking over the shoulder" of the user as the user is performing actions. The interface agent monitors the actions of the user over long periods of time, finds recurrent patterns and offers to automate them. If the user drags and drops a particular electronic mail message to a specific folder, the mail agent adds a description of this situation and the action taken by the user to its memory of examples. The agent keeps track of the sender and receiver of a message, the Cc: list, the keywords in the Subject: line, whether the message has been read or not, whether it is a reply to a previous message, and so on. If a new mail arrives, the agent predicts which action is appropriate for the current situation and also measures its confidence in each prediction by determining how many examples the agent has memorized. If the agent is confident, then it will autonomously take the action on behalf of the user. Otherwise, the agent either offers suggestion to the user and wait for the user's confirmation to automate the action, or wait and observe the user action.

3.3.1 Task and Dialog Dimension

The behavior of the filter agent and consequently the relationship among the functions mentioned earlier are better understood by following UCMs flows shown in Figure 3.5 Based on the root map in Figure 3.5, users and designers can consider early decisions regarding the sequence in which functions are performed. This map describes the system behavior that starts when a pre-condition is satisfied, for example, new email arrives (filled circle labeled **MA**).

A dialog illustrates that a conversation between the user and the agent is taking place where details are delayed to a sub-UCMs. The dialog notation is applied to our work not only to hide details, but also to decompose the system into small manageable units. This scenario ends when one or two of the following triggering events occur: user exits the email application (bar labeled **E1**) or an email is filed (bar labeled **E2**). These events are represented by post-conditions in the UCMs. A route is a path that links an initial cause to a final effect. For example, <**MA, [a2]**, **[a4]**, **D**, **E2**> represents a route that starts when new email arrives (AR), user reads the mail

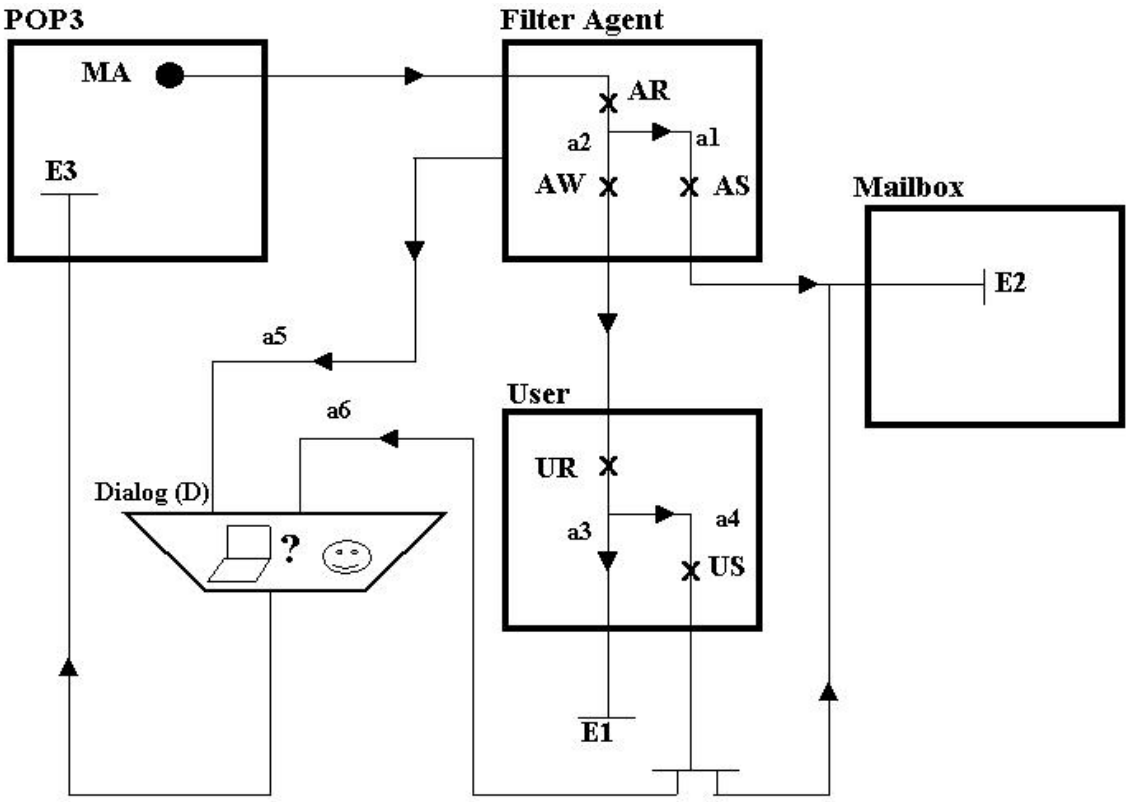
(UR) and sends it to a specific folder (US), followed by a dialog (D) between the user and the agent.

Figure 3.6 depicts the second level of the requirement model when a conversation or a dialog goes on between the agent and the user. The agent offers it suggestion to the user waiting for a confirmation. The user either confirms the suggestion or denies it. Alternative paths (called OR-forks) represent composite UCMs that can be split into two different paths (no level of concurrency is associated with them). For instance, a responsibility point (cross-labelled **AA** in the figure) is activated along the **[b1]** path to decide whether the user confirms the suggestion or deny it. The alternative sub-paths (labelled **[b2]** and **[b3]**) are generated after this suggestion.

3.3.2 Task and Presentation Dimension

The enriched UCMs not only describe the sequence of tasks and dialogs that can take place between the user and the system, but also help to understand and reason about the requirements of the user interface, including usability aspects.

Figure 3.7 shows the user interface structure of the email application that consists of a tool bar, folder menu, mailbox, and an agent window. Figure 3.7 describes the approximate position of the components of the user interface and the user/system behavior that starts when a user opens the email application. The user downloads new arrived mails using the tool bar menu. Then the user continuously reads a mail then folders it until either the agent initiates a dialog or the user quits the application.



Legend
MA: New email arrives
AR: Agent reads the email
[a1]: Agent is sure where to folder the mail
[a2] : Agent is not sure where to folder the mail.
AS: Agent sends the mail to a specific folder
AW: Agent waits and observes the user behavior.
UR: User reads the mail
[a3]: User does nothing with the mail
[a4]: User decides where to send the mail.
US: User sends the mail to a specific folder.
[a5]: Agent initiates a question and answer dialog
[a6]: User participates in the dialog
E3: Wait for new mail arrival.

Figure 3.5 Task and dialog UCM for Filter Agent.

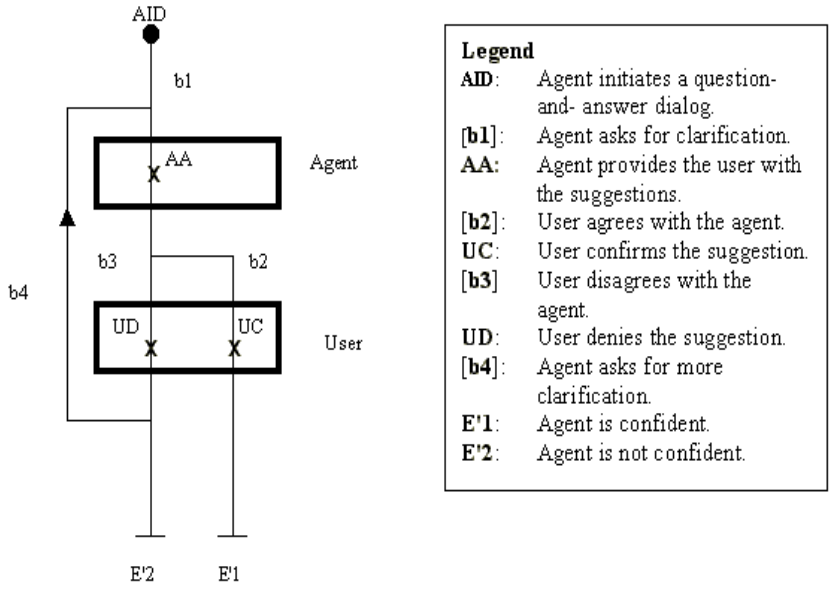
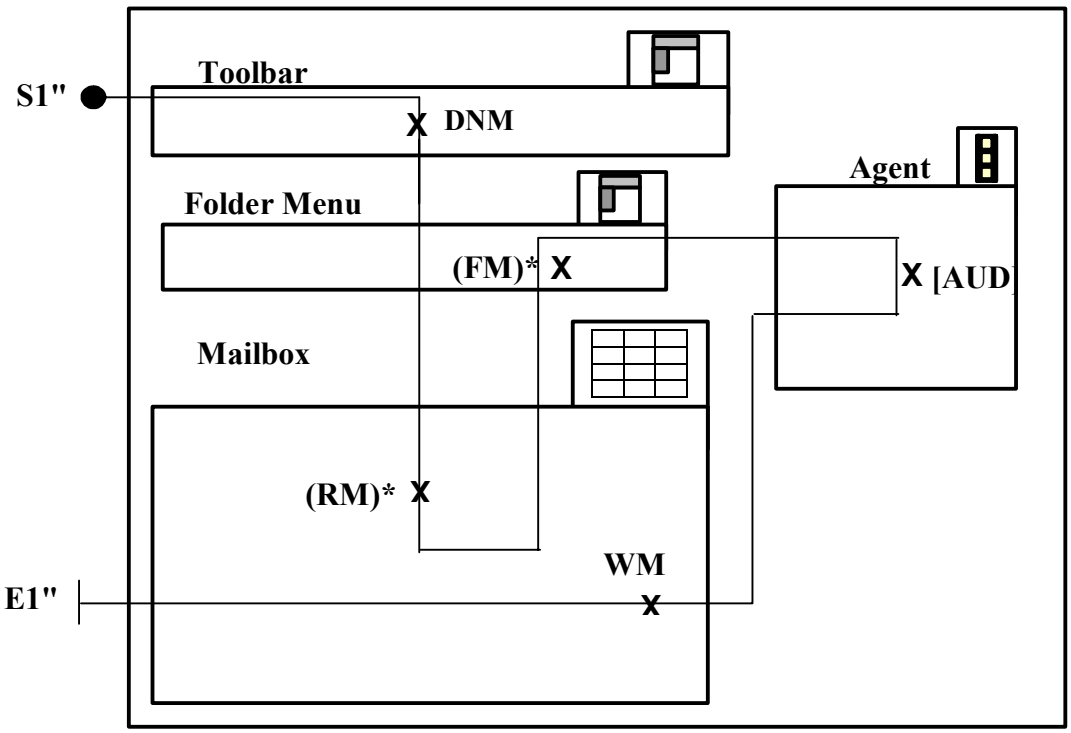


Figure 3.6 UCM explaining the dialog between the agent and the user.



1. **S1":** User open the application
2. **DNM:** user download email using toolbar
3. **(RM-FM)*:** user read then file the mail
4. **[AUD]:** agent initiate a dialog with the user (optional)
5. **WM:** user continue to work with the mail
6. **E1":** user exist the application

Figure 3.7 Task and presentation UCM for Filter Agent.

3.4 Conclusions

The most valuable contribution of this chapter is the development and analysis of a complete UCMs notation for high-level requirements of user interfaces. Our extended UCM-UI notations put together three dimensions of user interface requirements including the task, the dialog, and the structure of the user interface. In this manner, functions and tasks are distributed over the use case map defining how the users will be permitted to move among various tasks and how and when the user will move from one object to another in the interface. Moreover, the enriched UCM-UI captures the initiator and style of interactions between the user and the system. The improved UCMs have also the potential to capture a complete picture of user interface requirements by describing how a system will be used and to what ends.

Chapter 4

New Scenario and Use Case-Based Requirements Engineering Framework

Abstract

In this chapter, we propose a new framework for eliciting and specifying user interface and usability requirements. Among the popular and potential applications, scenarios and use cases provide an understanding of the whole process of the user activities. Our goal is to build a complete and consistent user interface requirement framework that is simple, intuitive, unambiguous and verifiable with the help of the extended UCM-UI notations to better suit interactive systems, and by providing step-by-step guidance for the employment of use cases.

4.1 Introduction

Scenarios have been used in both HCI and software engineering, sometimes with different meanings (Benner et al. 1993; Rolland et al. 1998; van Lamsweerde 2000a; Carroll 2002; Sutcliffe 2003). Although there is no widely accepted definition of the terms 'scenario', many software engineers consider scenarios as instances of use cases, while many others use them as interchangeable. The main purpose of developing scenarios in requirements engineering is to stimulate thinking about possible occurrences, assumptions relating these occurrences, possible opportunities and risks, and courses of action.

An important question when applying use case modeling for user interfaces is: What criteria should we have for creating and describing use cases for user interfaces? From an HCI viewpoint, dealing with scenarios and use cases – narrative, rich and non-formal descriptions is not considered a choice but forced on research by practice (Jarke 1998; Carroll 2002; Hertzum 2003). HCI as well as software engineers need to understand and document:

– Who and what is involved in each use case;

– What the use case is trying to achieve, and why;

– How often the use case occurs;

– What are the activates carried out by each participant in each use case;

– How activities depend on each other;

– What the requirements are for each activity and how they relate to the overall goals of the use case.

These questions must be detailed enough for user interface designers and usability engineers to design the activity as an isolated unit.

Requirements experts in software engineering and HCI, on the other hand, are faced with problems such as: How do we deal with collections of scenarios (i.e. collections of only weakly structured text)? How do we deal with coverage (writing a comprehensive set of use cases)? What are the boundary conditions for the applicability of scenario-based design? How much the details provided by a scenario or a set of scenarios are essential, and what is inconsequential? Scenarios can

represent different levels of task details and determining the level such detail remains a challenge for developers. Moreover, for large software systems, there may be a very large number of scenarios. The next section is an attempt to solve such problems using the proposed SUCRE framework.

This chapter is organized as follows: Section 4.2 introduces Scenario and Use Case Requirements Engineering (SUCRE, in English Sugar) as a novel framework that makes use of the extended UCM-UI notations (Chapter 3); to bridge the gap between users' needs and requirements. We present the general idea, principle and concepts of SUCRE frame work. Example on how to build a use case model according to the SUCRE framework is described in Section 4.4. Section 4.5 concludes this chapter.

4.2 SUCRE Architecture

The novel SUCRE framework divides the process of transforming requirements statements in a natural language to formal specifications through a number of iterating phases. Figure 4.1 depicts the main components of the SUCRE framework we are proposing. The framework starts with the informal description of users' needs and requirements. At the end of each phase in the framework the formality of the requirements representation has been increased by a small step. The process includes three typical phases:

– *Scenario Analysis phase*: Identify a set of scenarios, i.e. a set of situations having common characteristics. Work on each scenario separately to elicit information. Complicated problems are decomposed systematically and naturally. Information from different sources can be elicited independently. A requirements definition and use cases can be created from the descriptions of the scenarios.

– *UCM-model construction phase:* From the use cases we build two types of maps using our extended UCM notations, namely, a Conceptual Use Case Map (CUCM) and a Physical Use Case Map (PUCM). These two maps capture a comprehensive picture of user interface and usability requirements. The CUCM and PUCM visually integrate behavior and structural components in a single view. This helps to analyze the level of consistency between requirements of different use cases; it helps to resolve conflicts between different types of users, and different use purposes and different operation conditions can be discovered. CUCM and PUCM analysis furthermore provides a basis for requirements validation. By analyzing the consistency, completeness and precision of the requirements definition with respect to a set of well-defined use cases, the requirements definition can be validated and verified.

– *Formal validation phase:* During this phase any informal and undefined terminology used in the frameworks, are defined formally and specified in first order logic or some other appropriate mathematical notation. These formal definitions constitute the knowledge base of system-specific knowledge and domain knowledge. Formal requirements specifications are produced according to the frameworks of the users' requirements and formally defined system specific knowledge as well as domain knowledge.

In our proposed process of developing SUCRE framework, the engineers repeat the cycles of inputting information, checking consistency and completeness of the information and making modifications to resolve conflicts and cut down on incompleteness. Information is presented in different forms during the different stages of requirements elicitation, analysis and specification. Once the use case models have been constructed and validated, further requirements analysis and specification activities can start to produce formal requirements specifications according to the framework of the users' requirements. These activities are discussed in Chapter 5 and Chapter 7.

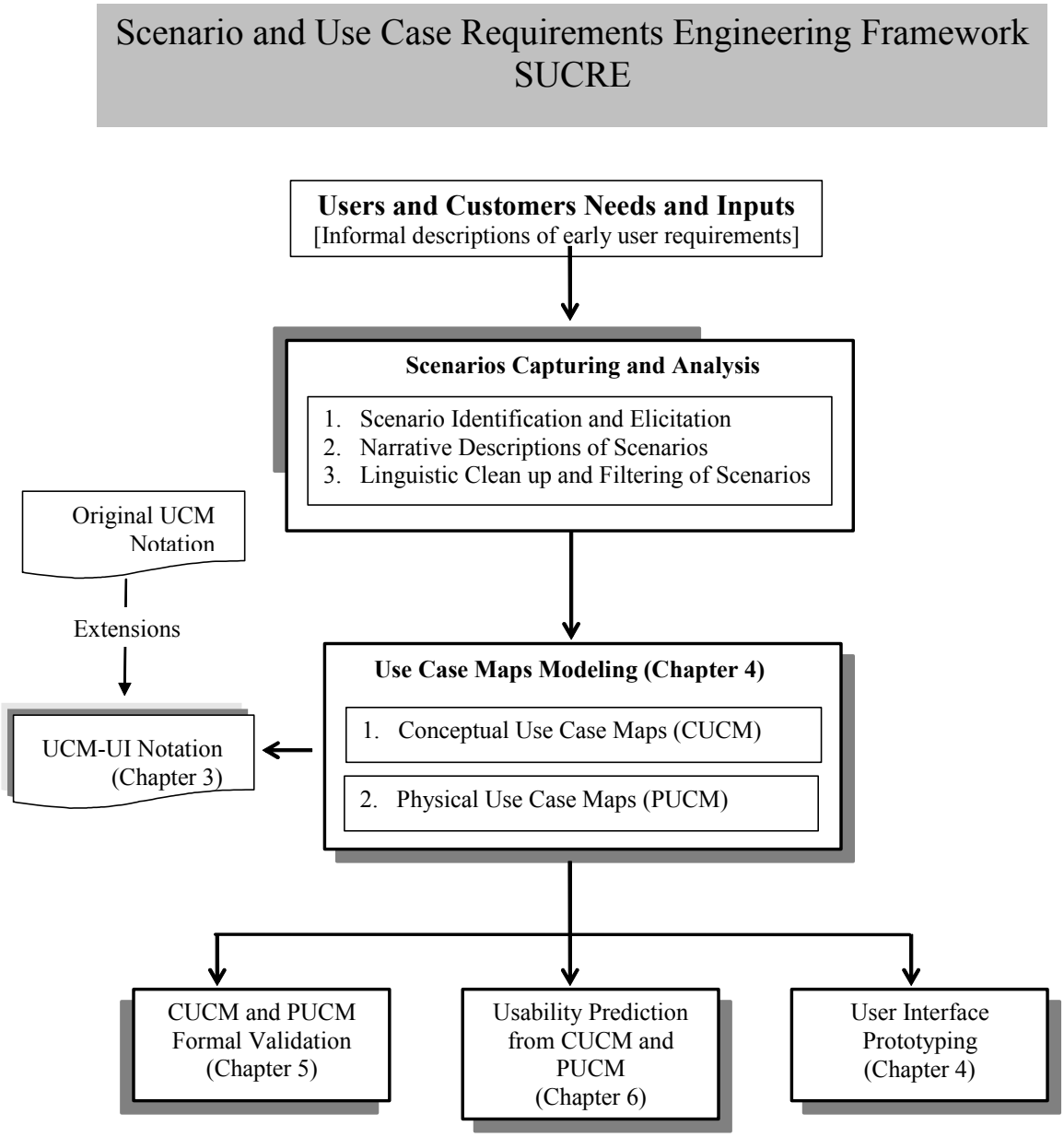


Figure 4.1 SCURE Frame work.

4.2.1 Scenario Analysis

Scenario analysis is a user-centred process that can play an important role in requirements engineering. At this stage, only very limited information is available owing to the elicitation natural to the stage. A process model of scenario analysis consists of the following interactive and iterating activities:

– *Identification of scenarios:* a set of scenarios is identified by analyzing the end-users, use purposes and operational conditions in the uses of the system. A number of

approaches to the identification of scenarios can be used such as interviews, focus group, essential use cases, and use case storyboards.

– *Elicitation of information and description of scenarios:* for each scenario, the requirements in the scenario are elicited and described so that further analysis of scenarios can be conducted on the basis of the description.

– *Build use cases:* use cases are created where each use case is a collection of related scenarios between the system and end-users, characterized by the goals the primary-user has showing how this goal might be delivered or might fail.

4.2.2 UCM-UI Model Construction

From the use cases we build two maps, the Conceptual Use Case Map (CUCM) and the Physical Use Case Map (PUCM). The two maps capture a complete picture of user interface and usability requirements.

4.2.2.1 Conceptual Use Case Map (CUCM)

The CUCM reflects the needs of the system and the user in their interaction with one another. It represents tasks that are relevant for interactions by describing task scenarios at an abstract level in terms of sequences of responsibilities and tasks over a set of components. Tasks can be split up into subtasks (actions, operations) or inherited from 'super'-tasks. CUCM is also intended to explain style of human-computer interaction while describing the sequence and type of dialogues that can take place between the user and the system. The proposed model help provide a logical order to the conceptual design and provide overview on what information will be needed from the user and the system to accomplish a task. Steps to create Conceptual Use Case Map (CUCM):

– *Partition the use case model:* Consider only the use cases with human actors.

– *Decompose use case into Tasks/Subtasks:* This help in determining the minimal information necessary at each decision point; at a decision point, an actor must be presented with the information necessary to make the decision. It will also help to determine the minimal content as well as the sequencing.

– *Determine Information Content:* Tasks represent decision points in the use case. At a decision point, an actor must be presented with the information necessary to make the

decision. This information is in the task itself or in prior tasks. After the decision is made, information acknowledging the decision is often returned to the actor.

– *Create Dialog Model:* three areas are considered.

  - Information presented to the user in a previous/upper level task and required by the user in the activities associated with the current task.
  - Information elements delivered as part of this task.
  - User responds to system request. Information returned to the system.

4.2.2.2 Physical Use Case Map

Once a complete understanding of the CUCM is developed, the physical user interface layout may start. The PUCM can greatly benefit from the graphical representation of use cases. PUCM deals with the selection of actual user interface components (user interface structure/layout). It represents the space within the user interface of a system where the user interacts with all the functions, containers, and information needed for carrying out some particular task or set of interrelated tasks. Moreover, successive display of different screens and interactive objects are presented. Before starting physical model, it is important to understand what information is needed by the user and the system, and at what time. This idea is the essence of the conceptual model. Steps to create PUCM:

– *Identifies user interface objects and components:* indicate the approximate placement or topological relations between group of objects and components constructing the user interface. Also, represent the space within the user interface of a system where the user interacts with all the functions, containers, and information needed for carrying out some particular task or set of interrelated tasks. This step can benefit from the graphical representation of use cases.

– *Establish logical screen order:* the order of the screens will follow the order of the tasks themselves.

– *Convert the use case model into prototype;* using a graphical user interface development tool. At this stage, usability experts and customers may examine the prototype to clarify their needs and explore alternative designs for satisfying them.

4.2.3 Requirements Validation

To validate the UCM-UI model, there are two major approaches: *experimental* evaluation, or *theoretical* evaluation. The experimental evaluation involves building a tool as a starting point for demonstrating and testing use cases. These are then exposed to expert critiques or lab experiments, prior to their use in industrial case studies which either construct industrial prototypes for further development into the commercial arena, facilitate/ monitor ongoing specific projects, or try a rational reconstruction of a past process (Zhu and Jin, 2000).

The theoretical evaluation approach investigates use case models independently of support tools. Often, the research claim, or a deeper theory underlying it, is elaborated into checklists which can directly be applied to laboratory experiments or industrial case studies, without necessarily going through a mediating tool (Jarke et al., 1998).

In both approaches, valuable insights can be drawn from comparative evaluation with competing claims, tools, or checklists. However, this is difficult due to the complexity of problems addressed by use cases models, and there have been few such studies to date. More often than it should, validation has therefore been restricted to the conceptual level. In Chapter 5, new operators are defined for the SUCRE framework to verify and validate the consistency, completeness and precision of the requirement model. The operators are used to validate the UCM-UI model is validated against the initial identified scenarios by analyzing whether the model is consistent with respect to scenarios. If the use cases are inconsistent, backtrack to the information elicitation and scenario description to resolve the conflicts. The adequacy of such validation is also analyzed, for example, by checking the completeness of the scenarios and use cases with respect to the requirements models. If incompleteness is discovered, backtrack to the scenario identification to identify new scenarios and use cases that cover the missing situation.

4.2.4 Predicting Usability of UCM-UI Model

Traditional approaches to measuring software usability have centered on testing. However, because testing depends on having something to test software, prototype, or simulation it is difficult to conduct testing early in the design process. Ideally, designers would like to be able to predict usability based on metrics computed from visual

designs or paper prototypes. In Chapter 6, a pragmatic suite of metrics; the UCM-UI Metrics Suite; has been developed to predict usability from scenarios documented as use case maps.

4.3 An Illustrative Example: Library System

In this section, a library system example is used to identify user interface modeling problems. The Library System (LB) in the case study could be considered too simple to catch real problems faced during the modeling of user interfaces. There are two types of users for the LB system: Librarians and Borrowers. The LB system must guarantee that only registered users can log into the system. Further, the system must guarantee that borrowers can only perform services associated with borrowers, and that Librarians can only perform services associated with Librarians. Librarians and borrowers are allowed to search for books, check the status of a book, and views users' library record. Moreover, only librarians are allowed to check books in and out of the systems, check users late fees, and extend due dates of borrowed books. The requirement specifications *RS* of the LB system are:

– *R1*: The LB allows borrowers to perform services associated with borrowers, and that Librarians to perform services associated with Librarians.

– *R2*: Librarians and Borrowers can search for books by author, title, year or a combination of these.

– *R3*: user can list the books borrowed from the library.

– *R4*: user can view the status of a book.

– *R5*: Librarians manage the book catalogue and the loan records. Librarians only need to inform to the Library System when books are checked into and checked out of the system to be able to manage loan records. Librarians check for late fees.

– *R6*: Librarians extends due date for a loan upon the request of the borrower.

4.3.1 Scenario Analysis

From the users' requirements we identified 6 major use cases each having a number of scenarios:

– UC_1: ConnectToSystem (3 scenarios)

- S_1: A librarian log into the system.
- S_2: A borrower log into the system.
- S_3: A user fail to log into the system

– UC_2: SearchBook. (8 scenarios)

- S_1-S_4: 4 scenarios when the user chooses a search for a book by (Author, Title, Year, or combination of these), views the list.
- S_5-S_8: 4 scenarios when the user chooses a search for a book by (Author, Title, Year, or combination of these), refine the list.

– UC_3: CheckBookStatus (1 scenario)

- S_1: user enters book title, views book status, returns to beginning of the system.

– UC_4: BorrowBook (2 scenarios)

- S_1: Librarian checks book status, check book out and updates the database.
- S_2: Librarian checks book status, book is not allowed to be borrowed, librarian refuses to loan the book.

– UC_5: ReturnBook (1 scenarios)

- S_1: Librarian checks due date for late fees, check book in and updates the database.

– UC_6: ListBooksBorrowedByUser (1 scenario)

- S_1: system verifies the user information and view a list of borrowed books.

4.3.2 UCM-model Construction

In this section we build the Conceptual Use Case Maps (CUCM) and a Physical Use Case Map (PUCM).

4.3.2.1 Building the CUCM

From the six use cases we can build the root CUCM depicted in Figure 4.2. The LB system may be decomposed into six main tasks, namely, (1) login to LB system, (2) search for a book, (3) view book status, (4) view users' record, (5) borrow a book, and (6) return a book. Each task can be further decomposed into sub-tasks that are delayed to sub-UCMs. The behavior of the LB system is better understood by following CUCM flows instead of reading the use cases above. The CUCM describes the system behavior that starts when a pre-condition is satisfied. Figure 4.3 illustrates the flow of one scenario where the user searches for a book by the author name upon which the

system presents such a list. The user then selects to refine the search and the view the status of a selected book.

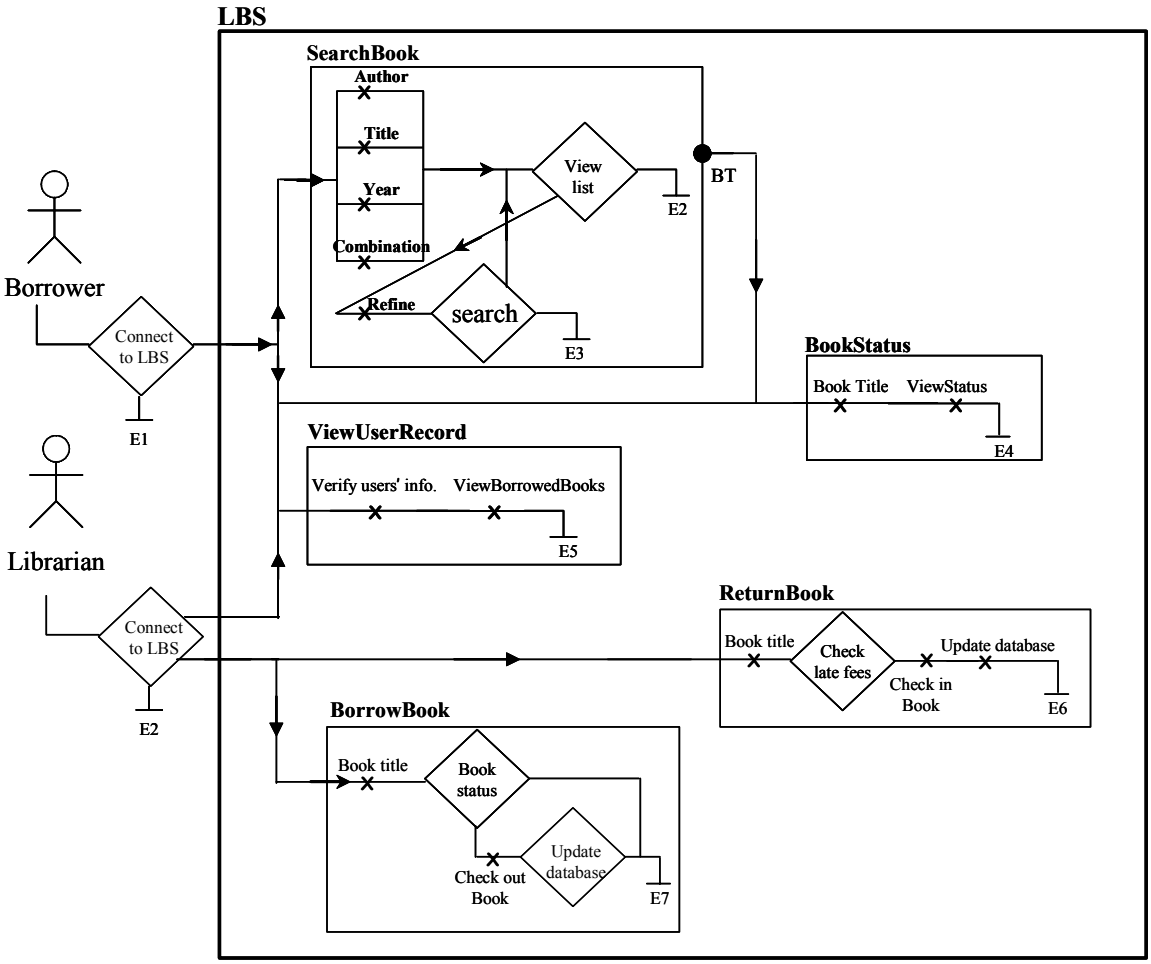


Figure 4.2 The root CUCM for the LBS software

Figure 4.4 depicts the second level of the UCM-model for the scenario of Figure 4.3. When a search for a book is selected, a menu dialog (SM) is initiated between the user and the system, the searched list is presented (VL), then the user chooses to either end the search or refine his search.

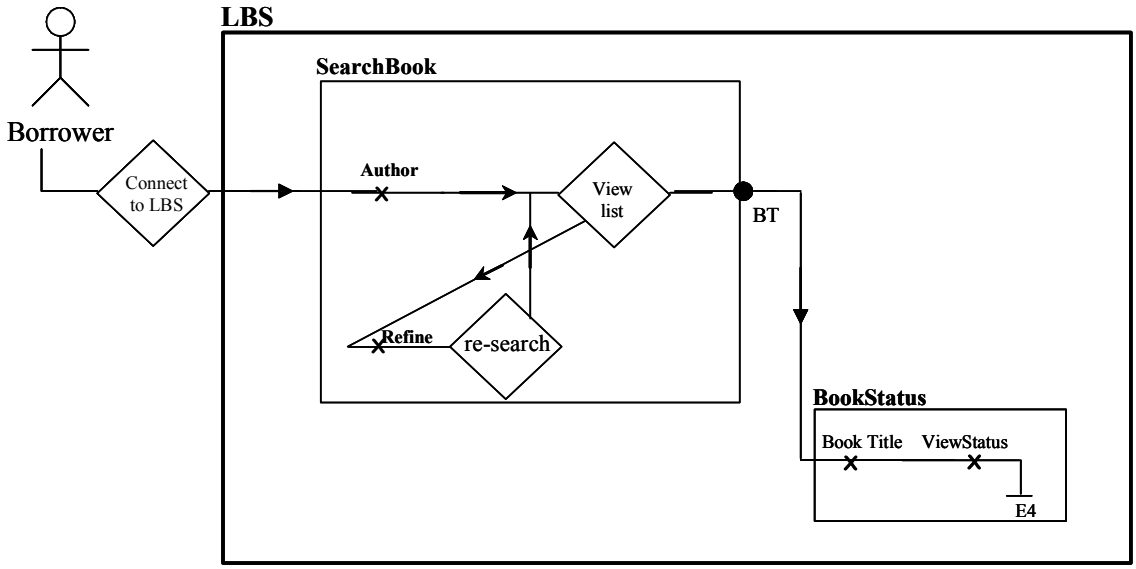


Figure 4.3 The CUCM for searching for a book by authors' name

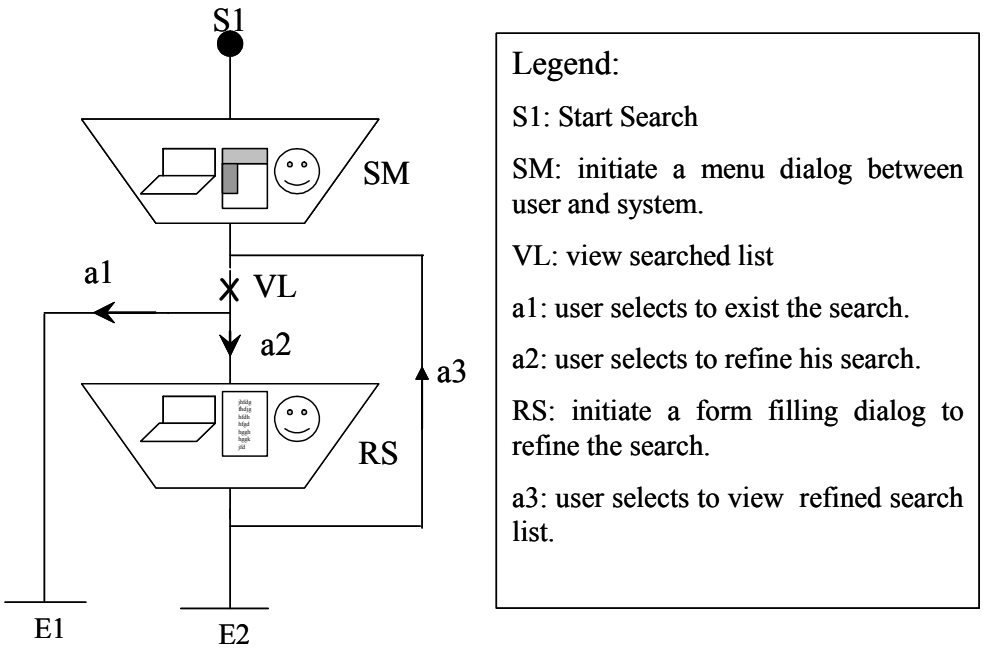


Figure 4.4 The second level of the CUCM for searching for a book..

4.3.2.2 Building the PUCM.

The PUCM of a library borrower in Figure 4.5 describes the scenario where a borrower searches for a book and refines his search. The user interface consists of two windows. The main window is for book searching, and the secondary window is to view the borrower library record. One scenario starts when the user logins' into the LB system, selects to search for a book (W1), enters Authors' name (AN) in (W2). A new window appears (W3) presenting the list of books (BL). The user refines the search (RS) in (W4). W3 would appear with the refined list. Another scenario is when the user logs into the system, selects to view his library record in the secondary window (W5).Next, the PUCM can be converted into an active prototype so that usability experts can measure the usability of the interface as shown in Figure 4.6.

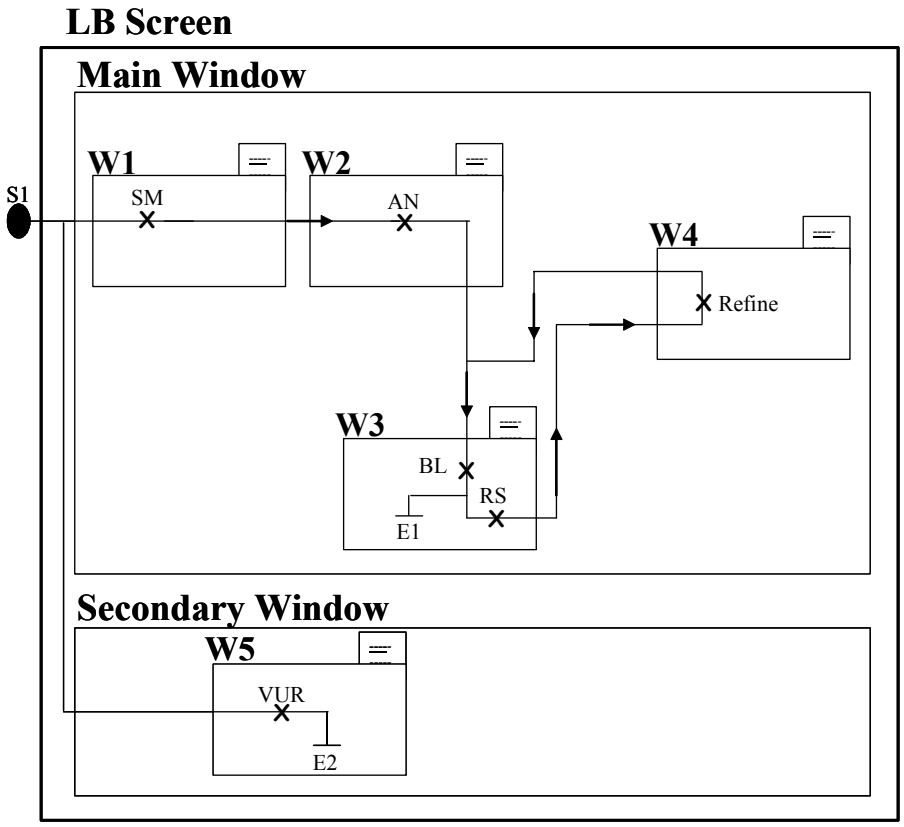


Figure 4.5 The PUCM for LB system.

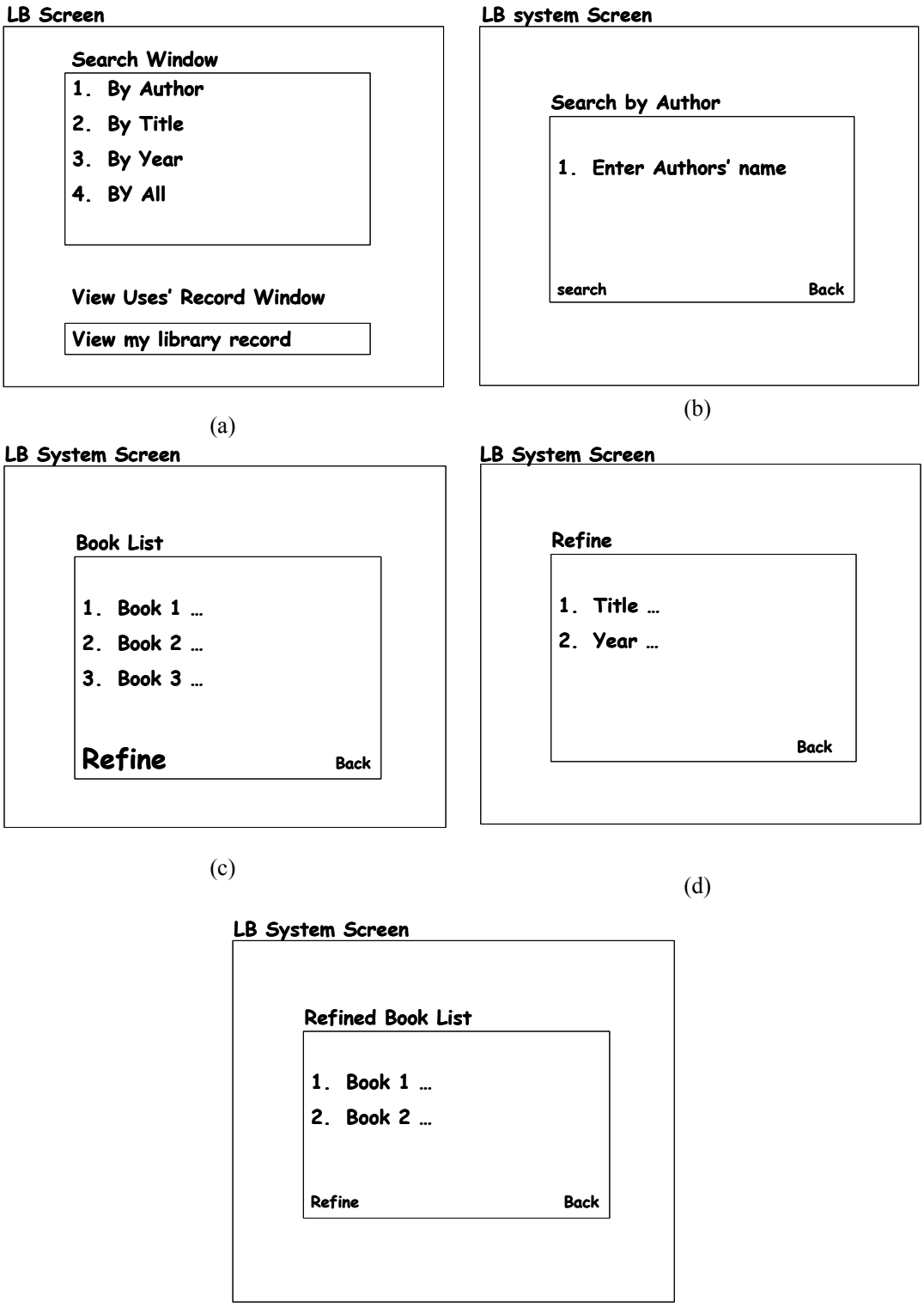


Figure 4.6 PUCM converted to a Paper Prototype.

4.4 Conclusions

This chapter proposes SUCRE; a scenario and use case based requirements engineering framework. The new framework ensures that: (1) a consistent and complete requirement specification can be captured using scenarios and use cases, (2) the specification is a valid reflection of user requirements, (3) the derivation of early design artifacts such as low fidelity prototypes is possible.

SUCRE divides the process of transforming users' requirement statements in a natural language to semi-formal specifications through a number of iterating phases. The process includes four typical phases: (1) scenario analysis phase where a requirements definition and use cases can be created from the descriptions of the scenarios, (2) UCM-UI model construction phase where two types of maps are easily constructed using UCM-UI notations, CUCM and PUCM. These two maps capture a comprehensive picture of user interface and usability requirements, (3) formally validating the UCM-UI model, and (4) early usability predictions of the UCM-UI models. This chapter provides a step-by-step guidance on how to perform the first and second phase, while the third phased will be discussed in the next chapter.

Chapter 5

Combining UCMs and Formal Methods for Representing and Checking the Validity of Scenarios as User Requirements

Abstract

In this chapter, we discuss how UCM-UI models, that enable the specification of user interface requirements and operational scenarios, with the complicity of formal requirements engineering methods, can lead to a comprehensive framework for representing and validating scenarios while improving and mediating the communication between usability engineers and software development teams. This chapter presents number of design heuristics for constructing a formal specification and demonstrates that these heuristics may be used to build operators that validate the UCM-UI model.

5.1 Introduction

The past two decades bear witness to a large amount of human-computer interaction (HCI) and software engineering research efforts in the search for appropriate requirements methods for interactive systems and in particular representations (Nuseibeh and Easterbrook, 2000; Amyot and Eberlein, 2003; Carroll, 2002; Benyon and Macaulay, 2002; Van Lamseerde, 2000).

On one hand, supporters of formal methods in software engineering have claimed to solve the problem of requirements in particular by providing unambiguous and mathematical notations and verification techniques, but the penetration of these methods in industry and in standardization bodies especially in North-America remains and among HCI practitioners unfortunately low (Ardis et al., 1996; Somé et al., 1996). In the field of HCI, formal specifications of user interfaces are generally presented to be easily communicable, mathematical, precise, unambiguous and they support the analysis, reasoning and prediction (Rouff, 1996; Duke et al., 1999).

On the other hand, as discussed in Chapter 2, scenario-driven approaches have been largely promoted as a non or semi formal vehicle to support user interface and functional requirements elicitation, analysis, and modeling (Carroll, 2002; ITU-T, 2003; Hsia et al.,1994), to bridge HCI and Software approaches (Benyon and Macaulay, 2002), and to close the current gap between requirements and design steps (Sutcliffe, 2003) while automating the generation of user interfaces prototypes (Elkoutbi et al. 1999; Harel 2001; Gervasi and Nuseibeh, 2002).

Even if scenarios-based approaches are less precise than formal methods, they are and more convivial. Their application to requirements and the early stages of the design process raises new hopes for the availability of concise, descriptive, maintainable, and consistent documents, standards, and design specifications that need to be understood by a variety of readers including, in our context end-users, stakeholders and usability professionals. Moreover, in this chapter we argue that scenarios pave the way towards the construction of detailed (formal) models and implementations through analytic and synthetic approaches. These construction approaches promise to generate models and implementations faster and at a lower cost while improving their correctness and traceability with respect to the requirements.

The layout of this chapter is as follows: Section 5.2 below provides some design heuristics for constructing a formal specification. In Section 5.3 we argue that these heuristics may be used to build operators to validate the UCM-UI model. This is followed by an example in Section 5.4. We conclude the chapter with an analysis and some pointers for future work.

5.2 Heuristics for Formal Specification, Validation and Code Generation

The process of constructing UCM-UI model may sensibly be augmented by a set of design heuristics for setting up a formal specification, e.g. Z (Spivey, 1992). In this section, we trace the development of such a suite of design principles for formal specification and show the utility of these heuristics for UCM-UI model construction. In total ten heuristics were proposed (van der Poll and Kotzé, 2002) and we found six of these principles to be useful in building a CUCM.

5.2.1 Format of a Precondition.

Consider a Z specification of the abstract state, FS, for an UNIX-like filing system (Morgan and Sufrin, 1993). We will use this state and the definition below of an operation for opening a file to develop three of our heuristics.

∪ *FS* _____

→fstore : FID ♣ FILE

→cstore : CID ♣ CHAN

→nstore : NAME ♣ FID

→*dnames* : Π *NAME* ; usedfids : Π *FID*

∩_____

→Front ⟨ dnames χ dom nstore ® ζ dnames

→usedfids = ran nstore χ {chan : ran cstore ∞ chan.fid}

→usedfids ζ dom fstore

∠_____

and CHAN is given by:

∪ *CHAN* _____

→fid : FID

→*posn* : N

∠_____

In *FS*, *FID* (a set of file identifiers), *CID* (a set of channel identifiers) and *SYL* (a set of syllables) are all basic types; *FILE* = seq *BYTE*, for *BYTE* = 0..255 and *NAME* = seq *SYL* (i.e. a sequence of syllables). A detailed discussion of *FS* appears in

Morgan and Sufrin (1993) and is beyond the scope of this paper. An operation to open a file is given by:

∪ *open* _____

→Δ*FS*

→name? : NAME; cid! : CID

→fid, fidə : FID

→report! : REPORT

∩_____

→(*name*? ε dom *nstore f cid*! ™ dom *cstore f*

→ fidə = fid = nstore name? *f*

→ (E*CHAN* ə ∞ *posn*ə = 0 *f fid*ə = fid *f*

→ *cstore*ə = *cstore* ± {*cid*! □ *θCHAN* ə}) *f*

→ nstoreə = nstore *f* report! = OK)

→ϖ (*name*? ™ dom *nstore f θFS* ə = *θFS f*

→ report! = NoSuchName)

→ϖ (dom *cstore* = *CID f θFS* ə = *θFS f*

→ report! = NoFreeCids)

∠_____

The precondition *cid*! ™ dom *cstore* is the negation of dom *cstore* = *CID* and vice versa, in the sense that the system attempts to obtain a new unused channel identifier (i.e. *cid*!) and if successful, the condition *cid*! ™ dom *cstore* holds. Otherwise there are no free identifiers left and dom *cstore* = *CID* prevails.

The partial preconditions of operation *open* are:

name? ε dom *nstore f cid*! ™ dom *cstore* (1)

name? ™ dom *nstore* (2)

dom *cstore* = *CID* (3)

The total precondition of *open*, namely (1), (2), and (3) is a tautology but not a *partition* since two of these conditions overlap (conditions (2), and (3) may hold simultaneously). Often in a specification this non-determinism is deliberate because it allows implementers flexibility. However, if preconditions overlap in this way, then a sequence of automatic refinement steps could generate the following incorrect structure:

if precondition 1 **then** S1

elseif precondition 2 **then** S2

elseif precondition 3 **then** S3

endif

The semantics of the above code fragment requires the preconditions to be pair-wise disjoint, leading to our first design heuristic:

Heuristic #1: Ensure that the precondition to a total operation is a partition whenever non-determinism is undesirable.

5.2.2 Communication with the User.

There is a further aspect to the above discussion as far as feedback to the user is concerned: Consider the scenario where there is no free channel available (i.e. dom *cstore* = *CID*) and the input file name, *name*?, is incorrect (i.e. *name*? ™ dom *nstore*). Suppose also that owing to the above non-determinism, the message '*NoFreeCids*' is displayed, telling the user to wait for a channel to become available before proceeding. However, once a channel is released by another process, the user can try to reconnect again, only to be faced with the message '*NoSuchName*'. One could argue that this message should have been displayed together with the message about the channel, so that the user could have fixed the problem in the meantime, instead of having to wait for a free channel. This leads to our second design heuristic:

Heuristic #2: Maximize communication with the user of the system.

Note that the above guideline agrees with the following principle proposed by Norman (1998):

"Narrow the gulfs of execution and evaluation. Make things visible, both for execution and evaluation."

5.2.3 Value of Undefined Output.

Consider the following definition of a simple file system Woodcock and Davies (1996) where Key and Data are basic types.

∪ *File* _____

→contents : Key ♣ Data

∠_____

A robust operation to read a file is:

∪ *Read* _____

→ΞFile

→*k*? : *Key*; *d*! : *Data*; *r*! : *Message*

∩_____

→(*k*? ε dom *contents* ƒ *d*! = *contents k*? ƒ *r*! = OK)

→ ϖ

→(*k*? ™ dom *contents* ƒ *r*! = Key_not_in_use)

∠_____

Note that *d*! is unspecified under the error condition *k*? ™ dom *contents* .Woodcock and Davies (1996) claim that an output variable 'can take any value' if the precondition is not satisfied. However, a possible interpretation of this claim is that *d*! may be given a value *contents k*, for any *k* ε dom *contents* which is undesirable.

Instead, we could specify the value of an output variable like *d*! to be *undefined* in the error case. This is achieved by insisting that all sets from which output may be generated be 'lifted' to make provision for undefined values, much like the technique used in the semantics of programming languages (Schmidt 1986). If we denote an undefined value by B then we extend the set *Data* to $Data_B = Data$ Y {B}. This observation leads to:

Heuristic #3: Ensure that all sets from which output may be generated are extended to allow for undefined values. (Note how this heuristic supports heuristic #2 above.)

5.2.4 Functional Cohesion.

Bahrami (1999) defines cohesion as a measure of the 'single-purposeness' of an object. High cohesion is desirable and low cohesion is considered bad design, since low cohesion implies the grouping together of unrelated activities. It is often stated that a module has good cohesion if its purpose can be expressed by 'a simple sentence containing a single verb and a single object' (Yourdon, 1994). A most desirable kind of cohesion is functional cohesion (Pfleeger, 1998) and this is the kind of cohesion we advocate in the design of a formal specification. The above natural language definition given by Yourdon (1998) is unfortunately too imprecise and we refine the idea below.

For the purpose of achieving high functional cohesion in a formal specification, we propose breaking up every operation in a specification into a sequence of *primitive* operations, inspired by a thesis put forward by Scheurer (1994): "Set theory, based on logic, is a universal language in which all (solvable) problems may be formulated and solved." Since the Z specification language is based on first-order logic and a strongly typed fragment of Zermelo-Fraenkel set theory (Enderton, 1977), we propose to define every primitive as manipulating just *one* component of the state of our system, using an operation or definition from standard set theory. The above ideas on cohesion crystallize into the following heuristic:

Heuristic #4: Maintain high cohesion in a formal specification by defining every operation on the state as a sequence of primitives such that every primitive manipulates at most one state component using a standard set-theoretic operation or definition.

5.2.5 Explicit Preconditions and Relationships.

The next heuristic stems from a philosophy proposed by Norman and also mentioned by the authors of the above UNIX filing system. Norman (1998) stresses the importance of making 'things visible', i.e., at every step showing which actions are applicable or allowable:

"In each state of the system, the user must readily see and be able to do the allowable actions. The visibility acts as a suggestion, reminding the user of possibilities and inviting the exploration of new ideas and methods."

Consider next, the expanded version of the operation to remove a file in UNIX, where *FS* is the state previously defined (Morgan and Sufrin, 1993).

∪ destroyFS _____

→ΔFS

→fid? : FID

∩_____

→fid? ε dom fstore *f*

→usedfids ζ dom fstore *f* usedfidsə ζ dom fstoreə *f*

→usedfids = ran nstore χ {chan.fid | chan ε ran cstore} *f*

→usedfidsə = ran nstoreə χ {chan.fid | chan ε ran cstoreə} *f*

→fstoreə = {fid?} ψ fstore *f* fidə = fid *f*

→posnə = posn *ƒ* cstoreə = cstore *ƒ* nstoreə = nstore

∠_____

The complete definition of `destroyFS` in Morgan and Sufrin (1993) also mentions directory names, but these definitions are beyond the scope of this paper. The precondition of `destroyFS` is given by (`fid?` represents the identifier of the file that is to be deleted and `usedfids` the set of file identifiers currently in use, e.g. open files):

fid? *ε dom* fstore *ƒ* usedfids *ζ dom* fstore *ƒ*

usedfids = *ran* nstore *χ {*chan.fid | chan *ε ran* cstore*}*

A file cannot be deleted while in use, as we show next:

1. usedfidsə ϕ dom fstoreə [postcondition of destroyFS]
2. usedfidsə = usedfids *[since* nstoreə = nstore *ƒ* cstoreə = cstore*]*
3. usedfids ϕ dom fstoreə [from 1. and 2.]
4. fid? ⌠ dom fstoreə [fstoreə = {fid?} ψ fstore]
5. fid? ⌠ usedfids [from 3. and 4.]

Therefore, a file cannot be destroyed while in use. Hence, condition *fid*? ⌠ *usedfids* is a further precondition of the correct operation of *destroyFS*. One could argue that the absence of this condition from *destroyFS* violates the principle put forward by Norman (1998) above, namely, to make things visible, that is, to explicitly show all the conditions which need to hold for an action to be applicable. In fact, Morgan and Sufrin (1993) label operation *destroyFS* as being 'dishonest'.

One can also argue that all postconditions, especially those involving relationships between before and after state variables, be explicitly shown (van der Poll and Kotzé, 2002). The above discussion leads to an important design heuristic:

Heuristic #5: Ensure that all operations are honest by

listing all preconditions explicitly, and

showing all postconditions including the relationship between each changed state component and its after state value, unless such relationship is 'easily provable' from the specification.

5.2.6 Undo Changes in State Components.

The last design heuristic relies on the well-known HCI principle of 'undo' as advocated by Norman (1998):

"Make it possible to reverse actions – to "undo" them – or make it harder to do what cannot be reversed."

The above philosophy suggests the following principle for specification work:

Heuristic #6: Specify an undo counterpart for every operation that changes the state. The idea is to reverse the effect of a state change.

Next we suggest ways in which these heuristics can be employed to validate the effectiveness of use case map representations. Important properties of UCM-UI models such as their completeness, consistency, precision and correctness are addressed.

5.3 Operators to Validate UCM-model

For the UCM-UI model, and for generally the UCMs model we introduced some operators to help identify possible inconsistencies, incompleteness and ambiguities in the requirements and we show below how the above heuristics may help to improve the overall quality of the requirements.

5.3.1 Consistency.

There are two notions of consistency related to use cases, namely, the consistency between two use cases and consistency of a UCM-UI model. Both these consistency issues may be validated in our proposed model.

5.3.1.1 The consistency of two use cases

The consistency between two use cases is a property whereby the information contained in one use case does not conflict with information in the other use case. In this paper we are concerned with two aspects of use case consistency, namely, *path consistency* and *precondition consistency*.

Definition 1a: Consider two use cases A and B with responsibilities a and b in each. Use case A is *path consistent* with respect to use case B, written $A \leq_{path} B$, iff every virtual path from a to b in A is matched by a corresponding path from a to b in B. Also, use case A is *precondition consistent* with respect to use case B, written $A \leq_{pre} B$, iff their preconditions are disjoint and if A and B are the only use cases in the system then the disjunction of their preconditions is a tautology.

Definition 1b: Use case A is consistent with respect to use case B, written $A \leq B$ iff $A \leq_{path} B$ and $A \leq_{pre} B$.

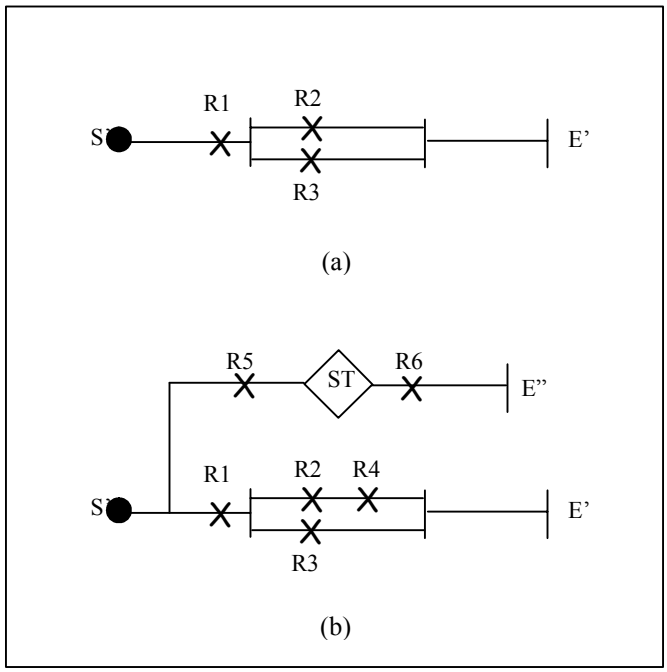


Figure 5.1 Example of inconsistency between two use cases

For example, the use case (a) in Figure 5.1 is not path consistent with the use case (b), since the path from *R1* to *E'* in (a) is not matched by a corresponding path from *R1* to *E'* in (b). These two UCMs are also not precondition consistent since both have the same precondition, S'. A high degree of consistency between a use case *A* and use case *B* may be achieved by removing any inconsistencies between *A* and *B*. Heuristics #1 and #5 in Section 5.1 can be used to minimize possible inconsistencies between *A* and *B* as follows:

– Apply the first part of heuristic #5 by explicitly stating the preconditions of each use case at its start point. Therefore, the conditions under which either *A* or *B* are applied are known and visible to the software engineer.

– Apply heuristic #1 by ensuring that the conjunction of the precondition of *A* and the precondition of *B* is false (i.e. the preconditions are disjoint). If *A* and *B* are the only two use cases in the system then ensure that the disjunction of their preconditions is *true*. Hence their preconditions form a partition.

5.3.1.2 The Consistency of a UCM-UI model.

A UCM-UI model is consistent if the set of use cases making up the model are consistent among themselves; i.e., if for any two uses cases *A* and *B* in the model we have both path consistency and precondition consistency between them. Hence, the set of use cases 'do not contain conflicting information among themselves'. This

notion of distributed consistency can be formalized using the notion of consistency between two use cases.

Definition 2: A set of use cases $X = \{U_1, U_2, \ldots, U_n\}$ is consistent iff there exists a syntactically correct use case Y such that $U_k \leq Y$, for all $k = 1, 2, \ldots, n$. Y then represents the UCM-UI model.

The following algorithm builds a consistent UCM-UI model from a set of use cases $X = \{U_1, U_2, \ldots, U_n\}$ if X is consistent; otherwise an error condition is raised.

Algorithm 1: Checking the consistency of a UCM-UI model.
Input: A set of use cases $X = \{U_1, U_2, \ldots, U_n\}$.
Output: A consistent UCM-UI model $Y = Y_n$, or confirmation of inconsistency.
Begin

Let $Y_1 = U_1$, a use case in X.
For k := 1 ***to*** n-1 ***do***
If Y_k is consistent with respect to U_{k+1}
then $Y_{k+1} = Y_k \cup U_{k+1}$
Else error
End For

End

To create a consistent UCM-UI model from a set of use cases, we start with a use case U_1 equal to Y_1 and then check if U_2 is consistent with Y_1. If consistent, form $Y_2 = Y_1 \cup U_2$. If any U_i is not consistent with Y_{i-1}, then fix the inconsistency and repeat the test.

Verifying the consistency of a given UCM-UI model is an extension of the above process involved by verifying the consistency of each use case with a partially completed UCM-UI model. Again design heuristics #1 and #5 are useful in this regard. Once a syntactically correct use case (Y above) has been identified, the consistency of each pair of use cases (U_k, Y) for k = 1, 2, . . . , n is verified, as before.

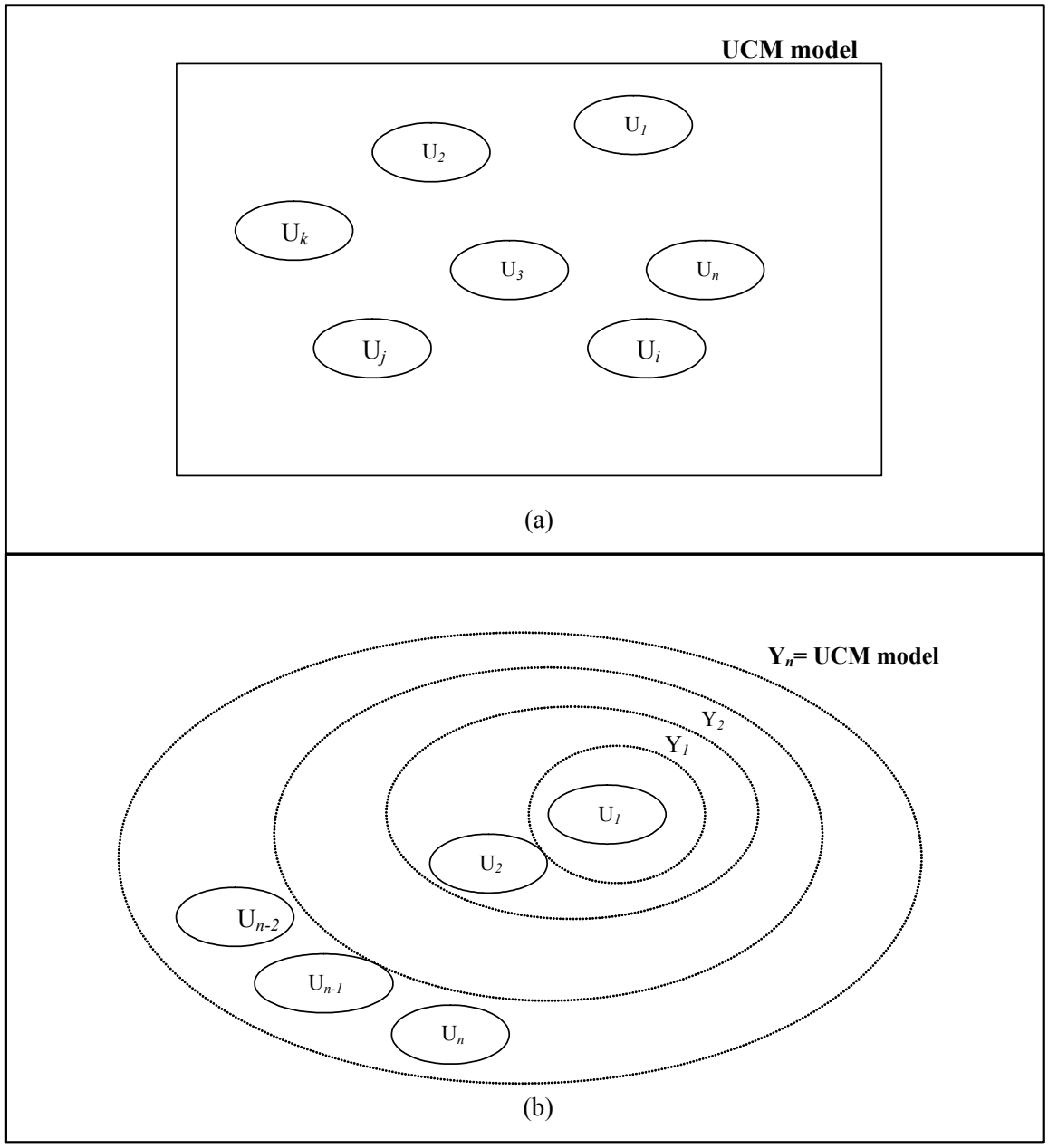


Figure 5.2 (a) A UCM-UI model constructed from a number of use cases. (b) Building Y, a consistent UCM-UI model from a sequence of Y_is, $1 \leq i \leq n$.

5.3.2 Completeness.

When a UCM-UI model is validated against a set of well-defined use cases, the question arises of how adequate such a validation is. This leads to the notion of completeness of a set of use cases with respect to a UCM-UI model. It means that any given piece of information contained in the model is covered by at least one scenario. Therefore, checking the consistency against the set of use cases covers all the information contained in the modeluse case. There are two notions of completeness related to use cases, namely, completeness of a UCM-UI model with respect to the requirement specifications of the system and self-completeness, both may be validated in our proposed UCM-UI model.

Verifying the completeness of a UCM-UI model against the requirement specifications (*RS*) of the system[1] boils down to checking the adequacy of validation. On the other hand, checking such completeness also means checking whether a UCM-UI model contains information not contained in the requirement specifications (*RS*).

For completeness we check whether any given piece of information contained in the requirements specification of the system is covered by at least one use case. Assuming a UCM-UI model is complete in the sense that it already contained all the information of the required system; incompleteness then implies that the model contains unnecessary or incorrect information.

Definition 3: The UCM-UI model UCM = $\{U_1, U_2, \ldots, U_n\}$ is complete with respect to a requirements specification *RS*, if:

– For all tasks (and sub-tasks) t in *RS*, there is U_k in UCM such that U_k contains responsibility r that represent the tasks (or sub-tasks) t; and

– For all flows f in *RS*, there is U_k in UCM such that U_k contains path f.

To check whether all the information contained in the requirements specification is covered by the UCM-UI model, we have to establish whether all relevant use cases were indeed incorporated into the construction of the requirements model. A possibility is to consider the precondition to each use case as follows.

[1] The Requirement Specifications of a system are the preliminary and general user requirements of a system. See *RS* in Section 4 for the MRS system.

– Use heuristics #1 and #5 to ensure that the preconditions of the individual use cases are pair wise disjoint and their disjunction true.

– Calculate the precondition (say *S*) of the UCM-UI model by taking the disjunction of the preconditions of all individual uses cases used to synthesize the model. *S* will be a partition; if not, then at least one use case is not covered by the UCM-UI model.

5.3.3 Self-Completeness

Given a set of use cases generated from a requirement specifications *RS* to synthesize a UCM-model, we require that the resultant UCM-UI model (1) includes all the information contained in *RS*, (2) but at the same time contains no information that cannot be inferred from *RS*. (3) Moreover, the UCM-UI model must be consistent with respect to all the use cases. This leads to the definition of completeness of a set of use cases or self-completeness of a set of use cases.

Definition 4: A set of use cases $U = \{U_1, U_2, \ldots, U_n\}$ is self-complete, if U is consistent with respect to Y, Y is complete with respect to RS and RS is complete with respect to Y, where Y is the simplest set of use cases that contains Y_K* (i.e., $X \leq Y$) for $Y_K = < R_K, P_K >$, $K = 1,2, \ldots, n$ and R_K the set of responsibilities in Y and P_K the set of paths in Y.

In other words, to create a UCM-model from a set of use cases *X*, (1) we need to generate the simplest consistent set of use cases *Y* following Algorithm 1. (2) check if the set of use cases *Y* is complete with the *RS*. If not, the information contained in the set of use cases *Y* is incomplete in the sense that there are paths not covered by any use case and we need to add new use cases to *Y*. (3) check if *RS* is complete with respect to *Y*. if not, then *Y* contain information that cannot be inferred from *RS*. To verify the completeness of a set of use cases one may use design heuristics #1, #2 and #5 as follows.

– Use a ‘high level’ version of heuristic #2 to check with the user that all possible use case scenarios have been covered.

– By using heuristics #1 and #5, establish for each use case identified its precondition such that the collection of all these preconditions form a partition (i.e. they are pair wise disjoint and their disjunction is true). This particular set of use cases is then complete at least as far as the conditions under which they are applicable (i.e. their

preconditions) are concerned. More research is needed aimed at formulating heuristics to check (formally) the completeness of the paths and responsibilities of a set of use cases.

A UCM-model can be uniquely synthesized from a set of use cases *U* such that *U* is consistent and complete with respect to the UCM, if and only if *U* is consistent and self-complete.

5.3.4 Precision.

Precision is the absence of ambiguity in the semantics of a representation. A use case representation is precise if the use case analyst can answer questions such as these: 'What happens next?', 'What can happen next other than what appears in this use case?', 'Who or what is responsible for doing this?', 'What are the consequences of this event?'. Use case representations fall on a spectrum with some representations having the goal of giving the analyst or stakeholder a 'feel' for the envisaged system, while others are essentially trace specifications (Anton and Potts, 1998). We believe that our proposed model is precise since it describes end-to-end causal scenarios at different levels of abstraction, allowing the behavior of complex systems to be described effectively.

– To verifying precision, let's consider how some of the heuristics previously defined may help us to answer some of the questions above:

– What happens next? Design heuristic #4 advocates breaking up an operation into a sequence of smaller operations giving us the benefit of being able to talk about a previous, a current and a next operation. Being a current element of an operation sequence uniquely defines any next operation. Therefore, given any current point in a sequence of events, 'What happens next' is uniquely defined as far as a next operation is concerned

– Who or what is responsible for this? This question could sometimes be translated to: 'Under what conditions can this happen'. Therefore, heuristic #1 and the 1st part of heuristic #5 may help in establishing the precondition, i.e. telling us the conditions that must prevail for the operation to be applicable.

– What are the consequences of this event? The consequence of an event is directly related to the post-condition of such event. Therefore, applying the second part of heuristic #5 helps in finding an answer to this kind of question.

Two additional issues are in order:

– System output is often generated in the Dialog dimension of an extended UCM (Seffah et al. 2002) and since heuristic #3 advocates that undefined output be explicitly shown, we propose that the extended UCM notation be further extended to make provision for undefined output.

– A responsibility in a UCM is a generic process, represented by a cross, and like any other process its effect may have to be reversed in line with design heuristic #6. Therefore, for every such responsibility one could add an adjacent symbol to this effect (e.g. ↺ indicating a generic undo).

5.4 Analysis of the MRS System

In this section we analyze the consistency, completeness, and precision of the CUCM for the Library System (LB) presented in Chapter 4.

5.4.1 Consistency between two use cases.

Applying Definition 1a, we find that 'update database' in UC4 is a stub and is inconsistent with 'update database' in UC5, which is a responsibility. Therefore, we had to change 'update database' in UC5 to a stub.

5.4.1.1 Consistency of a UCM-UI model.

By applying Definition 2 we can increase confidence in the consistency of our UCM-UI model. For example, let Y_1 be UC_1 = 'ConnectToSystem'. Then compare Y_1 with UC_2 = 'SearchBook' and if Y_1 is consistent with UC_2, then add UC_2 to Y_1 to obtain $Y_2 = Y_1 \cup UC_2$. Next Compare UC_3 to Y_2 and if they are consistent then add UC_3, i.e. $Y_3 = Y_2 \cup UC_3$. Repeat this procedure as needed till $Y = Y_n = (UC_1 \cup UC_2 \cup \ldots \cup UC_6)$. Resolve any inconsistencies that may arise along the way between any of the use cases.

5.4.1.2 Completeness of a set of use cases.

Validate the completeness of a set of use cases with respect to the requiemtns model by applying Definition 3. We find that according to the requirements specification R6 the Librarians extends due date for a loan upon the request of the borrower whereas the task 'extend loan due date' is not in any use case. Therefore, we have to add this task to the appropriate use case, namely, UC_5.

UC_5: ReturnBook (2 scenarios)

> *S_1*: Librarian checks due date for late fees, check book in and updates the database.
>
> *S_2*: Librarian checks due date for late fees, extend due date for the loan, and updates the database.

5.4.2 Self- Completeness.

Applying Definition 4 we find that the set of use cases U = {UC_1, UC_2, UC_3, UC_4, UC_5, UC_6} is self-complete and, therefore, a UCM-UI model can be synthesized uniquely from U since U is consistent and self-complete. The complete and consistent root use case map is shown in Figure 5.3.

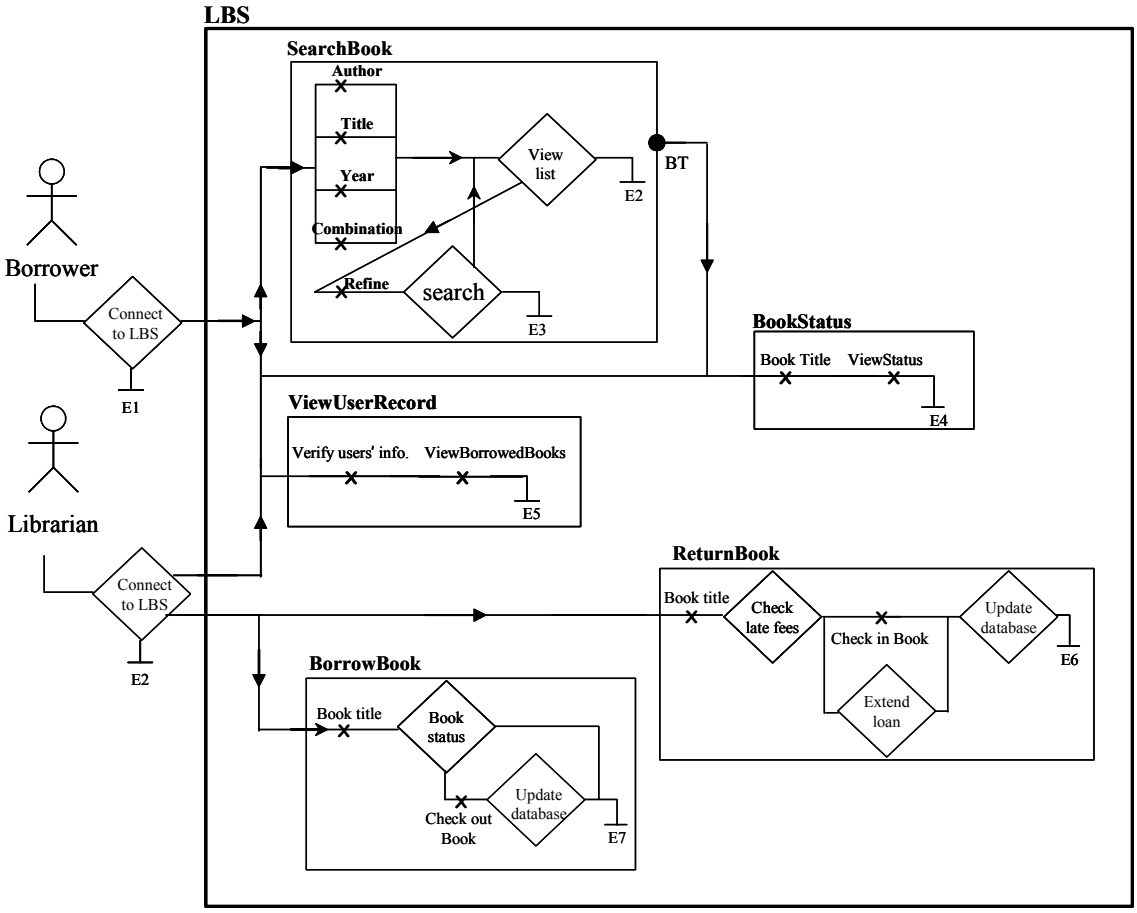


Figure 5.3 The complete and consistent CUCM of the LB System..

5.4.3 Precision

To examine the precision of the CUCM we answer the following questions.

– **What happens next?** The CUCM-root map and the describes end-to-end causal scenarios at different levels of abstraction, allowing the behavior of the systems to be described effectively, by breaking up stubs into a sequence of smaller operations giving the benefit of being able to talk about a previous, a current and a next operation.

– **Who or what is responsible for this?** In Figure 5.3 each responsibility in the UCM is bounded to a component that is in charge to perform the tasks and actions associated with that responsibility.

– **What are the consequences of this event?** In Figure 5.3 each use case is describes end-to-end causal scenarios. Thus, the consequence of any event is directly related to the ending point (post-condition) of such event.

– **Undo effect:** According to design heuristic #6. we must add for every responsibility an adjacent undo symbol to this effect (e.g. ↺ indicating a generic undo).

5.5 Conclusions

The contribution of this chapter is the embedding of a set of heuristics for drawing up operators to validate the extended UCM-UI model. The proposed operators allow for a formal analysis of the consistency, completeness and precision of these semi-formal requirement models. The analysis of Library System example illustrate that the use of the proposed operators had facilitated in generating models and implementations faster and at a lower cost while improving their correctness and traceability with respect to the requirements. However, the amount of work it generates shows the vital need for a CASE tool to manage the UCM-UI model along with additional information pertinent to user interface development.

Chapter 6

Supplementing Scenarios and Use Case Maps with Predictive Metrics

Abstract

Although metrics have long played an important role in computer science and software engineering, little has been done in applying metric strategies in the requirement specification phase. Traditional approaches to measuring software usability have centered on testing, and the schemes for laboratory and field-testing are well established. However, because testing depends on having something to test software, prototype, or simulation it is difficult to conduct testing early in the design process. Ideally, designers would like to be able to predict usability based on metrics computed from visual designs or paper prototypes. In this chapter, a pragmatic suite of metrics; the UCM-UI Metrics Suite; has been developed to predict usability from scenarios documented as use case maps. This suite of metrics is part of the Scenario and Use Case Requirements Engineering (SUCRE) Framework. The metrics suite includes simple structural measures as well as content-sensitive and task-sensitive metrics.

6.1 Introduction

As quality factor, usability of interactive systems is currently receiving more and more attention. In particular, industry now recognized the importance of adopting usability methods during the development cycle for increasing the acceptability of new products on the market (Madsen, 1999). However, one of the main complains of industry is that cost-effective usability evaluation tools are still lacking. This inhibits most companies from actually performing usability evaluation, with the consequent result that a lot of software is still poorly designed and unusable.

Usability depends on the match between the product and the users under the particular constraints of the environment and tasks being performed with the product. The problem is that usability, seen like this, depends on the world when the product is used not when it is designed. So if we want to design better products, foresight has to be used. Foresight can be based on case histories from previous product evaluations; sometimes foresight can be focused by general psychological or socio-technical knowledge. In all approaches, design for usability requires considerable expertise and commitment to usability, neither of which is conventionally nor realistically available in the crucial early stages of the design. Often one therefore seeks improvements in usability after the initial stages of design; once a prototype exists. Many usability-driven design insights come too late to address core issues. At the time of evaluation, prototypes may be considerably advanced in development, so changes suggested by the results of evaluation may be too expensive to implement or strongly resisted because of the investment of time and effort in the existing system (Ruthford, 2002; Grice, 2003).

Predictive metrics are an emerging and promising approach to solve some these problems while assessing the quality of software from early requirements design artifacts including user interface prototypes. Predictive metrics does not guarantee usability, but it can help to reduce the costly usability problems that can be discovered latter after developing a fully functional prototypes.

We are concerned with methods to improve usability that can be employed as early as possible in our proposed Scenario and Use Case Requirements Engineering (SUCRE) Framework. The motivation of the presented work is to make a contribution to user interface design, specifically to discuss ways of measuring usability from

requirement specifications rather than implementations. The aim is to show that solid results can be obtained with very simple and general assumptions, and to illustrate certain usability criteria that can be established in the design cycle. Ideally, of course, user interfaces should be empirically evaluated and then improved, but in practice many products are designed and then fobbed off to users, with little opportunity for improvements. It is therefore crucial to have design support for usability measurement.

The reminder of this chapter is as follows. Section 6.2 focuses on understanding usability, why and how to evaluate and predict usability. This section, also discusses briefly usability assessments and metrics. Section 6.3 is the core of the chapter, since it describes the UCM-UI metric suite. Section 6.4 illustrates via an example how to use the UCM-UI metric suite in the requirement phase. Finally, Section 6.5 gives the conclusions.

6.2 Usability Assessment and Metrics

There is not one agreed upon definition of usability. Several authors have proposed definitions and categorizations of usability and there seems to be at least some consensus on the concept of usability and they mostly differ on more detailed levels. The International Organization for Standardization (ISO) defines usability as: "the *effectiveness*, *efficiency* and *satisfaction* with which specified users can achieve specified goals in particular environments" (ISO 9241-11, 1991). Here, the effectiveness of a system relates to the accuracy and completeness with which users achieve specified goals while efficiency is the resources expended in relation to the accuracy and completeness with which users achieve goals. Satisfaction, according to ISO 9241-11 is the comfort and acceptability of use. The ISO definition approaches usability from a theoretical viewpoint and may not be very practical.

Nielsen (1993) has a slightly different definition that is specified in elements that are more specific. Nielsen only regards expert users when talking about efficiency although learnability is also directly related to efficiency. Memorability mainly relates to casual users and errors deal with those errors not covered by efficiency, which have more catastrophic results. A similar definition is given by Shneiderman (1998). Shneiderman does not call his definition a definition of usability but he calls it *"five measurable human factors central to evaluation of human factors goals"*. As seen from Table 6.1, Shneiderman's definition is essentially identical to Nielsen's definition and only differs in terminology.

ISO 9241-11	Nielsen	Shneiderman
Efficiency	Efficiency	Speed of performance
	Learnability	Time to learn
Effectiveness	Memorability	Retention over time
	Errors/Safety	Rate of errors by users
Satisfaction	Satisfaction	Subjective satisfaction

Table 6.1 Usability as in ISO 9241-11, J. Nielsen, and B. Shneiderman.

A wide range of usability evaluation techniques has been proposed. Some evaluation techniques, such as testing with users using software or paper prototypes are the traditional ways of evaluating usability. A disadvantage of testing with software prototypes is that it can only be done late in the design process when a lot of

design choices have already been made. Testing with paper prototypes has the disadvantage that it still does not ”feel” like a real system and data obtained in testing can only be used to evaluate general concepts. Ideally, designers should be able to evaluate their design solutions early in the design process when only high level and abstract specifications exist. However, if no users are involved, evaluation should be done carefully with regards to valid interpretations (Shneiderman, 1998; Bamum, 2002).

Usability evaluation during the design process is more problematic than evaluating with users. Although mockups and paper prototypes can be tested with users, the usability of the interface cannot be evaluated directly. What can be done is looking at the means that influence the usage indicators. Using walkthroughs and scenarios each of the means can be evaluated by looking at the way they are present in the design and by estimating the positive or negative impact on the usage indicators. This kind of early evaluation does not replace the need for late evaluation with users but can contribute when good choices of means can be made.

Another way of ensuring usability during the design process is by using formal design models. Many models and techniques exist for describing designs using formal notations. State charts, GOMS (Card et al., 1983), ConcurTaskTree’s (Palanque and Paterno, 1997) and similar notations are used to describe designs. These kinds of notations are usually strong in describing structural aspects of a design (the dialog structure) and very weak at describing presentational aspects. Payne says, “as far as TAG is concerned, the screen could be turned off” (Payne and Green, 1989). Measuring usability is not always easy. Task performance times are easy to measure but satisfaction and memorability are harder to quantify. More standardized measurement can be obtained using questionnaires such as the Questionnaire for User Interaction Satisfaction (QUIS) and the Software Usability Measurement Inventory (SUMI). However, these questionnaires are only valid for certain classes of systems.

Usability metrics are another method of quantifying a qualitative evaluation process of the interface. They are software quality metrics that measure or estimate some factors or dimensions of the software quality. Usability metrics offers interface designers additional tool to help guide them towards more usable solutions (Nielsen 1996; Contantine, 1996, Bevan and Curson, 1996).

Thomas and Bevan (1996) defined Usability Metric as “a number expressing the degree or strength of a usability characteristic, possessing metric properties, obtained by objective counting rules, and with known reliability and validity”. Usability metrics have known maxima and minima, their scale of measurement is known, they possess scale metric properties, they are gathered by objectively verifiable rules of counting, and they have demonstrated reliability and validity. Usability metrics are said to instantiate or operationalise a characteristic of usability and must be interpreted according to the context in which they were measured.

Usability metrics for user interfaces comes in several flavors: structural metrics, semantic metrics, and procedural metrics. Structural metrics, which are based on surface properties of the configuration and layout of the user interface architectures that can simply be counted or toted. Various metrics have been devised and tried, including: (i) number of visual components or widgets on a screen or dialogue. (ii) amount and distribution of white space between widgets. (iii) alignment of widgets relative to one another. (iv) number of adjacent screens or dialogues directly reachable from a given screen or dialogue. (v) longest chain of transitions possible between screens or dialogues. Structural metrics do not typically address questions of great significance in everyday design problems. User interface designers are confronted with more vexing and substantive problems than how many widgets are on a dialogue box or amount of white space surrounding them.

Semantic metrics, which are content-sensitive that take into account the nature of user interface components or features in terms of their function, meaning, or operation. Semantic metrics can measure aspects of user interface designs that depend on the concepts and actions that visual components represent and how user makes sense of component and their interrelationships.

Procedural metrics, which are task-sensitive on aspects of the actual tasks or scenarios that may be carried out with a user interface. Procedural metrics deal with the fit between the various tasks and a given design in terms of its content and organization. In general, we can expect designers to get more practical direction from context sensitive and task sensitive measures that from simple structural metrics. Effective metrics should reliably predict important aspects of the usability of software in actual application, such as task performance, learning time, or error rates. User interface design metrics need also to be sufficiently sensitive so as to distinguish

between similar designs that are likely to differ in ultimate usability. Ideally, the metrics itself or the process by which it is computed should provide information that suggests ways to improve a design (Constantine and Lockwood, 1999).

6.3 The Proposed Usability Metrics Suite for UCM-UI

There is no one metric that can encapsulates enough of the detail and complexity of a user interface design to serve as a global indicator of quality. It takes a suite of metrics to cover the various factors that make for a good user interface design. In our research, we constructed a UCM-UI Metrics Suite for guiding user interface design. The goal of the suite was to develop design metrics that are simple to use, are conceptually sound, and have a clear and transparent rationale connecting them to establish principles of good design. Constantine and Lockwood in (1999) pointed up that practical metrics should be sound, simple, and easy to use and should fulfill the following criteria:

– Easy to calculate and interpret

– Apply to paper prototypes and design models

– Have a strong rationale and simple conceptual basis

– Have sufficient sensitivity and ability to discriminate between designs

– Offer direct guidance for design

– Effectively predict actual usability in practice

– Directly indicate relative quality of designs

The UCM-UI Metrics Suite has evolved considerably using metrics from the literature and adding new metrics to it. Currently ten metrics are included that together cover assortment of measurements likely to be significant to designers seeking to improve the usability of their software. The UCM-UI Metrics Suite does not guarantee usability, but it can help to reduce the costly usability problems that can be discovered latter after developing the actual interface.

6.3.1 New Metrics for UCM-UI Model

In this section, we defined new metrics that can help with the metrics presented in Section 6.3.2 to build a metrics suite that predicts usability of the UCMs-UI models early in the requirement phase.

6.3.1.1 Task Analysis (*TA*)

Task Analysis (*TA)* metric makes it possible to design and allocate tasks appropriately within the new software. The functions to be included within the software and the user interface can then be accurately specified. *TA* can be conducted to understand the current software and the information flows within the user interface. Failure to allocate sufficient resources to this activity increases the potential for costly problems arising in later phases of development.

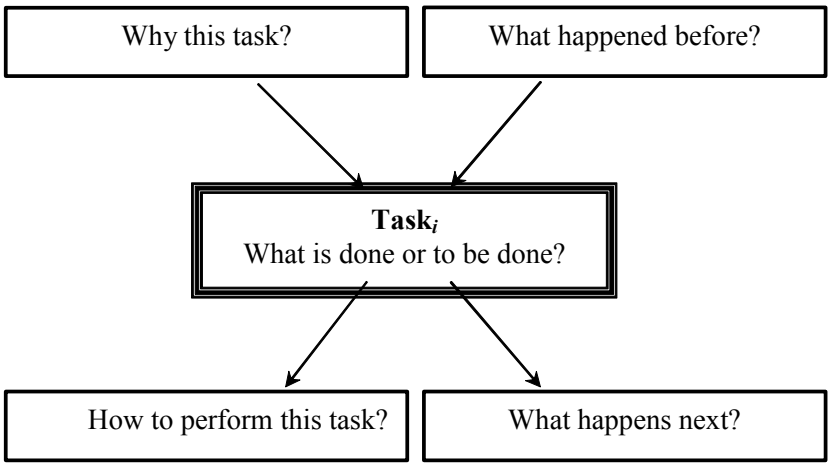


Figure 6.1 Analysis of Task$_i$.

Definition: Task Analysis is the study of what a user is required to do in terms of actions and/or responsibilities to complete a task. See Figure 6.1

Formula:

$Task_i$	Why this task?	What happened before?	What is done Or to be done?	What happens next?	How to perform this task?
$Task_1$	To pay a bill	S_2	S_3	S_6	$[a_1]$, r_2,$[a_2]$,r_3
$Task_2$	To …	S_2	S_5	E_2	$[a_6]$, r_7, r_8
…					
$Task_n$	To…	S_6	S_7	E_3	$[a_2]$, r_6,$[a_5]$,r_9

Figure 6.2 Task Analysis Form.

It should be noted that using the *TA* metric as a formal way to analysis task can be of considerable value in focusing attention to relevant issues to consider when designing products from the users' point of view.

Scale: *TA* provides detailed knowledge of the tasks that the user wishes to perform on the interface. Thus, it is a reference against which the value of the interface functions

and features can be tested. Uncompleted cells in the *TA* metrics form indicate areas where task are poorly understood, or specified.

6.3.1.2 UCMs Model-Consistency Metric

There are two notions of consistency related to use cases as discussed in the previous chapter: consistency of two use cases, and consistency of a UCMs model, both can be predicted in our proposed model.

The consistency between two use cases is a property whereby the information contained in a use case does not conflict with other use case. Consider two use cases *A* and *B* with responsibilities *a* and *b* in each. Use case *A* is *consistent* with respect to use case *B*, written $A \leq B$, if: *A* is a sub-graph of *B*, i.e., for each virtual from *a* to *b* in *A* is matched by a corresponding path from *a* to *b* in *B*.

The second notion of consistency; consistency of UCMs model; can be formally defined by using the notion of consistency between two use cases as follows. A UCMs model is consistent if the set of use cases constructing the UCMs model are consistent; i.e., if the set of use cases do not contain conflicting information among themselves.

Definition: The UCMs Model-Consistency metric gives a quick handle on the coherence of the set of use cases constructing the UCMs model.

Formula:

UC_1	**UC_1**			
UC_2	≤	**UC_2**		
UC_3	-	≤	UC_3	
…				
…				UC_{n-1}
UC_n	⊄	≤	-	-

-: the two use cases do contained shared responsibilities, thus we do not check their consistency.

≤: the two use cases are consistent.

⊄: the two use cases are inconsistent

Figure 6.3 UCMs Model-Consistency From.

To ensure the construction of consistent UCMs model, we need to guarantee that the requirements captured by the set of use cases are not in conflict with each other. Thus, we consider use cases with shared responsibilities. If the use cases are inconsistent, it will backtrack to the information elicitation and scenario description to resolve the conflicts.

Scale: The metric used to evaluate consistency is a count of the $\not\subset$ symbol used in the form. The $\not\subset$ symbol is used to indicate that two use cases with shared responsibilities are inconsistent with each other and inconsistency need to be solved.

6.3.1.3 UCMs Model-Completeness Metric

Completeness of the requirement specification of the new systems means the validation of the requirement specification of the new systems against the UCMs model is adequate. On the other hand, checking such completeness also means checking if the requirement specifications of the new system contains information not contained in the UCMs model. Assuming a set of use cases is complete in the sense that it already contained all the information of the required system; incompleteness then means that the model must contain unnecessary or incorrect information.

Definition: the UCMs Model-Completeness metric examines the comprehensiveness of UCMs model $M = \{U_1, U_2, \ldots, U_n\}$ with respect to the requirement specifications of the new system *RS*, such that:

– For all tasks (and sub-tasks) *t* in *RS*, there is U_k in *M* such that U_k contains responsibility *r* that represent the tasks (or sub-task) *t*; and

– For all paths *p* in *RS*, there is U_k in *M* such that U_k contains path *p*.

Formula: to ensure completeness, all tasks (including their sub-tasks) in the requirement specifications of the required system must be covered by at least one use case following the correct order of task performance.

System	UCMs Model-Completeness Metric					
Tasks (sub-task)	UC 1	UC 2	UC 3	…	UC *n*	Comments *TBA*= to be added

Task 1			
Sub-task 1.1	Y		
Sub-task 1.2	Y		
Task 1			
Sub-task 2.1		Y	
Sub-task 2.2		Y	
Sub-task 2.3			TBA
Task *n*		Y	

Figure 6.4 UCMs Model-Completeness Form.

All use cases not marked in the metric must be reviewed to check their significance to keep or remove from the UCMs model. Moreover, if incompleteness is discovered, it will backtrack to the scenario identification to create new scenarios that cover the missing situations.

Scale: If UCM-UI model is complete, it will have all of the tasks and their sub-tasks specified in adequate detail in the use cases to allow design and implementation to proceed. The metric used to evaluate completeness is a count of the *TBA* (to be added) acronyms used in the form. The *TBA* symbol is used to indicate that a task or a sub-task must be added at some future time to a use case in the UCM-UI model.

6.3.1.4 Task Performance (*TP)*

Task Performance (*TP*) metric specifies the scenarios that different types of users will require to perform a task. The most critical tasks are identified so that more time can be paid to them during usability testing later in the design process. Moreover, the designers can compare different scenarios to perform a task and decide the best scenario that is used for a specific type of users.

Definition: Task Performance is a study and examination of the task accomplishment procedure by different types of users.

Formula:

System	Use Case/Scenario	Comments
Users and Tasks		

Novice User A	
Task A	UCi-Sx[2]
Task B	UCj-Sy
Task C	UCk-Sy
Intermediate User B	
Task A	Uci-Sx
Task B	UCj-Sr
Task C	UCl-Sy
Expert User C	
Task A	UCi-Sy
Task B	UCk-Sr
Task C	UCn-Sy

Figure 6.5 Task Performance Form.

Figure 6.5 above shows the structure of a *TP* metric. It is important that input from different user groups (novice, intermediate, and expert)-users are obtained in order to complete the metric fully.

Scale: This metric is useful for systems where there are number of possible scenarios to perform a specific task and where the range of tasks that the user will perform is well specified. In these situations, the *TP* metric can be used to trade-off different scenarios, or to add scenarios depending on their value for supporting specific tasks. Most importantly, it is useful for multi-user systems to ensure that the tasks of each user type are supported.

6.3.1.5 Task Simplicity (*TS*)

Good design makes more frequent tasks easier and simpler. This metrics shows how well a user interface reduces the number of steps to complete more frequent tasks

Definition: Task Simplicity is an index of how well the expected frequencies of the tasks match their difficulty.

Formula:

[2] *UCi-Sx= this task is performed using Scenario x in Use case i.*

$$TS = \frac{F \times Ns}{100}$$

F = frequency of performing a task[3]

Ns = the number of steps to complete a task

Thus, the total task simplicity of a design for an interaction context could be computed by summing recursively over all tasks:

$$Total\ TS = \frac{\sum_{t=1}^{n} (F_t \times NS_t)}{100}\text{; for tasks } t_1, t_2, \ldots t_n$$

When more frequent tasks are always shorter than less frequent task the design is simpler and easy to use.

Scale: The design is perfect in terms of *TS* when every task is simply completed in one step, i.e. *TS* = 1. Longer, more complex tasks may legitimately need to perform more steps. As a consequence, Task Simplicity is an index of how well the expected frequencies of the tasks match their difficulty.

6.3.1.6 Use Case Complexity (UCC)

Use Case Complexity (*UCC*) metric is one measure of structural complexity of a use case. The *UCC* metric corresponds to the number of possible *execution paths* specified in the use case and therefore can be used to determine the number of tests required to obtain complete coverage of a use case. The assumption regarding this metric is that use cases with high *UCC* are more difficult to understand and consequently maintain than use cases with low *UCC*.

Definition: Use Case Complexity is defined as the number of decisions (or predicates) specified in a use case.

Formula: The *UCC* is calculated from a starting point (pre-condition) of the use case

Use Case Complexity (*UCC*) = *OR_F*+ *AND_F* – *AND_J* +1.

Where *OR_F* = number of OR-forks

AND_F = number of AND-forks

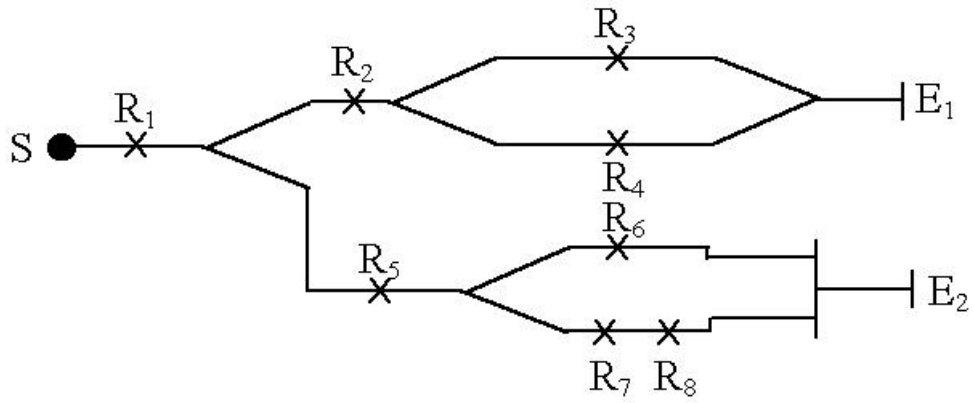
---

3 The frequencies of the task can be estimated by performing a survey.

AND_J = number of AND-joins

Scale: The complexity number is generally considered to provide a stronger measure of a use case structural complexity. A low complexity (1 <*UCC* < 10) contributes to a use case's understandability and indicates it is amenable to modification at lower risk than a more complex use case (*UCC* > 10).

Figure 6.6 a simple use case with a UCC of 3.



The *UCC* complexity of the use case shown in Figure 6.6 is 3 (OR_F)+ 0 (AND_F)– 1 (AND_J) +1 = 3+ 0- 1+ 1 = 3. There are three independent paths for the use case in Figure 6.6.

Path 1: S, R_1, R_2, R_3, E_1

Path 2: S, R_1, R_2, R_4, E_1

Path 3: S, R_1, R_5, (R_6+ R_7, R_8), E_2.

6.3.2 Basic Metrics

In this section four metrics defined in the literature are added to the UCM-UI metric suite. The four metrics are: Layout Uniformity, Task Visibility, Visual Coherence, and Task Effectiveness. Those metrics are added to our usability metrics suit based on their ability to be applied to our proposed model and their valuable interpretation for the interface design.

6.3.2.1 Layout Uniformity (*LU*)

Layout Uniformity (*LU*) metric defined by Constantine and Lockwood (1999) is based on the rationale that usability is held up by highly disordered or visually messy arrangement. *LU* measures only selected aspects of the spatial arrangement of the interface components without taking into account what those components are or how

they are used; it is neither task sensitive nor content sensitive. This metric assesses the uniformity or regularity of the user interface layout.

Definition: Layout Uniformity is a structural metric that gives a quick handle on how well visual components of interface are arranged.

Formula:

$$LU = 100 \times \left(1 - \frac{Nh + Nw + Nt + Nl + Nb + Nr - M}{6.Nc - M}\right).$$

$$M = 2 + 2 \times \lceil \sqrt[2]{Nc} \rceil$$

Nc= total number of visual components on screen, dialogue box, or other interface composite.

Nh, *Nw*, *Nt*, *Nl*, *Nb*, and *Nr* are respectively, the number of different heights, widths, alignments, left-edge alignment, bottom-edge alignments, and right-edge alignments of visual components.

LU is concerned with the interface appearance and can be useful to the designers who lack an eye for layout to know when a visual arrangement might be improved.

Scale: A review of well-designed dialogues suggests that, in general, a value of *LU* anywhere between 50% and 85% is probably reasonable (Constantine and Lockwood, 1999).

6.3.2.2 Task Visibility (*TV*)

Task Visibility (*TV*) metric defined by Constantine and Lockwood (1999) shows how many percent of necessary features (objects or elements) to complete a task or a use case are visible to the user and it is reduced when we get unnecessary features on the user interface. The visibility principle in *TV* is that user interfaces should show users exactly what they need to know or need to use to be able to complete a given task.

Definition: Task Visibility is a metric grounded in the visibility principle. It measures the fit between the visibility of the features and the capabilities needed to complete a given task or a set of tasks.

Formula:

$$TV = 100 \times \left(\frac{1}{S\_total} \times \sum_{\forall i} Vi\right)$$

S_total=Total number of performed steps to complete the use case

Vi=Feature visibility (0 or 1) of performed step *i*[4].

The formula for *TV* is expressed as a percent of the total number of steps. *TV* reaches a maximum of 100% when everything needed for a step is visible directly on the user interface as seen by the user at that step. Task Visibility takes into account only one aspect of the concept WYSIWYN or What You See Is What You Need. It ignores whether things that are not needed are also found on the user interface. In principle, we could reduce Task Visibility whenever unused or unnecessary features are incorporated into the user interface.

Scale: *TA* ranges from 0% (poor) to 100% (good). However, high-security interfaces should correlates with very low (zero) visibility.

6.3.2.3 Visual Coherence (*VC*)

Visual Coherence measures how well a user interface keeps related components together and unrelated components apart (Constantine and Lockwood, 1999). More specifically, it is a semantic or content-sensitive measure of how closely an arrangement of visual components matches the semantic relationship among those components.

Definition: Visual Coherence is the ratio of the number of closely related pairs of visual elements to the total number of enclosed pairs.

Formula:

$$VC = 100 \times \frac{G}{N(N-1)/2}$$

G = the number of related pairs in the group.

N = the number of visual components in the group.

[4] How to count performed steps and allocate a visibility value to them is defined by some rules in the (Constantine and Lockwood, 1999). Visibility Score *Vi* for each enacted step *i* is: 0 if recall is needed (Hidden); 0.5 if it is available but needs to be exposed (Exposing), or if the interaction context is changed (Suspending); 1 if visible (Direct).

Total Visual Coherence of a design for an interaction context could be computed by summing recursively over all the groups at each level of grouping:

$$VC = 100 \times \left(\frac{\sum_{\forall k} Gk}{\sum_{\forall k} Nk(Nk-1)/2} \right)$$, where k is a group of component.

with $$Gk = \sum_{\forall i,j | i \neq j} R_{i,j}$$

Scale: The overall Visual Coherence of each deign depends on the semantic relatedness among the features or components contained or enclosed with in each of its visual group. Based on the principle that well-structured interfaces group together components that represent closely related concepts, Visual Coherence reflects important and fundamental aspects of user interface architecture that strongly affect comprehension, learning, and use.

6.3.2.4 Task Effectiveness (*TES*)

Task Effectiveness (*TES*) comes from MUSiC project (Bevan, 1999). It relates the goals or sub-goals of using the system to the accuracy and completeness with which these goals can be achieved.

Definition: Task Effectiveness with which a user uses a product to carry out a task is comprised of two components: the quantity of the task the user completes, and the quality of the goals the user achieves. *Quantity* is a measure of the amount of a task completed by a user. It is defined as the proportion of the task goals represented in the output of the task. *Quality* is a measure of the degree to which the output achieves the task goals. As Quantity and Quality are both measured as percentages, Task Effectiveness can be calculated as a percentage value

Formula:

$$TE = \frac{Quantity \; x \; Quality}{100}$$ [5]

Quantity = Percent of task completed.

Quality = Percent of goals achieved.

[5] How to specify and measure quality and quantity is discussed in a great detail in (Bevan 1995).

It is sometimes necessary to calculate effectiveness for a number of sub-tasks, for instance this might be the individual element in a drawing task. The average effectiveness across sub-tasks is a useful measure of the product's capabilities, but may not reflect the effectiveness of the final task output. For instance, if the user were unable to save the final drawing, the overall effectiveness would be zero. Similarly, the effectiveness may be reduced if the final drawing contains additional unwanted elements.

Scale: A *TES* value of less than 25% should be considered completely unacceptable because it means that the users were unable to achieve at least half of the task goals with at least 50% acceptability. If *TES* reaches 50%, then use of the product can be considered a reasonably effective way for the users to achieve their task goals. When *TES* reaches 100%, the product has enabled the users to attain all the task goals perfectly**.**

6.4 An Illustrative Example

In this section, we illustrate via an example, a Movie Recommender System (MRS) software of a handheld computer (PDA); how to build a UCM-UI model. The MRS software provides recommendations for movies based on preferences of other users. The user of MRS has the option of getting recommendations for movies of a certain genre, new movies of certain genre, and of movies that are similar to some movie in their theme. The Requirement Specifications (RS) of the Movie Recommender System are:

– RS1: The user gets recommendations (list of movies) for the Latest movies by looking for a All, Action (A), Drama (D),.or Comedy (C).

– RS2: The user gets recommendations (list of movies) by searching for a All or specific type.

– RS3: the user is able to refine his search until he is satisfied with the results.

– RS4: The user creates his own favourite movie list.

– RS5: The user rates a movie.

On choosing any one of RS1 and RS2, the user can get movie detailed information of a selected movie. For this example there are five major use cases illustrated in Table 6.2. Figure 6.7 present the CUCM and PUCM of the MRS software.

Use Case	Scenarios
UC1: List Latest movies (8 scenarios)	***S1-S4:*** 4 scenarios when the user chooses one type (All, A, D, C) from latest movie list, view the list and end the software. ***S5-S8:*** 4 scenarios when the user chooses one type (All, A, D, C) from latest movie list, view the list and returns to beginning of the software.
UC2: Search for a movie (12 scenarios)	***S1-S4:*** 4 scenarios when the user searches one type (All, A, D, C) from search menu view the list and end the software. ***S5-S8:*** 4 scenarios when the user searches one type (All, A, D, C) from search menu, view the list, refine the search, and view the refined list and end the software. ***S9-S12:*** 4 scenarios when the user searches one type (All, A, D, C) from search menu, view the list and refine the search and just quit

UC3: To See (2 scenarios)	***S1***: User adds a movie to the "To See" list. ***S2***: User views the "To See" list.
UC4: View a movie detailed information (1 scenario).	***S1***: User views detailed information of a selected movie.
UC5: Movie Rating (1 scenario)	***S1***: User views the rating of a movie.

Table 6.2 Use cases and scenarios of the MRS software.

We are interested to show the usage of the UCM-UI metrics defined in Section 6.3.1 and 6.3.2.

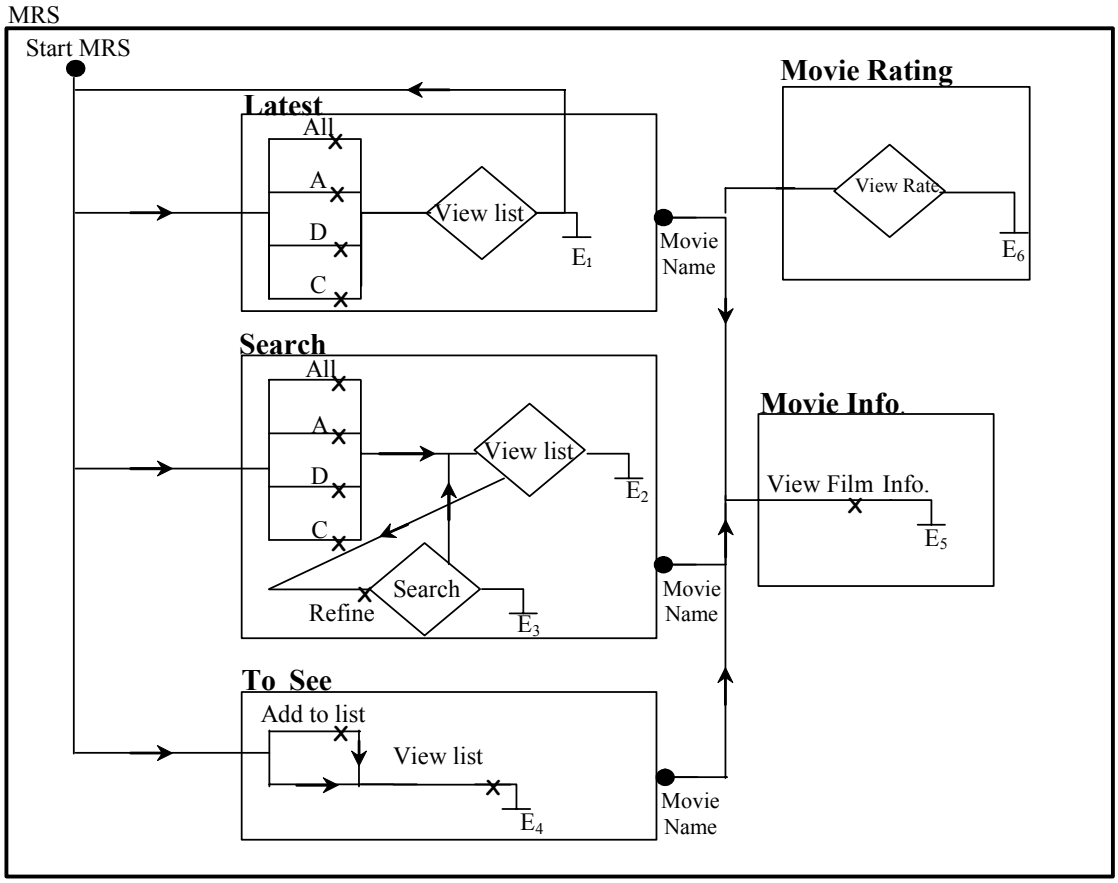


Figure 6.7 CUCM of the MRS software.

6.4.1 Task Analysis

Task Analysis (*TA)* metrics is conducted to understand the MRS software and the information flows within the user interface. *TA* is useful to identify the pre-conditions and how use cases are interrelated to perform a task. For the sake of simplicity, the *TA* metrics in Figure 6.8 illustrate only analysis of two tasks.

$Task_i$	Why this task?	What happened before?	What is done Or to be done?	What happens next?	How to perform this task?
$Task_1$	Add a movie to "TO See" List	Start MRS	UC3-S1	UC5-S2	Start MRS, Add to see list, view to see list, E_3
$Task_2$	Rate a Movie	UC1, UC2, or UC3	UC5-S2	E_4	Insert Movie name, rate, E_4

Figure 6.8 *TA* metric for two tasks in MRS software.

6.4.2 UCMs Model-Consistency

The UCMs Model-Consistency metric ensures that the requirements captured by the use cases are not in conflict with each other, so a consistent UCMs model can be synthesized from the use cases. After filling the UCMs Model-Consistency metric form in Figure 6.9, we found that there was inconsistency in (*view list*) in the (To See) use cases with (*view list*) in *UC1* and *UC2*. In *UC1* and *UC2* (*view list*) is a stub while in *UC3* (*view list*) is a responsibility. Thus, the CUCM of the MRS System is improved in Figure 6.10.

UC_1 (Latest Movies)	UC_1			
UC_2 (Search for a Movie)	$\leq$	UC_2		
UC_3 (To See)	$\not\subset$	$\not\subset$	UC_3	
UC_4 (View Movie information)	-	-	-	UC_4
UC_5 (Rate a Movie)	-	-	-	-

Figure 6.9 UCMs Model-Consistency metric for MRS.

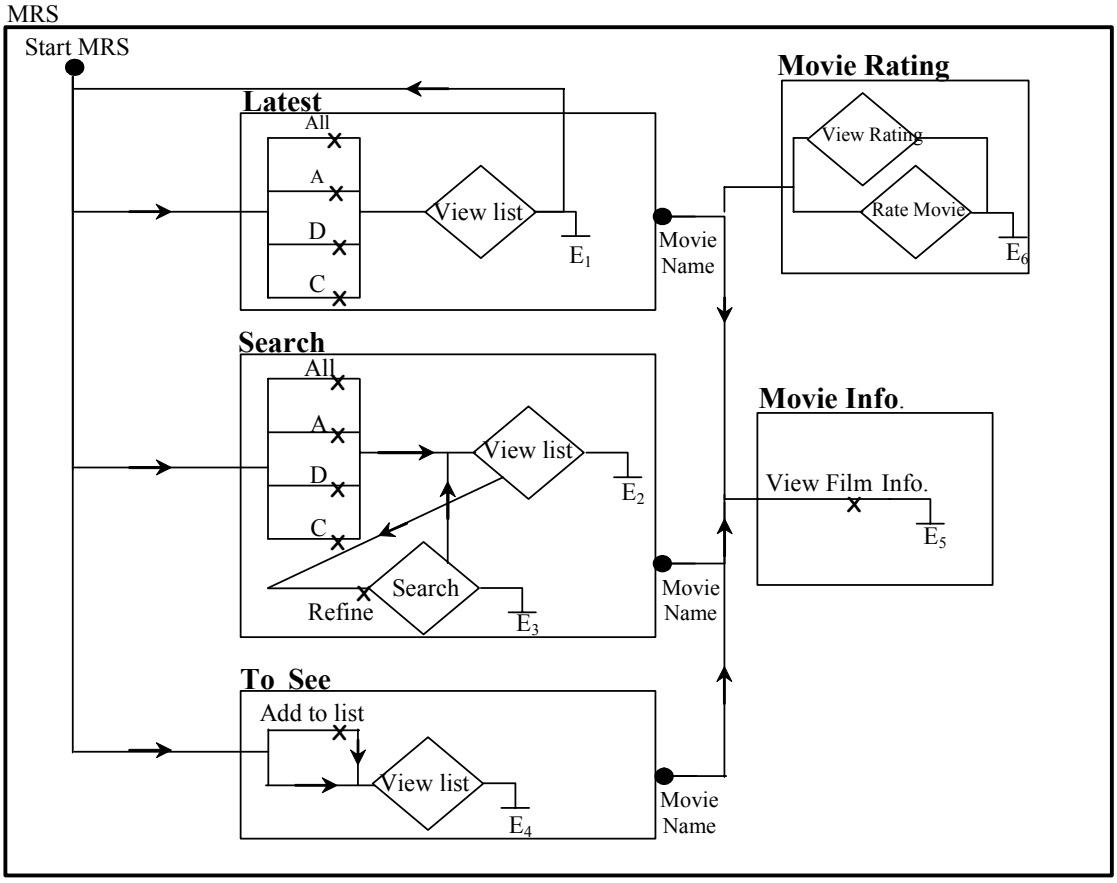


Figure 6.10 Improved CUCM of the MRS software.

6.4.3 UCMs Model-Completeness

To ensure completeness, all tasks (including their sub-tasks) in the requirement specifications of the required system must be covered by at least one use case following the correct order of task performance. For example, the completeness of the following tasks can be examined:

Task 1: User selects latest Comedy movies and gets movie recommendations, then User views details of a recommended movie

Task 2: User enters an actor name and gets movie recommendations, then add the movies to his "TO SEE" list.

Task 3: User loads the latest Drama movies list and rate as many movies as he can.

User Tasks	UC1	UC2	UC3	UC4	UC5
Task 1: User views latest list of Comedy movies					

Sub task 1.1: User selects latest comedy movies	Y			
Sub task 1.2: List movie recommendations	Y			
Sub task 1.3: User views information on Movie *Mi*			Y	
Task 2: User search for a Action movie for a specific Actor and adds the movie to his "To See" list				
Sub task 2.1: User selects search for Action movies		Y		
Sub task 2.2: User enters the Actor name		Y		
Sub task 2.3: List of movie recommendations		Y		
Sub task 2.4: User view information on Movie *Mi*			Y	
Sub task 2.5: User adds the movie to his "To See" list		Y		
Task 3: User loads the latest Drama movies list and rate as many movies as he can				
Sub task 3.1: User selects latest drama movies	Y			
Sub task 3.2: List of movie recommendations	Y			
Sub task 3.3: User rates movie *Mi*				Y

Figure 6.11 UCMs Model-Completeness metric for MRS for three tasks.

To ensure the UCMs model-completeness we need to examine all possible task that can be performed by in the user interface of the MRS software. However, examining all possible tasks can be time consuming. Therefore, user interface designers are encouraged to apply the completeness metrics for important and complicated use cases.

6.4.4 6.4.4 Task Performance (*TP)*

In this metrics, the most critical tasks are identified so that more time can be paid to them during usability testing later in the design process. The designers can also compare different scenarios to perform a task and decide the best scenario that is used for a specific type of users. To complete this metric, we must obtain input from different user groups (novice, intermediate, expert users). In our case, this usability test was not performed, however, Figure 6.12 shows a sample of the *TP* metric form.

System:	Use Case/Scenario	Comments
Users and Tasks		
Novice User A		
User views latest list of Comedy movies		
User search for a Action for a specific Actor		
User search for latest dram movie by the Actor x and adds the movie to his “To See” list		
Intermediate User B		
User views latest list of Comedy movies		
User search for a Action for a specific Actor		
User search for latest dram movie by the Actor x and adds the movie to his “To See” list		
Advance User C		
User views latest list of Comedy movies		
User search for a Action for a specific Actor		
User search for latest dram movie by the Actor x and adds the movie to his “To See” list		

Figure 6.12 Sample of the Task Performance metric for MRS.

6.4.5 Task Simplicity (TS)

In this metric we’ll try to make more frequent tasks easier and simpler. The Task Simplicity formula is:

$$TS = \frac{F \times Ns}{100}$$ [6],

The PUCM[7] of the MRS software is shown in Figure 6.13. The paper prototype of the PUCM is demonstrated in Figure 6.14. The *TS* for the five main tasks of the MRS software are calculated in Table 6.3.

[6] F = frequency of performing a task, N*s* = the number of steps to complete a task

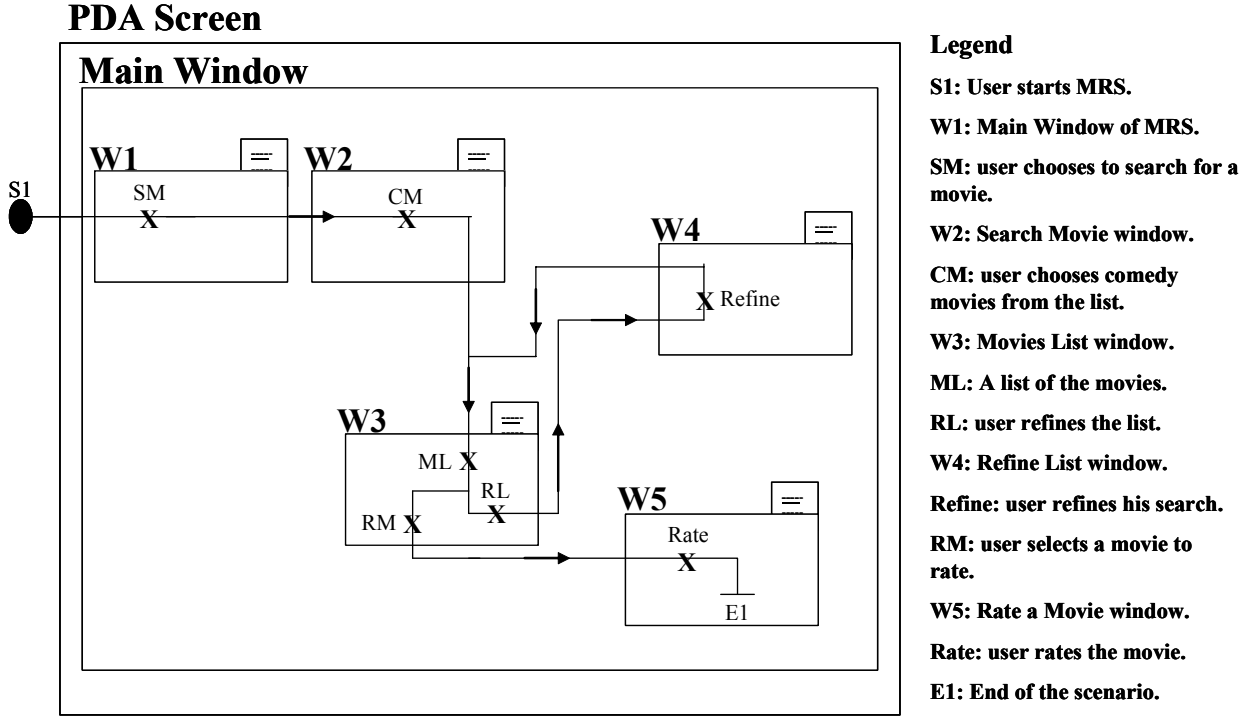


Figure 6.13 PUCM of the MRS software.

[7] PUCM = Physical Use Case Map is the Task and Presentation UCM explained in detailed in Chapter 4.

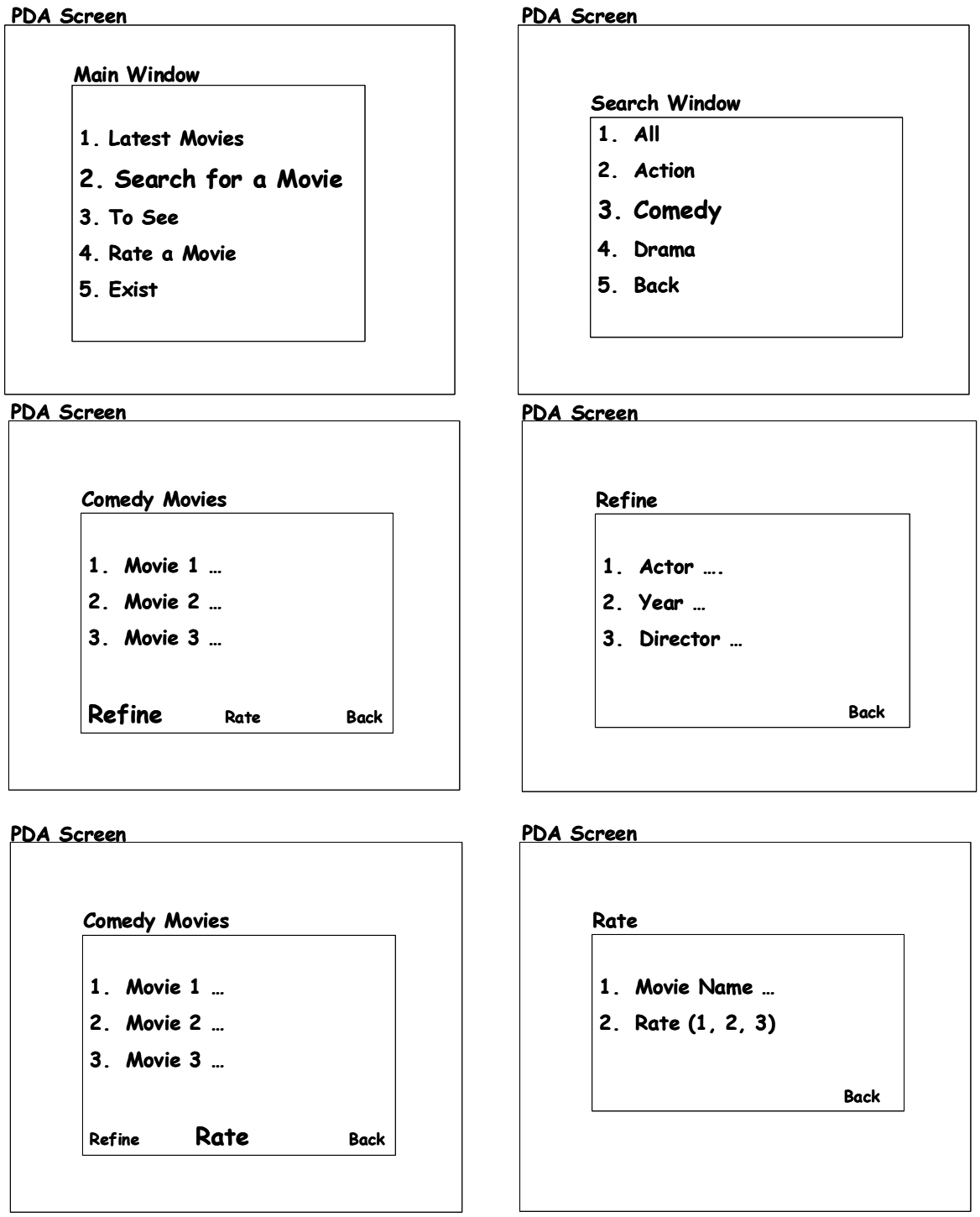


Figure 6.14 Paper prototype of PUCM of Figure 6.13.

Task No.	Description of the Task	Frequency	Steps No.
Task 1	The user gets recommendations for the Latest movies by looking for a genre or specific type.	30	2
Task 2	The user gets recommendations for movies by searching for a genre or specific type.	25	2
Task 3	The user is able to refine his search.	20	4
Task 4	The user has his own favourite movie list where he can also add a movie to the list.	15	4
Task 5	The user rates a movie.	10	4

Table 6.3 Tasks frequencies and number of steps for the MRS software.

TS = (30*2 +25*2 +20*4 + 15*4 +10*4)/100

= 2.9

To improve our design, we try to simplify each task. Task 1 and Task 2 are simple since each task is performed in two steps and cannot be performed in less number of steps. However, the number of steps to perform Task 3, Task 4, and Task 5 can be reduced. Figures 6.15 and 6.16 show the improved user interface design. Table 6.4 indicates the number of steps to perform each task in the improved design.

Task No.	Description of the Task	Frequency	Steps No.
Task 1	The user gets recommendations for the Latest movies by looking for a genre or specific type.	30	2
Task 2	The user gets recommendations for movies by searching for a genre or specific type.	25	2
Task 3	The user is able to refine his search	20	3
Task 4	The user has his own favourite movie list where he can also add a movie to the list.	15	2
Task 5	The user rates a movie.	10	2

Table 6.4 Tasks frequencies and number of steps of the improved design.

TS = (30*2 +25*2 +20*3 + 15*2 +10*2)/100

= 2.2, which is probably quit improvement at the end of the day.

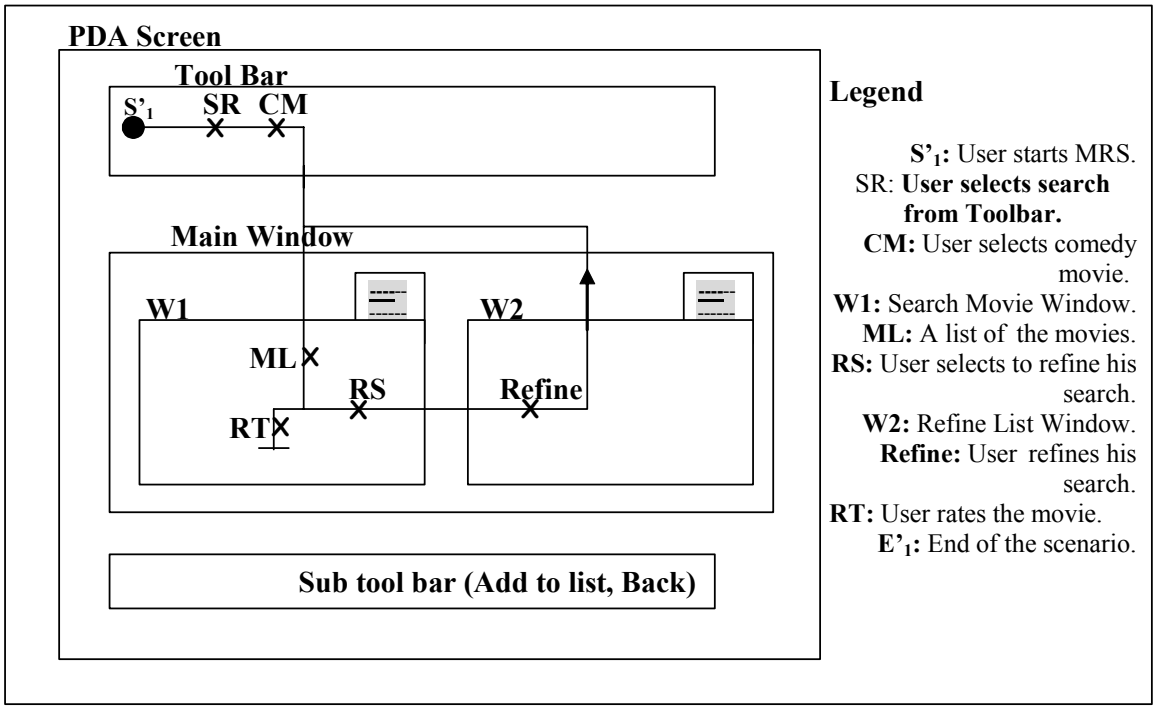


Figure 6.15 Improved PUCM of the MRS system.

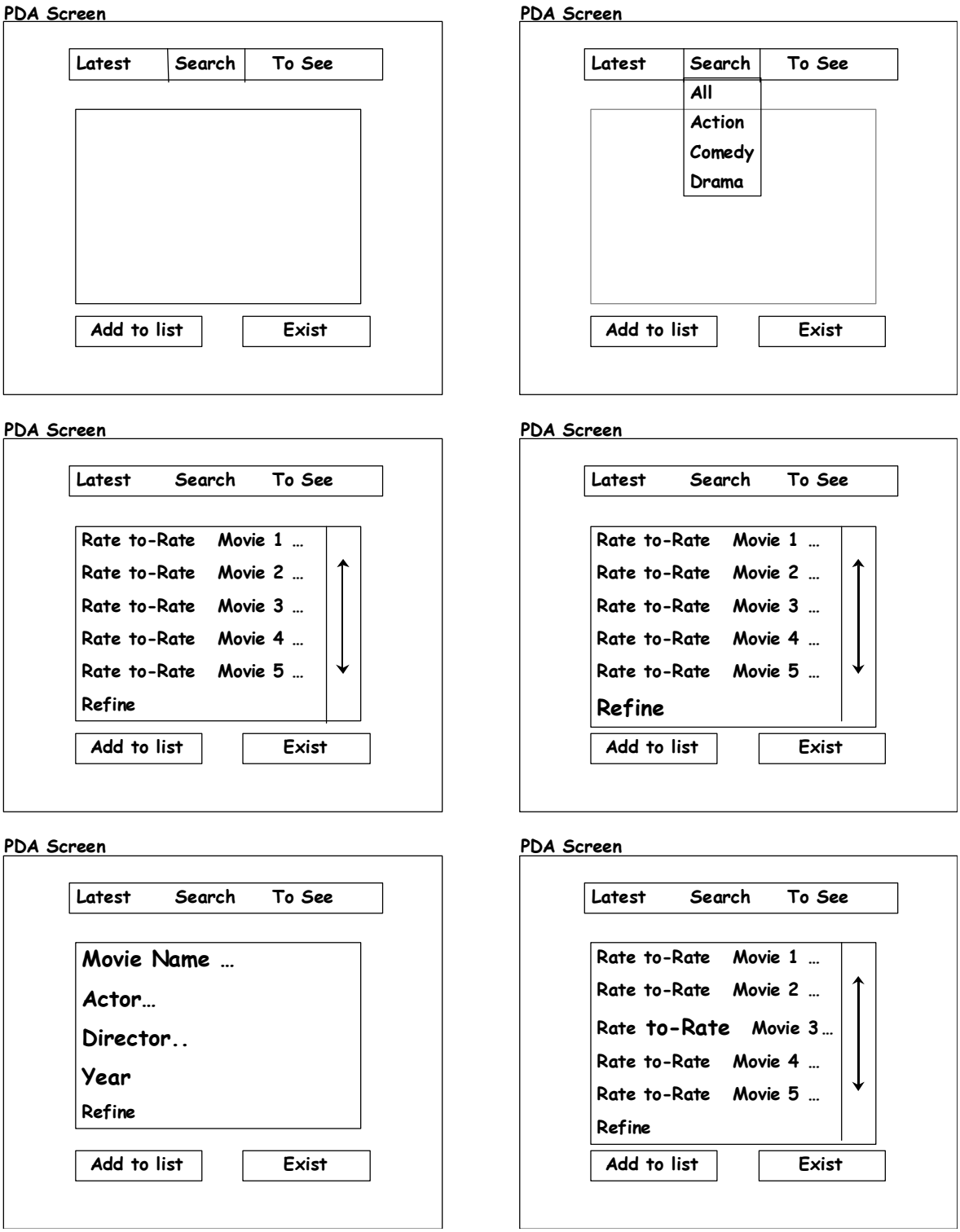


Figure 6.16 Improved paper prototype of PUCM Figure 6.15.

6.4.6 Use Case Complexity (*UCC*)

The complexity number is generally considered to provide a stronger measure of a use case structural complexity. For example, the *UCC* complexity of *UC5* (Rate a movie) shown in Figure 6.17 is 2 (OR_F)+ 0 (AND_F)– 0 (AND_J) +1 = 2+ 0- 0+ 1 = 3; i.e., there are three independent paths for *UC5* which contributes to a use case's understandability and indicates it is amenable to modification at lower risk. During analysis of the use cases, any use case with a *UCC*> 10 should be simplified since it is more difficult to understand and consequently maintain.

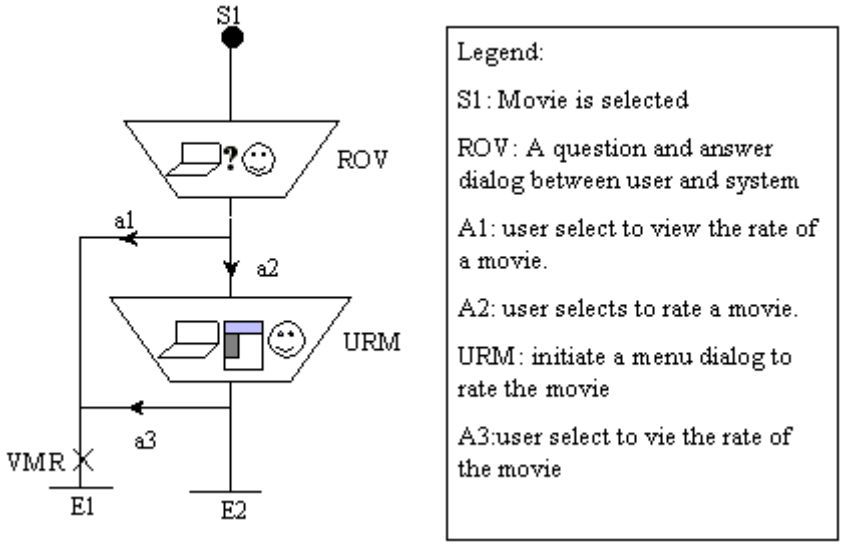


Figure 6.17 CUCM for *UC5* (Rate a movie).

6.4.7 Layout Uniformity (*LU*)

Layout Uniformity goes up when visual components are lined up with one another and when there are not too many different sizes of components. The role of *LU* metric can be applied to user interface of Figure 6.16 as follows.

$$M = 2 + 2 \times \lceil \sqrt[2]{Nc} \rceil$$

$$= 2 + 2 \times \left| \sqrt[2]{6} \right| = 6.898$$

$$LU = 100 \times \left(1 - \frac{Nh + Nw + Nt + Nl + Nb + Nr - M}{6.Nc - M}\right). = 79.03\%.$$ As Constantine and Lockwood (1999) discussed, a value of LU anywhere between 50% and 85 % is probably reasonable.

6.4.8 Task Visibility (*TV*)

Task Visibility can be evaluated for individual use cases or for extended task scenarios that might incorporate any number of use cases. For the use cases expressed in Figures 6.15 and 6.16 the visibility of the enacted steps to complete the use case are calculated in Table 6.5.

Enacted Step	Type	Visibility
Open **Search** menu	Exposing	0.5
Open Comedy Movie List	Suspending	0.0
Click on **Refine** button	Exposing	0.5
Fill the Form	Direct	1.0
Click **search** button	Exposing	0.5
Open drop-down list	Exposing	0.5
Rate the selected movie	Direct	1.0
Click **Exist** to close dialogue	Suspending	0.0
	Total =	4.0

Table 6.5 Visibility of the enacted steps.

Since there are eight steps in this enactment, the task visibility *TV* = 50 %. This indicates that for the above use case, 50 % of the icons needed for a step are visible on the user interface as seen by the user at that step.

6.4.9 Visual Coherence

Visual Coherence is a semantic or context-sensitive measure of how closely an arrangement of visual components matches the semantic relationships among those components. The *VC* of the tool bar in Figure 6.16 is calculated as follows:

$N = 3$ components.

$$VC = 100 \times \frac{G}{N(N-1)/2}, \text{ with } \qquad G = \sum_{\forall i,j | i \neq j} R_{i,j}$$

$$= 100 \times \frac{1+1+0}{3 \times 2/2} = 66.66\%$$

To evaluate Visual Coherence, the list of concepts is scanned to determine with which concept each visual component is most closely related. If two components are both determined to be most closely related to concepts in the same cluster, then they are considered to be substantially related and are assigned as $R_{i,j}$ =1 if they are associated from different clusters, then $R_{i,j}$ = 0.

The visual coherence of the tool bar in Figure 6.16 is 66.66%, but to get a real feeling of the coherence of the interface we need to compute the overall visual coherence of the interface starting from the innermost visual groups, and then repeat the same thing for the next level outward, until the complete interaction context is covered.

6.4.10 Task Effectiveness (*TES*)

Task Effectiveness *TES* is a metric that gives a measurement of how well a user achieves the goals of an evaluation task, irrespective of the time needed or the difficulties encountered.

$$TE = \frac{Quantity\ x\ Quality}{100}$$

Where Quantity is a measure of the amount of a specified task that a user attempts. It is defined as the proportion of the task goals represented in the output of a task and Quality is a measure of how good the task goals represented in the output are compared to their ideal representation. It is defined as the degree to which the task goals represented in the output have been achieved.

Before the usability sessions take place, the evaluation team must identify the goals of the evaluation task and how the users will represent them in the task outputs. Then the designer analyzes the task output and measures:

– Quantity (how many goals the user attempts)

– Quality (how successfully the user completes those goals)

of the task output that the user produces and thus can calculate the Task Effectiveness. For this example case, this usability test was not performed and therefore the *TES* was not calculated.

6.5 Conclusions

In this chapter, we build UCM-UI metrics suite to predict usability early in the requirements phase. The UCM-UI Metrics Suite has evolved considerably using metrics from the literature and adding new metrics to it. Ten metrics are included that together cover assortment of measurements likely to be significant to designers seeking to improve the usability of their software. This metrics suite complement rather than replace empirical evaluation of user performance and satisfaction. An illustrative example on how to predict usability using the UCM-UI metrics suite was also presented in this chapter.

The UCM-UI metrics suite as shown in the example allows interface designers using the SUCRE framework to predict and improve the interface quality, to control and improve the requirement processes, and to decide on the acceptance of the interface. However, the UCM-UI Metrics Suite does not guarantee usability and interface designers should be wary of allowing metrics to cloud their judgment. Quantitative comparisons are no substitute for thought, careful design, systematic review, and judicious testing.

One of the important factors we would like to explore is the effectiveness of metrics initiatives and in particular how the numbers are put to use, and usability prediction is certainly not expectation to that. Utilized inappropriately, predictive metrics can take on exaggerated significance and may come to dominate design decisions. For example, the effect of immediate feedback on design quality has been investigated at the university of technology, Sydney (Noble and Constantine, 1997]. Given instance numeric feedback on their layouts, designers can sometimes unconsciously work to maximize their score rather than derive the best design. The result can be good looking numbers but poorer interfaces when all factors are taken into account.

Chapter 7

Towards Formal Specifiations

Abstract

The process of going from informal requirements to a high-level formal specification is difficult and nontrivial task. Engineers often using informal requirements complain of the lack of straightforward transition to formal requirements. This chapter points toward avenues to build formal requirements from UCM-UI. One way to bridge the gap separating requirements and use cases from more detailed views is to link the UCM-UI concepts to UML and UML*i* for the modeling of complex user interfaces. Another avenue that smoothen the transition to a formal specification language is to define a method to translate UCM-UI to LOTOS automatically. LOTOS is an appropriate formalism here because it supports many UCM and UCM-UI concepts directly, and it allows a formal verification and verification of requirements and specifications of interactive system. Finally, the generation of a linear textual form from UCM-UI models can be used as a front end for other software engineering tools and it enables the validation of requirements and facilitates the transition from requirements to design. UCMs are already represented in XML by the tool UCMNav according to the XML Data Type Definition for UCMs. This chapter presents how to generate XML and XUL files from UCM-UI models.

7.1 Introduction

The process of going from semi-formal requirements to a high-level formal specification is a research subject where much work has been done. However, engineers are faced with challenges when deriving formal specification from the informal/semi-formal requirements. The UCM-UI is semi-formal requirement models that facilitate abstract thinking at the right level of detail for architectural design since UCM-UI allows the engineers to model the essential structure of the interface tasks without hidden and premature assumptions about design details. After number of refinement phases of the UCM-UI models with the user more detailed views and models in other modeling/specification languages may be derived. This will certainly improve the validity of the UCM-UI models and detect undesirable interactions early in the design cycle. The UCM-UI is used to model the interrelationships among user interface scenarios, mapping the overall structure of the tasks to be supported by interface.

The proposed UCM-UI model can be used to bridge the gaps between: (a) the user interface requirements and its underlying functional application, and (b) between requirements and design. This chapter investigates how UCM-UI fits in many different software engineering methodologies and design processes. Section 7.2 present how UCM-UI can be used to extend the semantics and notations of UML for the modeling of interactive systems which will reduce the gap between user interface requirements and detailed requirements of the system. To allow a formal verification and verification of requirements and specifications of interactive system Section 7.3 presents a method to translate UCM-UI to LOTOS automatically. Moreover, Section 7.4 investigates the possibility to produce a textual linear form for UCM-UI expressed in XIML and add the UCM-UI notations to the UCMNav tool.

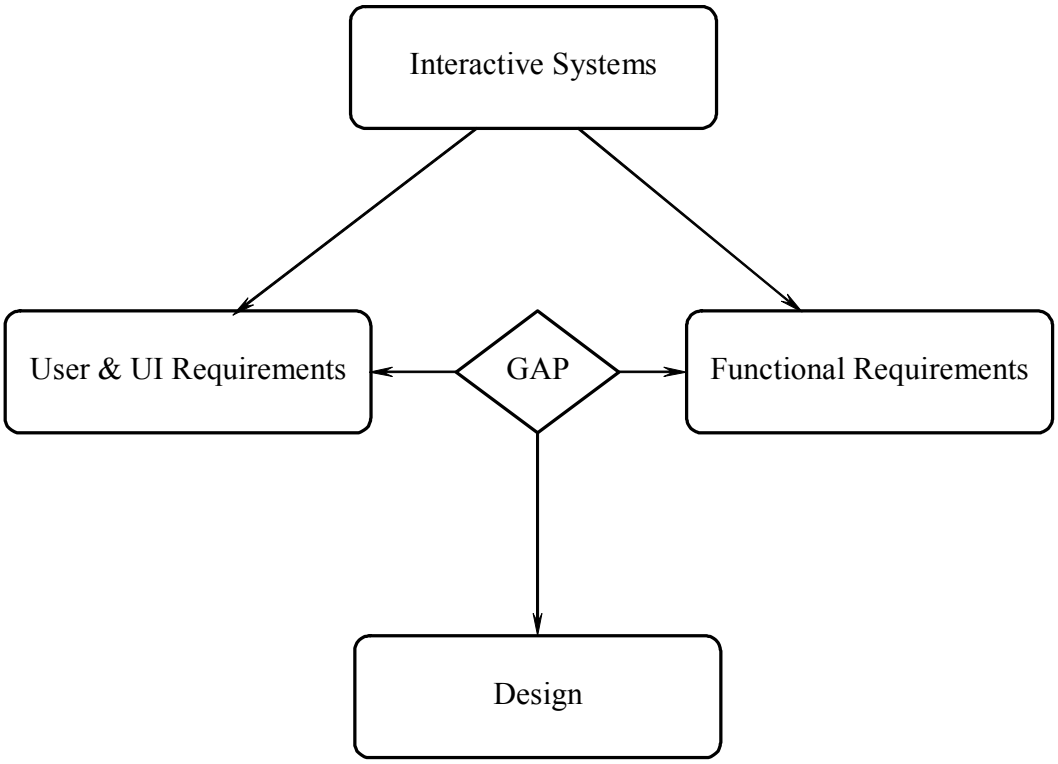


Figure 7.1 The Gap in requirements and design of interactive systems.

7.2 Avenue 1: the Extension of UML*i* with UCM-UI

The Unified Modeling Language (UML) is a notation for creating software application designs in object-oriented manner. Developers should be able to model complete applications using UML. There are several necessary concepts appear to be absent from UML, but present in the UCM notations. In particular, UCMs allow scenarios to be mapped to different architectures composed of various component types. The UCMs notation supports structured and incremental development of complex scenarios at a high level of abstraction, as well as their integration. UCMs specify variations of run-time behavior and scenario structures through sub-maps 'pluggable' into placeholders called stubs. Amyot and Mussbacher (2001) presented how UCM concepts could be used to extend the semantics and notations of UML for the modeling of complex reactive systems. Absolutely, adding a 'UCM view' to the existing UML views can help bridging the gap separating requirements and use cases from more detailed views (e.g. expressed with interaction diagrams and state-chart diagrams).

On the other hand, UML suffers from its lack of support for modeling user interfaces (Fabio, 2001; Srdjan, 1998). For example, class diagrams are not entirely suitable for modeling UI presentation. As a result of these difficulties, research has taken place with a view to improving the effectiveness of UML for UIs. The Unified Modeling Language for Interactive Applications (UML*i*) is a research project that has been developed in the Information Management Group of the University of Manchester since 1998. UML*i* extends UML, to provide greater support for UI design. It aims to show that using a specific set of UML constructors and diagrams it is possible to build declarative UI models.

This section builds on top of the research developed on the extension of UML with UCMs concepts (Amyot and Mussbacher, 2001), and the extension UML*i* (Pinheiro da Silva, 2000; Pinheiro da Silva and Paton, 2003).

7.2.1 Brief introduction to UML*i*

UML*i* is a UML extension for modeling interactive applications. UML*i* makes extensive use of activity diagrams during the design of interactive applications. Well-established links between use case diagrams and activity diagrams explain how user requirements identified during requirements analysis are described in the application

design. UML*i* provides two additional facilities for user interface design: (1) a new diagram for modeling UI presentations called a *user interface diagram*. (2) a new set of activity diagram constructors for modeling UI behavior.

Each user interface diagram is composed of *FreeContainer*, may contain *Containers, Editors, Displayers, Inputters*, and *ActionInvokers*. Table 7.1 illustrates the components of the user interface diagram. The user interface diagram constructors are *Interaction-Classes* that are specialized UML *Classes*. The instances of these Interactive-Classes are the *interaction objects* (or *widgets*).

Component	Symbol	Comments
One FreeContainer		is a top-level "window"; FreeContainers may contain *Containers* , *Editors* , *Displayers* , *Inputters* and *ActionInvokers*;
Displayers		Displayers are responsible for sending visual information to users
Inputters		Inputters are responsible for receiving information from users
Editors		Editors are interaction objects that are simultaneously Inputters and Displayers.
ActionInvokers		ActionInvokers are responsible for receiving information from users in the form of events
Containers		Containers may contain other Containers, Editors, Displayers, Inputters and Action Invokers

Table 7.1 UML*i* User Interface Diagram Components.

The new set of activity diagram constructors for modeling UI behavior in UML*i* provides: (1) new activity diagram *Pseudo-State s* for modeling common UI behavior called *selection states*; (2) new stereotypes for modeling object flows of UI objects. These stereotypes identify the *interaction object flows* and they are responsible for specifying the collaboration among interaction objects, and the collaboration between interaction objects and objects from the domain.

The UML*i* user interface diagram introduced for modeling abstract user interface presentations simplifies the modeling of the use of visual components (widgets). Additionally, the UML*i* activity diagram notation provides a way for modeling the relationship between visual components of the user interface and domain objects. Finally, the use of selection states in activity diagrams provides a

simplification for modeling interactive systems (Pinheiro da Silva, 2000; Pinheiro da Silva and Paton, 2003).

7.2.2 Linking UCM-UI Concept to UML and UML*i*

A UCM-UI model is characterized by the dialogs, the user tasks it supports, and the presentations of information that it generates for capturing and documenting the user interface requirements. The description of these aspects could be represented by UML and UML*i*. However, the effort to link the presentation aspects; the Physical Use Case Map (PUCM) to UML and UML*i* model does not seem to be justified because we obtain models that describe features that can be easily translated to paper prototypes and be understood by direct inspection of the implemented user interface. Whereas in the case of user interface dialogues and task representations, the Conceptual Use Case Map (CUCM), there are aspects that are more difficult to grasp in an empirical analysis because when users navigate in an application they follow only one of the many possible paths of actions. In addition, CUCM models are highly concurrent systems because they can support the use of multiple interaction devices, they can be connected to multiple systems, they can support the performance of multiple tasks, and they often can support multi-user interactions.

The UCM-UI concepts can be linked to UML and UML*i* by taking the advantage of similarities between the CUCM in UCM-UI and UML and the new set of activity diagram constructors in UML*i*.

Activity diagrams share many concepts with UCMs, in general, as described in (Amyot and Mussbacher, 2000). They have common constructs and even the notations are alike, to some extent. A UCM can effectively be described in terms of activity diagrams without any difficulty. The suggested uses of these notations are however slightly different. UCMs and Activity diagrams both target the modeling of system-wide procedures and scenarios, but activity diagrams focus on internal processing, often found in business-oriented models (e.g. workflows), whereas UCMs are also concerned with external (asynchronous) events, which are essential to the modeling of systems that are reactive in nature. Table 7.2 presents, through a simple mapping, how Activity Graphs meta-classes already support many UCM concepts.

UCM	Concept Corresponding Meta-classes
Map	Activity-Graph (from Activity Graphs), a child class of State-Machine.
Path Element	State-Vertex (from State Machines), the parent class of State and Pseudo-State, which is also similar to a node in a graph.
Start Point	Simple-State (from State Machines), a State without nested states.
End Point	Simple-State (from State Machines), a State without nested states.
Action Element	Action-State (from Activity Graphs), an atomic action. In UML, an Action-State is a Simple-State with an entry action whose only exit transition is triggered by the implicit event of completing the execution of the entry action. This is similar to a UCM responsibility.
Responsibility	Associated with Action-State (from Activity Graphs), an atomic action referenced by an Action Element (in UCM terms).
Continuation Element	State-Vertex (from State Machines), the parent class of Pseudo-State and (indirectly) of SubActivityState.
OR	Pseudo-State (from State Machines), of kind choice for an OR-fork and of kind junction for an OR-join.
AND	Pseudo-State (from State Machines), of kind fork for an AND-fork and of kind join for an AND-join.
Stub	Composite-State (from State Machines), which may contain sub-machines.
Static Stub	SubActivityState (from Activity Graphs), which may reference only one sub-Activity-Graph, just like a UCM static stub contains only one plug-in.
Dynamic Stub	Dynamic Stub is a new child class of Composite-State. Dynamic stubs may reference possibly many sub-maps, and they handle a binding relationship for each reference (instead of only one as in Sub-activity-State and Sub-machine-State). A Selection-Policy, which is an abstract Relationship, is associated to each dynamic stub.

Table 7.2 Mapping UCM concepts to Activity Graphs meta-classes (Amyot and Mussbacher, 2001).

For UCM-UI there are two concepts to be mapped to UML, the dialog concept, and the task concept. A dialog is a stub to which multiple Maps (plug-ins) are bound and to which a Selection Policy is associated. A selection policy instance should be defined as a potentially reusable object rather than as a mere attribute. As for dynamic stubs, extensions to the UML meta-classes appear necessary. Figure 7.2 proposes a solution with Dialog as a new child class of Composite-State. Dialog may reference possibly many sub-maps, and they handle a binding relationship for each reference, instead of only one as in Sub-activity-State and Sub-machine-State. A *Selection-Policy*, which is an abstract *Relationship*, is associated to each dialog. Table 7.3 presents, how Activity Graphs meta-classes can support many the dialog concepts

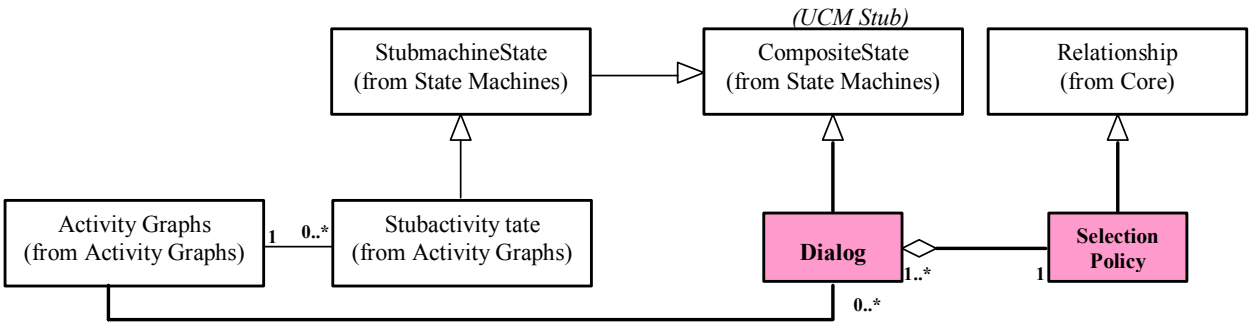
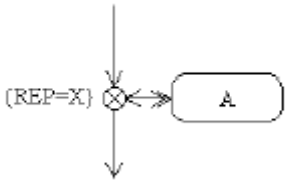


Figure 7.2 Extending activity graph with Dialog concept.

UCM-UI	Concept Corresponding Meta-classes
Dialog	*Dialog* is a new child class of Composite-State. Dialog may reference possibly many sub-maps, and they handle a binding relationship for each. A *Selection-Policy*, which is an abstract *Relationship*, is associated to each dialog

Table 7.3 Mapping dialog notation to UML Activity Graphs meta-classes.

Activity diagrams in UML*i* extend activity diagrams in UML. In fact, UML*i* provides a notation for a set of macros for activity diagrams that is used to model behavior categories usually observed in user interfaces: optional, order-independent, and repeatable behaviors. Using these macro notations, activity diagrams in UML*i* can cope better with the tendency that activity diagrams have to become complex even when modeling the behavior of simple user interfaces. Table 7.4 presents, through a simple mapping, how the new UML*i* set of macros support UCM-UI tasks concepts.

UCM-UI	UML*i*	Concept Corresponding Meta-classes
- Iteration A(*), and - Finite Iteration(s) A(X)		The repeatable selector additionally requires a REP constraint used for specifying the number of times that the associated activity should be repeated.

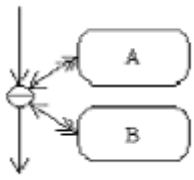
Optional responsibility ([A, B])		The optional selector which identifies an optional selection state is rendered as a circle overlaying a minus signal. .
Undo responsibility (A (↺))- d	**<<cancels>>**	A <<cancels>> is an interaction object flow that relates an instance of ActionInvoker to any composite activity or SelectionState. It specifies that the activity or has not finished normally. The Cancel object is responsible for allowing the user to cancel an activity.

Table 7.4 Mapping tasks notation to UML*i* Activity Graphs meta-classes

Figures 7.3 and 7.4 illustrate a simple example of mapping the CUCM of borrow a book in Figure 5.3 to a UML*i* activity diagram.

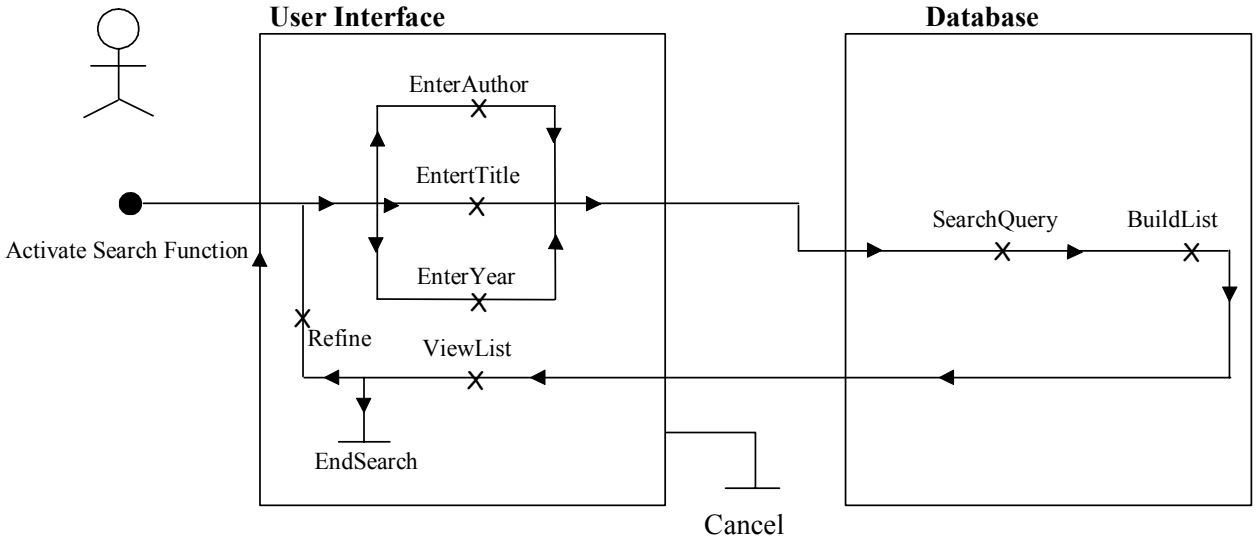


Figure 7.3 CUCM for search a book.

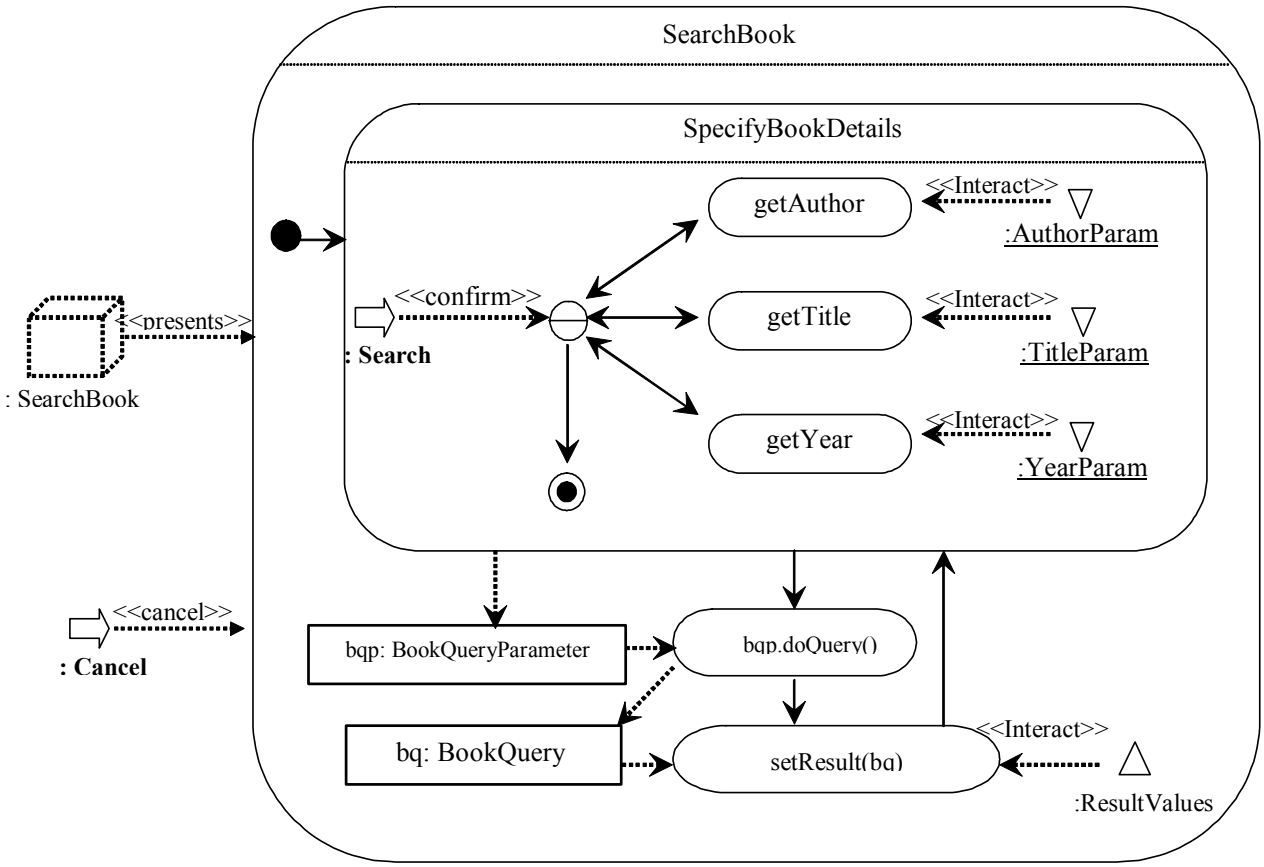


Figure 7.4 Activity Diagram for search a book.

7.3 Avenue 2: UCM-UI to LOTOS

The Language of Temporal Ordering Specifications (LOTOS) is an algebraic specification language and a Formal Description Techniques (FDT) standardized by ISO for the formal specification of open distributed systems (OSI-1089). LOTOS is executable and its models allow the use of different validation and verification techniques. Several tools can be utilized for automating these techniques (e.g. LOLA, ELUDO).

UCM-UI help user interface designers capture the user interface requirements to visualize the interface design in early stages before implementation starts. UCM-UI mostly focus on interface functionality and presentation, and they are easily learnable and understandable. However, UCM-UI has no formal semantics. They are not created for presenting detailed information. Therefore, they are not suitable for detailed analysis and design. On the other hand, LOTOS provides a solid platform for detailed analysis of functionality and protocols. It permits to perform validation and verification. Although LOTOS has many benefits, its complexity keeps designers from using it.

Over the past years, much work has been done on the derivation of LOTOS specifications from UCMs (Amyot et al., 1999, 2000; Guan, 2002). LOTOS is an algebraic language that can formalize the ordering of events found in UCMs, even in the absence of a component structure. This enables formal verification and verification of requirements, specifications, and designs, something that lacks from many case tools.

In this section, we will propose a method for the generation of LOTOS specifications from UCM-UI. The key idea of the translation is to establish a relationship between UCM-UI and LOTOS and then automatically translate UCM-UI to LOTOS specification. There has been already some research on automatic generation of LOTOS specification from UCMs (Guan, 2002). Based on their contributions we will build our method to translate UCM-UI to LOTOS.

7.3.1 Overview of LOTOS

LOTOS is a language for formal specification and formal modeling systems. By 'formal specification' we mean that it specifies the behavior of a system with a sound semantic basis. By 'formal modeling' we mean that a LOTOS specification is executable, thus it constitutes an 'abstract model' of the system. A number of LOTOS tutorials exist in the literature (Logrippo et al., 1992; Bolognesi and Brinksma, 1987). Therefore, we limit ourselves to a very brief overview of the language and of its use in the context of our research. A LOTOS specification as shown in Figure 7.5 consists of two main parts: the Abstract Data Types (ADT) part and the Control part.

```
specification systemName[gate1, ..., gateN]: <exit-behavior>
(* Abstract Data Type part: data types and value expressions *)
behavior
(* Control Part: system behavior *)
endspec
```

Figure 7.5 LOTOS specification.

The Abstract Data Type Part defines the data types and value expressions needed to specify the behavior of a system. It is based on the formal theory of algebraic abstract data types ACT-ONE. The most commonly used predefined libraries are *Boolean* and *NaturalNumber*. The library *Boolean* defines the constants *true* and *false* and defines the *not* operation that complements a Boolean value. The *NaturalNumber* library defines positive numbers (including zero).

The Control Part is the part of the specification that describes the internal behavior of the system. It is defined by a *behavior expression* followed by possible *process* definitions. A behavior expression is built by *combining* LOTOS *actions* by means of operators and possibly instantiations of processes. By composition we mean sequence, choice or *parallelism*.

7.4 UCM-UI to LOTOS Transformation

The UCM-UI to LOTOS transformation is performed following multiple steps:

– Analysis of the UCM-UI by looking at the decomposition of the maps into stubs and at the components involved in each map.

– Representation in a graph of the decomposition of the UCM maps and the stubs into stubs/plug-ins.

– Establishment of mapping rules between UCM and LOTOS elements.

7.4.1 Dealing with the Difference of Abstraction Levels

UCM-UI is abstract and is not meant to be very detailed. For this purpose, they are suitable for requirement specification. On the other hand, LOTOS specifications can be very detailed; they represent the interactions among system components in terms of message passing, rendezvous and synchronization. One of the reasons for this difference is that LOTOS specifications are executable, UCM-UI are not. When transforming UCM-UI into LOTOS, messages and other design details are added into the specification. This step must be done in full interaction with the UCMs designers in order to represent faithfully the system's expected behavior.

7.4.2 UCM and UCM-UI to LOTOS Mapping

The transformation from the semi-formal notation UCM-UI to the formal language LOTOS is not straightforward. Some *mapping* rules that transform UCM-UI into LOTOS elements were set up for this purpose. The mapping presented below is partially based on previous work by Amyot (1994) and Charfi (2001) to transform UCM to LOTOS.

– **start point:** It is most generally a LOTOS action having the start point's label. It could also be a sequence of actions, a guarded behavior, or nothing if the start point's label is empty.

– **end point:** It is most generally a LOTOS exit action carrying a value having the end point's label. It could also be an action or sequence of actions, or nothing if the end point's label is empty.

– **responsibility:** A LOTOS action having the responsibility's label or sequence of actions.

- Some of the UCM-UI responsibilities such as those for indicating optional and iterative responsibilities need to be mapped onto LOTOS expressions. An example is in the translation of the Responsibility iterative operator (*), its translation into LOTOS just requires a recursive call of the LOTOS process associated with the responsibility in order to simulate the behavior of restarting an activity just after the completion of an its previous execution.

– **or-fork:** LOTOS choice operator [] preceding each branch that represents a path on the right side of the or-fork.

– **and-fork:** Parallel composition operator preceding each branch that represents a path on the right side of the and-fork.

– **or-join**, **and-join:** Enable operator (>>).

– **components C1**, **C2:** Processes C1 and C2. If there is a direct UCM path from C1 to C2 in the map then processes C1 and C2 communicate through the gate C1 to C2.

– **static stub:**, plug-in Process having the stub's label. Responsibilities in the stub must be in the process gate list.

– **dynamic stub:** Process having the stub's label and composed of processes representing each plug-in of this dynamic stub.

– **alternative:** between plug-ins []

– **timed waiting place** [] between sequences of actions representing each alternative UCM path from the timed waiting place.

– **access to database** The database is represented by a LOTOS process. Each LOTOS action in this process corresponds to an action performed on the database of the UCM.

– **Dialog:** The dialog is represented by a LOTOS process. Each LOTOS action in the process corresponds to an action performed in the dialog of the UCM-UI. All responsibilities in the dialog must be in the process gate list.

A simple example on the transformation from a CUCM to LOTOS specification is illustrated using the ReturnBook use case in the CUCM of library system Figure 5.3. First, all stubs of the CUCM are *flattened* to represent in one map all the design details that are hidden by the stubs of the ReturnBook use case. The

process of *flattening stubs* is the process of replacing the stubs of a map by their internal behavior. Figure 7.6 covers all the possible behaviors of stubs. Figure 7.7 and 7.8 show the derived LOTOS specification of Figure 7.6 CUCM using the mapping presented above.

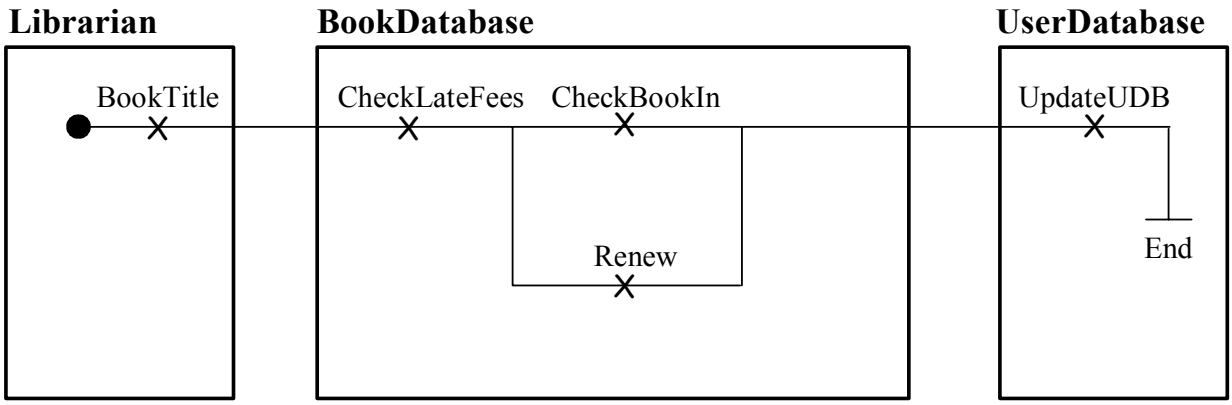


Figure 7.6 UCM for Return a book to a library.

```
specification RETURNBOOK[BookTitle, CheckLateFees, CheckBookIn, Renew,
UpdateUDB,End]: noexit:
(* Abstract Data Types here... *)
behaviour
hide Chan1, Chan2 in
        ( Librarian [BookTitle,Chan1] ||| UserDataBase[UpdateUDB,End,Chan2]
|       [ Chan1, Chan2 ]|
        BookDatabase[CheckLateFees, CheckBookIN,Renew,Chan1,Chan2])
where
(* Component processes here... *)
endspec (* ReturnBook *)
```

Figure 7.7 Structure of Components of the RenewBook use case.

```
process BookDatabase[CheckLateFees, CheckBookIn,Renew,Chan1,Chan2] : noexit:=
Chan1 !LibrarianToBookDatabase ! m1; LateFees;
(
        BookIn; Chan2 ! BookDatabaseToUserDB ! m2; (*_PROBE_*)
        BookDatabase [CheckLateFees, CheckBookIN,Renew,Chan1,Chan2]
        []
        Renew; Chan2 ! BookDatabaseToUserDB ! m3; (*_PROBE_*)
        BookDatabase [CheckLateFees, CheckBookIN,Renew,Chan1,Chan2]
)
endproc (* BookDatabase *)
```

Figure 7.8 : Component Behaviour of BookDatabase process.

7.5 Avenue 3: UCM-UI to Linear Textual Form

UCM-UI models describe user interface requirements and high-level designs with causal paths overlaid on a structure of components. The generation of a linear textual form from UCM-UI models can be used as a front end for other software engineering tools and it enables the validation of requirements and facilitates the transition from requirements to design. In this section, we address the challenges faced during the automated generation of such linear textual forms. Scenario definitions and traversal algorithms are first used to extract individual scenarios from UCM-UI and to store them as XML files. Transformations to other scenario languages (for instance, Message Sequence Charts) are then achieved using XSLT, however this is beyond the scope of this thesis. A number of researches were carried out on the transformation of UCMs to *eXtensible Markup Language* (XML) (Miga, 1998; Miga et al., 2001, Amyot and Eberlein. 2003). XML is a markup language for documents containing structured information. XML was created to structure, store and to send information. It is a common tool for all data manipulation and data transmission.

UCMs are represented in XML by the tool UCMNav according to the XML Data Type Definition for UCMs. Figures 7.9, 7.10, and 7.11 show parts of the data structure for UCM in XML. The elements in XML are shown in boxes and their attributes are listed below them. A UCM model is called an *UCM-Design* in XML. It contains root-maps and plugin-maps, which are represented as *Models* in XML. In the *UCM-Design*, *Components*, *Responsibilities* and *Plugin-bindings* are also specified. Each *Model* is made up of a collection of nodes (*hyperedges*) connected together (*hyperedge-connection*). For UCM-UI models the CUCM model can be easily represented by the XML using the Data Type Definition for UCMs. On the other hand PUCM are not. Therefore, we've chosen XUL a linear transformation language to transform PUCM to XUL.

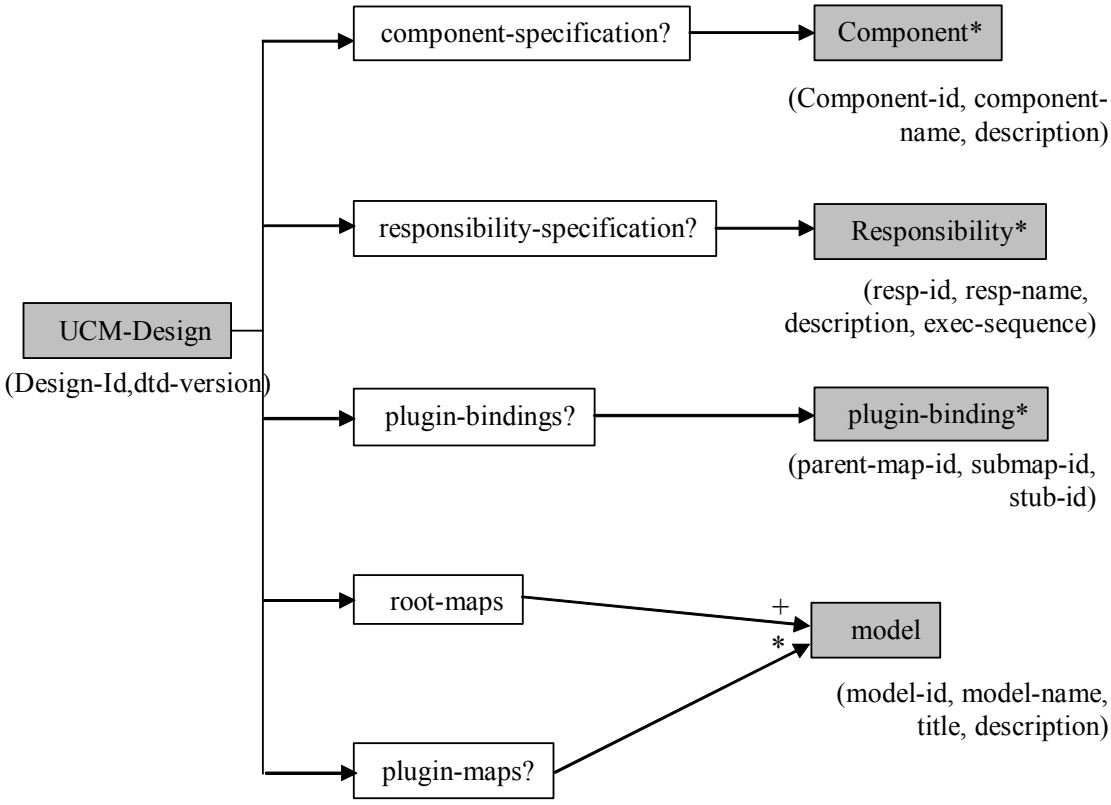


Figure 7.9 XML representation for UCM (Guan, 2002).

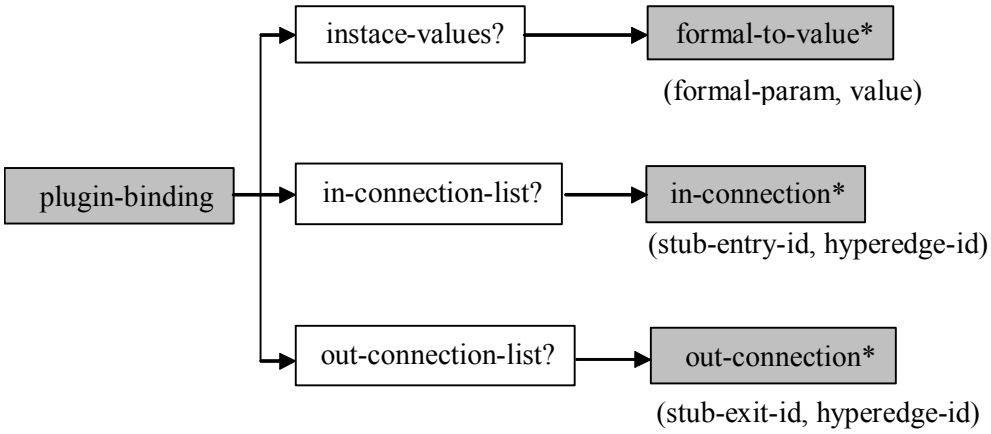


Figure 7.10 XML representation for plugin-binding (Guan, 2002).

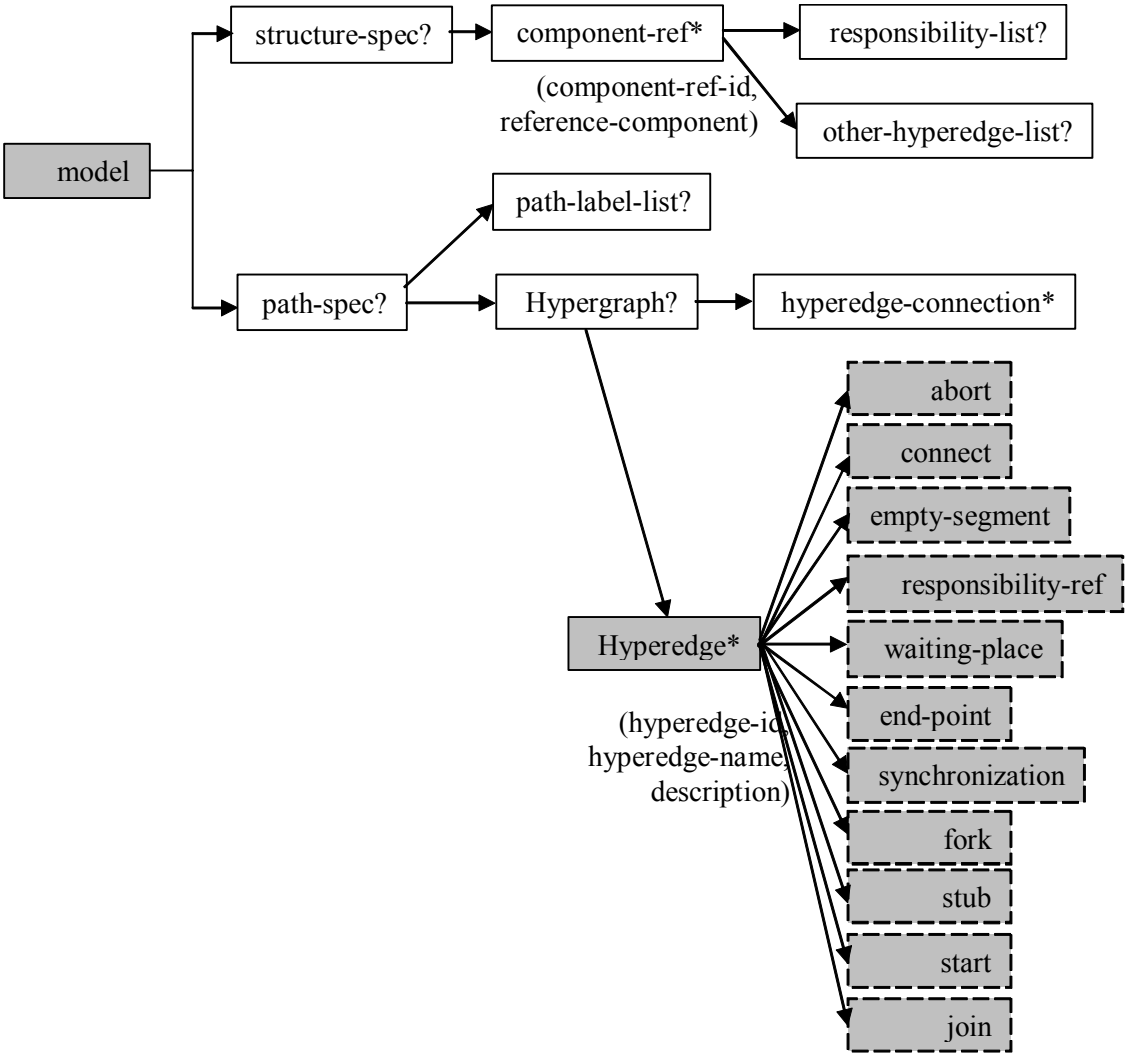


Figure 7.11 XML representation for model (Guan, 2002).

XUL stands for XML User-interface Language. XUL provides the ability to create most elements found in modern graphical interfaces. It is generic enough that it could be applied to the special needs of certain devices and powerful enough that engineers can create sophisticated interfaces with it. XUL is an XML language so all features available to XML are also available to XUL. The transformation of PUCM model to XUL can be implemented quickly and easily. Readers are referred to www.xulplanet.com for more information and examples on XUL.

7.5.1 UCM Navigator Tool

The challenge of extracting scenarios from UCMs was first tackled by Miga et al. in (2001), and their solution was prototyped in the UCM Navigator (UCMNav) tool. UCMNav is a multi-platform tool written in C++. This tool can highlight the UCM paths traversed according to scenario definitions, and generate individual

scenarios in the form of Message Sequence Charts using the Z.120 textual syntax. The UCM Navigator tool offers the following features:

– Creation and manipulation of Use Case Maps that are always syntactically correct.

– Path transformations and path connections based on internal hypergraph-based semantics.

– Nested levels of stubs and plug-ins (sub-maps), with explicit binding of plug-ins to stubs.

– Exporting and importing of root and plug-in maps to/from files.

– Separation of responsibility definitions from inclusion along paths, allowing reuse.

– Binding of path elements to Buhr's architectural notation:

  - Component attributes (type, formal, anchored, fixed, replicated, color, etc.) are automatically reflected visually.
  - Components can be nested inside other components (warnings can be displayed for invalid bindings).
  - Pools and information on dynamic responsibilities are supported.
  - Moving a component moves all its sub-components, elements, and path segments.
  - Resizing a component can automatically reshape the paths bound to it.

– Support of extensions for agent systems and performance modeling.

– Comment fields for almost any part of a map (path elements, components, goals, etc.).

– Pre/post conditions which can be attached to many path elements.

– XForms-based GUI with scalable maps, large drawing area, zoom, and scroll bars.

– Generation of XML files valid with respect to the UCM Document Type Definition.

– Flexible report generation in PostScript:

  - Selection of sections to include: map description, responsibilities, path elements, components, stub descriptions, and goals.
  - Ready for PDF generation: index, map stubs, and plug-in names are hyperlinks. (Very useful for on-line presentations.)

- Option for maps on separate pages.

– Exports maps in Encapsulated PostScript (EPS) and Maker Interchange Format (MIF)

– Easy-to-install binaries available for three platforms: Solaris, Linux (Intel and SPARC), and HP/UX

The UCM Navigator is designed to handle any valid UCM as well as software components. It is capable of binding these UCM elements to components. UCM Navigator is also capable of creating multi-level maps in which sub-maps of a lower level are expressed as stubs in a higher-level map. The editor currently supports nested levels up to 10 deep. UCM Navigator generates industry-standard XML files which can be used as input to other tools, including Agent generation tools and performance simulators. More research is needed to extend the UCM Navigator to handle UCM-UI models.

7.6 Conclusions

UCM-UI was never intended to be a stand-alone methodology but rather a complement to traditional design specification techniques. UCM-UI fit in many different software engineering methodologies and design processes. For example, Section 7.1 shows the smoothly integration of the UCM-UI concepts into UML and UML*i*.

In this chapter we also show that from the UCM-UI models a LOTOS specification is derived. The translation from UCM-UI to LOTOS corresponds to the transformation and formalization of an abstract, semiformal model into a less abstract, formal and executable one. First, the UCM-UI are analyzed. Design errors can already be corrected at this stage. Second, a LOTOS specification is derived following some UCM to LOTOS mapping rules presented in section 4.6.3. In this experience, the LOTOS specification that we obtained from the UCMs was hand-prepared. However, some work has been done by Amyot to formalize the transformation of a UCM map into a LOTOS specification (Amyot, 1994). In addition, the automation of the transformation of UCMs into a skeleton of LOTOS specification is currently being investigated within the LOTOS research group of the University of Ottawa. However, this automation is not part of our work.

We also show that a textual linear form for UCM-UI expressed in XML and XUL can be defined. This form is suitable for input to different tools and for generating documentation. Having this XML and XUL Document Type Definition also enables an easier integration of UCM-UI with upcoming standards for UML such as the *XML Metadata Interchange* (XMI) and the *UML eXchange Format* (UXF).

Chapter 8

Conclusions and Future Research

This chapter concludes this thesis. It summarizes the previous chapters that discussed various aspects of *Scenario and Use Case-Based for User Interface Requirement Engineering*. On the one hand this thesis shows the benefits of scenario and use case based requirements but on the other hand it shows that many improvements can still be made. We first summarize the work and state the main contributions of this thesis. Since the field of user interface requirements is still developing rapidly, we discuss some ideas for future research on task-based design.

8.1 Summary of the Thesis

Better techniques for improving human understanding and communication will result in better user interfaces built more quickly with fewer design errors in it and with fewer design-violating errors creeping into it through implementation changes made during maintenance, reengineering, and evolution. This dissertation offers the prospect of having a significant impact on representing and understanding user interface requirements by proposing a new framework. The new framework SUCRE; a scenario and use case based requirements engineering framework, define a set of extensions to UCM in order to improve the support it provides for modeling interactive systems. The necessity of extending UCM comes from the shortcomings of UCM identified in Chapter 3. We build a complete UCMs notation for high-level requirements of user interfaces and put together three dimensions of user interface requirements including the task, the dialog, and the structure of the user interface.

SUCRE framework ensures that: (1) a consistent and complete requirement specification can be captured using scenarios and use cases, (2) the specification is a valid reflection of user requirements, (3) the derivation of early design artifacts such as low fidelity prototypes. SUCRE also, defines operators for a formal analysis of the consistency, completeness and precision of the UCM-UI models. Chapter 5 illustrate that the use of the proposed operators had facilitated in generating models and implementations faster and at a lower cost while improving their correctness and traceability with respect to the requirements.

In the proposed framework, we also aim at rapid prototyping for the purpose of end user validation at an early stage of development. The generated prototype serves as a vehicle for evaluating and enhancing the UI and the underlying specification. When only paper mockups are available it is already possible to ask users to perform their tasks. This usually leads to high level comments about the interface. When a running prototype is been developed, detailed usability testing can be done. In early phase, we use the UCM-UI usability metrics suite from Chapter 6 to predict usability, measure the correct aspects and to look for problems or possible improvements.

This thesis also argues that detailed requirements need a mix of formal and informal representations. Sometimes there is a need for precision while creativity

often asks for informal representation. UMLi, LOTOS, and XML are precise techniques that allow bridging the gap between use case modeling for user interface and detailed design. There notations are suitable for the proposed UCM-UI model. Techniques to link UCM-UI model to the above precise representations has been described in Chapter 7.

Concisely, SUCRE framework provides rich and precise mode of expression that not only meets the needs of user interface designers and software engineers, but is also easily understood and validated by end users. It acts as an effective bridge between usability engineering and user interface design on the one hand and software designer on the other. For usability engineers, it provides a concise medium for modeling user requirements. In the hands of user interface designers, it is a powerful task model for understanding user needs and guiding user interface design. For software engineers, it guides the design of communicating objects to satisfy functional requirements.

8.2 Contributions

This thesis discusses *Scenario and Uses Case-Based for User interface Requirement Engineering*. Starting with a general outline of the requirement process, each of the main activities is discussed in detail. Not only the theoretical aspects of the framework are discussed but also practical aspects such as representation techniques and tool support. In Chapter 2, the main pitfalls of current scenario-based approaches for user interfaces are presented. Next we discuss our contributions to solve these problems:

– *Improving the available techniques of scenario and use cased–based models for user interfaces.* Developing better techniques has been done by first addressing the most significant and missing aspects in the current use of scenarios and use case-based models for user interface descriptions. A detailed study of the existing models for user interface was carried out in Chapter 2. The different scenario and use case-based models are evaluated and their pros and cons are highlighted. Then, we built on top of this knowledge the Use Case Maps for User interface (UCM-UI) model, a comprehensive proposal for improving the supporting UCM to model interactive systems. The model is based on common concepts in scenario analysis techniques and has been developed further over the duration of the research project. Using the UCM-UI model we have looked at representations for designers to answer the question *how* and *what* can be modeled.

– *Reducing the effort of developing detailed use case models.* The effort required is reduced in several ways. First of all, by offering improved techniques and models designers have a much better idea of *what* and *how* to model a user interface. It gives them conceptual tools to start doing scenario analysis. Secondly, the manual activities such as creating and editing the UCM-UI models are supported by UCM Navigator tool. We demonstrate how UCM-UI can be implemented in the UCM Navigator tool.

– *Formal validating the requirements*. The process of constructing UCM-UI model may sensibly be augmented by a set of design heuristics for setting up a formal specification Chapter 5 demonstrates that these heuristics may be used to build operators that validate the UCM-UI model. The use of the operators defined in results in consistent, complete, and precise user interface requirement that is built more quickly with fewer design errors in it and with fewer design-violating errors creeping

into it through implementation changes made during maintenance, reengineering, and evolution.

– *Effective employment of use cases in the design of the actual product.* One of the difficult steps in the user interface development process is the transition from the requirement phase to the design phase. This transition must be characterized by a combination of engineering and creativity in order to incorporate requirement results in a concrete design**.** Chapter 7 developed strategies for (a) linking UCM-UI to UML, (b) generating LOTOS specifications from UCM-UI, and (c) storing UCM-UI scenarios in XML and XUL files.

– *Investigate metrics to predict system usability in the requirements phase.* Chapter 6 of this thesis is entirely devoted to this issue. We have developed a metrics suit the UCM-UI metrics suite to allow interface designers using the SUCRE framework to predict and improve the interface quality, to control and improve the requirement processes, and to decide on the acceptance of the interface. This metrics suite gives guidance when making design decisions and when evaluations are done.

8.3 Future Research

Developing a requirement framework almost never ends. There are still many areas that need to be improved both on the theoretical and the practical side. Theories are important to link together many aspects in the process of requirements. With the help of sound theories we can develop practical techniques that make improvements of the process and product possible. Many items left for future work are distributed among the previous chapters. The following list recalls the most important ones, which target the automation and generalization of this work:

– *Bringing SUCRE to the work practice*. In the area of user interface requirement engineering this thesis has made advances both on theory and on practical techniques. Nonetheless, those techniques probably need to be refined through many applications before they are established techniques that every engineer uses. Extensive application experiences can show missing aspects, tune techniques and improve insights.

– *Integrated tools supporting several activities of the SUCRE framework*. This tool, which could be part of the existing UCM Navigator tool. UCM Navigator needs to be extended to support the UCM-UI models. Moreover, a data dictionary for early usability prediction of the UCM-UI scenarios must be defined. The current UCM Navigator is a prototype that is not finished and more development could lead to a tool of commercial quality. Tools are nowadays essential elements in most design processes. Any method or framework for software development could greatly benefit from tool support both for the process itself but also for acceptance in the industry.

– *The development of automated tools supporting consistency, completeness and precision* checking for UCM-UI requirements at each step of evolution, thus providing automated proofs of UCM-UI correctness. Such automated support will allow us to test the validity of our argument by applying it in a case study over an entire release of a product family that can be developed with the UCM Navigator.

.

References

Alsumait, A.; Seffah, A; & Radhakrishnan, T. (2002). Use Case Maps: A Roadmap for Usability and Software Integrated Specification". In *Proceedings of IFIP World Computer Conference*, Montreal, Canada, 119-131

Alsumait, A.; Seffah, A; and & Radhakrishnan, T. (2003). Use Case Maps: A Visual Notation for Scenario-Based Requirements. In *Proceedings of the 10th International Conference on Human - Computer Interaction*, Crete, Greece, June 22-27, 3-7.

Amoroso, D; & Cheney, P. (1992). A report on the state of end-user computing in large North American insurance firms. *Journal of Information Management 8*(2), 39-48.

Amyot, D.; & Eberlein, A. (2003). An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunications Systems Journal* 24(1), 61-94.

Amyot, D. (1994). *Formalization of Timethreads Using LOTOS.* Master thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

Amyot, D. (1999). *Use Case Maps and UML for Complex Software-Driven Systems*. Technical Report, August 1999. www.usecasemaps.org.

Amyot, D. (2000). Use Case Maps as a Feature Description Language. In: *Language Constructs for Designing Features*, S. Gilmore and M. Ryan, eds. Springer-Verlag, 27-44.

Amyot, D.; & Mussbacher, G. (2001). Bridging the Requirements/Design Gap in Dynamic Systems with Use Case Maps (UCMs), Tutorial in: *23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada.

Amyot, D.; Andrade, R.; Logrippo, L.; Sincennes, J.; & Yi, Z. (1999). Formal Methods for Mobility Standards. IEEE 1999 Emerging Technology Symposium on Wireless Communications & Systems, Dallas (TX), USA, April 1999. Editor: Traci

Annett, J.; & Duncan, K. (1967). Task analysis and training in design. *Occupational Psychology 41*, 211–221.

Antón, A. I.; & Potts, C. (1998a). A Representational Framework for Scenarios of Systems Use. *Requirements Engineering Journal 3*, 219-241.

Antón, A. I.; & Potts, C. (1998b). The Use of Goals to Surface Requirements for Evolving Systems, In: *International Conference on Software Engineering (ICSE '98),* Kyoto, Japan, April, 157-166.

Ardis, M. A.; Chaves, J. A.; Jagadeesan, L. J.; Mataga, A, P.; Puchol, C.; Staskauskas, M. G.; & Olnhausen, J. V. (1996). A Framework for Evaluating Specification Methods for Reactive Systems - Experience Report. *IEEE Transactions on Software Engineering*, 22 (6), 378-389.

Avison, D; & Fitzgerald, G. (1995). Information systems development: methodologies, techniques and tools. McGraw-Hill, Maidenhead, UK.

Bahrami, A. (1999). *Object-Oriented Systems Development using the Unified Modelling Language*. McGraw-Hill.

Bamum, C. (2002). *Usability Testing and Research*. Allyn and Bacon, NY.

Bell, T. E.; & Thayer, T. A. (1976). Software requirements: Are they really a problem?. In: *Proceeding ICSE-2: 2nd Intrnational Conference on Software Enginering*, San Francisco, 61-68.

Ben Achour, C.; Souveyet, C.; & Tawbi, M. (1999). Bridging the Gap between Users and Requirements Engineering: the Scenario-Based Approach. *International Journal of Computer Systems Science and Engineering,* 379-385.

Benner, K. M.; Feather, M. S.; Johnsin, W. L.; & Zorman, L.A. (1993). Utilizing Scenarios in the Software Development Process. In *Proceedings of of IFIP WG 8.1 Working Conference on Information Systems Development Process*, 117-134.

Benyon, D.; & Macaulay, C. (2002). Scenarios and the HCI-SE design problem. *Interacting with Computers 14(4)*, 397-405.

Berry, D. (1994). Involving Users in Expert System Development. *Expert Systems, 11(1)*, 23-28. 1994.

Bevan, N. (1999). Quality in Use: Meeting User Needs for Quality, *Journal of System and Software, 49(1)*, 89-96.

Bevan, N.; & Curson, I. (1997). Who Needs Usability Metrics?. *INTERACT.* 123-125.

Bickerton and Siddiqi, 1993

Bly, S. (1997). Field work: Is it product work? *Interaction, 4(1)*, 25-30.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall.

Bolognesi, T.; & Brinksma, E. (1987). Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN systems 14*, 25-59.

Buhr, R. (1996). Use Case Maps for Attributing Behaviour to System Architecture. In *Proceedings of Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS*), Honolulu, Hawaii, April 15-16.

Buhr, R. (1998). Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering 24 (12)*, 1131-1155.

Buhr, R.; & Casselman, R. (1996). *Use Case Maps for Object-Oriented Systems*, Upper Saddle River, NJ: Prentice Hall.

Card, S.; Moran, T.; & Newell, A. (1983). *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates.

Carroll J. M. (1999). Five reasons for scenario-based design. In: *Proceedings of the 32nd Hawaii International Conference on Ssystems Sciences*, Wailea, HI. Page

Carroll, J., (2002). Scenarios and design cognition. In *Proceedings. IEEE Joint International Conference on Requirements Engineering (RE'02)*. Essen, Germany 9-13 Sept. 2002. 3 – 5.

Chandrasekaran, P. (1997). How use case modeling policies have affected the success of various projects. In *Proceedings Addendum to the 1997 ACM SIGPLAN Conference on Object-Oriented Programming, Systems*, Langauages, and Applications. 6-9.

Charfi, L. (2001). Formal Modeling and Test Generation Automation With UCMs and LOTOS. Master thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

Checkland, P. 1981. *Systems Thinking, Systems Practice*. Wiley, Chichester, UK

Chin, G.; Rosson, M.; & Carroll, J. M. (1997). Participatory Analysis: Shared Development of Requirements from Scenarios. In *Proceedings on Human Factors in Computing Systems*, Atlanta, Georgia, United States, 162-169.

Clavadetscher, C. (1998): User involvement: key to success. *IEEE Software, Requirements Engineering 2*. 30-33

Cockburn (1999)

Cockburn, A. (1997). Structuring Use cases with goals. *Journal of Object-Oriented Programming, 10(5)*, 56-62.

Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley.

Collins, D. (1995). *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, Redwood City, CA.

Constantine, L. (1995). Essential Modeling: Use Cases for User Interfaces. *ACM Interactions 2*, 34-46.

Constantine, L. (1996). Usage-Centered Software Engineering: New Models, Methods, and Metrics. In *Software Engineering: Education and Practice,* M. Purvis, ed. IEEE Computer Society Press. Los Alamitos, California.

Constantine, L.; & Lockwood, A. D. (1999). *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley Publishing ACM Press.

Darke, P; & Shanks, G; (1997). User Viewpoint Modelling: Understanding and Representing User Viewpoints during Requirements Definition. *Information System Journa 7(3)l,* page

Davis, A. M. (1993). Software Requirements-Objects, Functions and States. Prentice Hall.

Duke, D. J.; Fields, R. E; & Harrison, M. D. (1999). A Case Study in the Specification and Analysis of Design Alternatives for a User Interface. *Formal Aspects of Computing. 11(2)*, 107-131.

Elkoutbi, M.; Khirss, I.; & Keller, R. K. (1999). Generating User Interface Prototypes from Scenarios. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, Limerick, Ireland. 150.

Enderton, H. B. (1977). *Elements of Set Theory*. Academic Press, London, UK.

Gervasi,V.; & Nuseibeh, B. (2002). Lightweight validation of natural language requirements. *Software: Practice and Experience 32(2)*, 113-133.

Glinz, M. (1995). An integrated formal Model of Scenarios based on Statecharts, *Lecture Notes in Computer Science*, 254-271.

Goguen and Linde, (1993).

Grice, R.A. (2003). Comparison of cost and effectiveness of several different usability evaluation methods: A classroom case study. In *Proceedings of IEEE International Professional Communication Conference*, Sept. 21-24, 2003 140 – 144.

Guan, R. (2002). *From Requirements to Scenarios through Specifications: A Translation Procedure from Use Case Maps to LOTOS*. Master thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada.

Harel, D. (2001). From Play-In Scenarios to Code: An Achievable Dream. *IEEE Computer 34(1),* 53-60.

Haumer, P. (2000). *Requirements Engineering with Interrelate Conceptual Models and Real world Sense*. Ph.D. thesis. Dept. of Information Systems, Technical University of Aachen, Aachen, Germany.

Haumer, P.; Heymans, P.; Jarke, M.; & Pohl, K. (1999). Bridging the gap between past and future in re : a scenario-based approach. In *Proceedings of the Fourth IEEE*

International Symposium on Requirements Engineering (RE'99), Limerick, Ireland. Page.

Hertzum, M. (2003). Making use of scenarios: a field study of conceptual design. *International Journal of Human-Computer Studies 58 (2)*, 215-239.

Hix, D.; Hartson, H. (1998). Developing User Interfaces: Ensuring Usability Through Product and Process. John Wiley and Sons Inc.

Holzblatt, K.;. & Beyer, K. R (1995). Requirements gathering: the human factor. *Communication of the ACM 38 (5),* 31.

Hsiaf, P.; Samuel, J.; Gao, J., Kung, D.; Toyoshima, Y.; Chen, C. (1994). Formal Approach to Scenario Analysis. *IEEE Software 11(2)*, 33-41.

IEEE-610.12. (1991). IEEE Standard Glossary of Software Engineering Terminology.

IEEE-830. (1984). Guide to Software Requirements Specification, ANSI/IEEE Std. 830.

Iivari, J; Hirschheim, R. (1996). Analysing information systems development: a comparison and analysis of eight IS development approaches. *Information Systems 21(7)*, 551-575.

ISO 9214-11. (1991). *Ergonomic Requirements for office Work with VDT's – Guidance on Usability*.

ITU-T 2003 Recommendation Z.150, *User Requirements Notation (URN) - Language Requirements and Framework*. Geneva, Switzerland. http://www.UseCaseMaps.org/urn/.

Ivory, M.; & Hearst, M. (2001). The state of the art in automating usability evaluation of user interfaces, *ACM Computing Surveys (CSUR) 33 (4)*, 470–516.

Jacobson, I. (1992). *Object-Oriented Software Engineering*. Addison-Wesley, Readin, MA.

Jacobson, I. (1995). The use-case construct in object-oriented software engineering. In *Scenario-Based Design, Envisioning Work and Technology in System Development*, Carroll, J. M, Ed. John Wiley and Sons, New York, 309-336.

Jarke, M. (1999). Scenarios for modeling. *Communications of the ACM 42(1*), 47-48.

Jarke, M.; Bui, X.; & Carroll, J. M. (1998). Scenario Management: An Interdisciplinary Approach. *Requirements Engineering 3*, 155-173.

John, B. E.; & Kieras, D. E. (1996). The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction 3*, 320–351.

Kirwan, B.; & Ainsworth, L. K. (1992). *A guide to task analysis*. Taylor and Francis, Washington, DC.

Koh, S; & Heng, M. (1996). Users and Designers as Partners: Design Method and Tools for User Participation and Designer Accountability within the Design Process. *Information System Journal 6(4)*, page.

Kruchten, P.; Ahlqvist, S.; & Bylund, S. (2001). User Interface Design in the Rational Unified Process, In *Object Modeling and User Interface Design*, Van Harmelen, Ed. Addison-Wesley, Reading, Massachusetts.

Kujala, S. (2002). User Involvement: A Review of the Benefits and Challenges. In *Preprints, Software Business and Engineering Institute*, T. Soininen, ed. Helsinki University of Technology, Report no.: HUT-SoberIT-B1. Espoo, Finland, 1-32.

Kujala, S. (2003). User involvement: a review of the benefits and challenges. *Behavior and Information Technolog, 22(1)*, 1-16.

Kurke, M. I. (1961). Operational sequence diagrams in system design. *Human Factors 3*, 66-73.

Kyng M. Year. Creating contexts for design. In *Scenario-based design: envisioning work and technology in system development*, J. M. Carroll, ed. Wiley, New York, 85–107.

Lee, J.; & Xue, N. (1999). Analyzing user requirements by use cases: A goal-driven approach. *IEEE Software 16(4)*, 92-101.

Lilly, S. (1999). Use case pitfalls: Top 10 Problems from real projects using use cases. In *Proceedings of Technology of Object-Oriented Languages and Systems, TOOLS 30*, 174-183.

Logrippo, L.; Faci, M.; & Haj-Hussein, M. (1992). An Introduction to LOTOS: Learning by Examples. *Computer Networks & ISDN Systems23(5*), 325-342.

Lutz, R. (1993). Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. In: *Proceedings of First International Symposium on Requirements Engineering*, San Diego, 126-133.

Maack, J. N. (2002). Scenario Analysis: A Tool for Task Managers, ….

Macaulay, L. A. (1996). Cooperation, requirements analysis and CSCW. In *CSCW Requirements and Evaluation*, P. Thomas, Ed. Springer-Verlag London, page

Madsen, K. H. (1999). The Diversity of Usability Practices. *Communications of the ACM 42(5)*, 60-62.

Miga, A. (1998). *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Canada.

Miga, A.; Amyot, D.; Bordeleau, F.; Cameron, C.; & Woodside, M. (2001). Deriving Message Sequence Charts from Use Case Maps Scenario Specification. In *Tenth SDL Forum (SDL'01)*, Copenhagen, 2001. LNCS 2078, 268-287

Morgan, C. C.; & Sufrin, B. (1993). Specification of the UNIX Filing System. In *Specification Case Studies*, 2nd ed., I. Hayes, Ed. Prentice-Hall, London, UK.

Mumford, E. (1985). Defining System Requirements to Meet Business Needs: A Case Study Example. *The Computer Journal 28(2)*, 97-104.

Mumford, E. (1985). Participation: From Aristotle to Today. In *Beyond Productivity: Information Systems Development for Organizational Effectiveness*, T. Bemelmans, Ed. Elsevier Science Publishers, page.

Nasr, E.; McDermid, J.; & Bernat, G. (2002). A Technique for Managing Complexity for Large Complex Embedded Systems. In Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002). 29 April- 1 May, Washington, D.C., USA, 225-232

Nielsen J. (1996). Usability Metrics: Tracking Interface Improvements. *IEEE Software 13(6)*, 12-13.

Nielsen, 1995

Nielsen, J. (1993). *Usability Engineering*. Academic Press.

Nielson, J. (1990). Paper versus computer implementations as mockup scenarios for heuristic evaluation. In *Proceedings of the Third International Conferences on Human-Computer Interaction*, 1-8.

Noble J.; & Constantine, L. (1997). Interactive Design Metric Visualization: Visual Metric Support for User Interface Design. In: *Proceedings of OzCHI '96*. Los Alamitos, California.: IEEE Computer Society Press.

Norman, D. A. (1998). *The Design of Everyday Things*. The MIT Press.

Nuseibeh, B.; & Easterbrook, S. (2000). Requirements engineering: A roadmap. .In *Proceedings of International Conference on Software Engineering (ICSE-2000)*, ACM Press, June 2000, Limerick, Ireland.

Overmyer, S. P. (1999). The Use of Scenarios in Developing, Validating, and Specifying Requirements for Interactive Systems: A case study from a NASA project. *Paper presented at REFSQ'99*, Heidelberg, Germany. http://www.ifi.uib.no/konf/refsq99/papers.html.

Palanque, P.; & Patern`o, F. (1997). *Formal Methods in Human Computer Interaction*, Springer Verlag.

Palmer, J. D. (1987). Uncertainty in software requirements. *Large Scale Systems 12*, 257-270.

Parker et al., (1997).

Paternò, F. (2001). Towards a UML for Interactive Systems. In *Engineering HCI '01, Lecture Notes Computer Science*, Springer Verlag, Toronto, 7-18.

Paternò, F., (1999). Model-Based Design and Evaluation of Interactive Applications, Springer-Verlag, London, UK.

Paternò, F.; Mori, G.; & Galimberti, R. (2001). CTTE: An Environment for Analysis and Development of Task Models of Cooperative Applications. In *Proceedings ACM CHI'01*, March 2001, Seattle, 21-22.

Payne, S.; & Green, T. (1989). Task-Action Grammar: The Model and its Developments, In *Task Analysis for Human-Computer Interaction*, D. Diaper, ed. Ellis Horwood, Cambridge MA.

Pfleeger, S. L. (1998). *Software Engineering: Theory and Practice*. Prentice-Hall, New Jersey, USA.

Phillips, C.; & Kemp, E. (2002). In support of user interface design in the rational unified process. In *ACM International Conference Proceeding, Third Australasian conference on User interfaces*, Melbourne, Victoria, Australia, 21-27.

Phillips, C.; & Kemp, E. (2002). In support of user interface design in the rational unified process. In Proceedings *of the 3rd Australasian conference on User interfaces - ACM International Conference*, Melbourne, Victoria, Australia, 21 – 27.

Pine, B: (1989). Design, Test and Validation of Application System/400 Through Early User Involvement. *IBM Systems Journa, 28(3)*,376-385.

Pinheiro da Silva, P. (2000) UML*i*: Integrating User Interface and Application Design. In *Electronic proceedings of the UML2000 Workshop on Towards a UML Profile for Interactive Systems Development (TUPIS2000)*, York, United Kingdom.

Pinheiro da Silva, P.; & Paton, N. W. (2003). User Interface Modeling in UML*i*. *IEEE Software 20(4)*, 62-69.

Pohl K. (1994).The three dimensions of requirements engineering: a framework and its applications. *Information Systems 19(3)*, 243–258

Pohl, K. (1996). Process Centered Requirements Engineering. J. Wiley and Sons Ltd.

Potts, C.; Takahashi, K.;. & Anton A. (1994). Inquiry-based Requirements Analysis. *IEEE Software 11(2)*,21-32.

Pomerol, J. (1998). Scenario Development and Practical Decision Making under Uncertainty: Application to Requirements Engineering. *Requirements Engineering 3*,174-181.

Regnell, B.; Kimbler, K.; & Wesslen, A. (1995). Improving the Use Case Driven Approach to Requirements Engineering. In *Second IEEE International Symposium On Requirements Engineering*, York, England, March 1995, I. C. S. Press, Eds., 40-47.

Robotham, T.; Hertzum, M. (2000). Multi-Board Consept- A Sacenario based approach for supporting product quality and life cycle oriented deign. In *Proceedings of TMCE 2000: Third International Symposium on Tools and Methods of Competitive Engineering*, Delft, The Netherlands, April 18-21, 2000, I. Horvth, A. J. Medland, and J. S. M. Vergeest, eds. 763-774.

Rolland, C.; Ben Achour, C.; Cauvet, C.; Ralyte, J.; Sutcliffe, A. G.; Maiden, N., Jarke, M.; Haumer, P., Pohl, K., Dubois, E.; & Heymas, P. (1998). A proposal for a scenario classification framework. *Requirements Engineering Journal, 3(1)*, 23-47.

Rouff, C. (1996). Formal Specification of user interfaces. *SIGCHI Bulletin. 28(3)*, 27-33.

Rubin, S.; & Goldberg, A. (1992). Object Behaviour Analysis. *Communications of the ACM 35(9)*, 48-62.

Rudd, J.; Stern, K.; Isensee, S. (1996). Low vs. high fidelity prototyping debate. *Interactions 3*, 76-85.

Rumbaugh, J. (1994). Getting started-Using use cases to capture requirements. *Journal of Object-Oriented Programming 7*, 8-23.

Rumbaugh, J.; & Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. (1991). *Object-Oriented Modeling and Design*, Prentice Hall.

Rumbaugh, J.; & Booch, G. (1996). *Unified Method", Notation Summary" Version 0.8*. Rational Software Corporation.

Ruthford, M. A. (2002). Mix and match usability methods: picking the pieces for our project In *Proceedings of IEEE International Professional Communication conference*, 17-20 Sept. 343 – 351.

Sack, K. (1985). User Participation in Software Development: What is it, Why, and How?. In *Education for System Designer/User Cooperation*, U. Briefs and E. Tagg, eds. Elsevier Science Publishers, 1985.

Scapin, D.; & Pierret-Golbreich, C. (1989). Towards a method for Task Description: MAD, *Work With Display Units 89,* 371–380.

Scheurer, T. (1994). *Foundations of Computing: System Development with Set Theory and Logic. International Computer Science Series*. Addison-Wesley, University Press, Cambridge.

Schmidt, D. A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc.

Seffah, A.; & Hayane, C., (1999). Integrating Human Factors in Use Case and OO Methods. In *Proceedings of Integrating Human Factors in Use Case and OO Methods-13th ECOOP*. June 14-19, Lisbon, Portugal, 240-254.

Seffah, A.; Alsumait, A.; & Radhakrishnan, T. (2002). Use Case Maps: A Roadmap for Usability and Software Integrated Specification. Usability Stream. In *Proceedings of the IFIP World Computer Conference*. Montreal, Canada, August 2002, 25–30.

Shneiderman, B. (1998). *Designing the User Interface*, Addison-Wesley Publishing Company, USA.

Silvestre, P. (1996). Le Développement des Systèmes d'Information : de Merise à RAD. Hermès, Paris, France.

Some, S.; Dssouli, R.; & Vaucher J. (1996). Toward an Automation of Requirements Engineering using Scenarios. *Journal of Computing and Information, 2(1)*, 1110-1132.

Sprivey, J. M. (1992). *The Z Notation: A Reference Manual*, 2nd edition, Prentice-Hall.

Srdjan, 1998

Standish Group, (1998). *Software Chaos*. http://www.standishgroup.com/chaos.html.

Sulack et al., (1989).

Sutcliffe, A. (2003). Scenario-based requirements engineering. In *Proceedings of the 11th IEEE International, Requirements Engineering Conference.* 8-12 Sept. 2003. 320 – 329.

Sutcliffe, A.; & Ryan, M. (1998). Experience with SCRAM, a Scenario Requirements Analysis Method. In *Proceedings: IEEE International Symposium on Requirements Engineering: RE '98*, Colorado Springs C., Los Alamitos, CA: IEEE Computer Society Press. 6-10 April 1998. 164-171.

Thomas, C.; & Bevan, N. (1996). Usability Context Analysis: A practical guide, Version 4. National Physical Laboratory, Teddington, UK.

Torkzadeh, G.; & Doll, W. (1994). The Test-Retest Reliability of User Involvement Instruments. *Information and Management 26*, 21-31.

van der Poll, J. A.; & Kotzé, P. (2002). What Design Heuristics May Enhance the Utility of a Formal Specification? In *Proceedings of SAICSIT 2002: Enablement Through Technology*. Port Elizabeth, South Africa. 179 – 194.

van Der Poll, J.; Kotzé, P.; Seffah, A.; Radhakrishnan T.; & Alsumait, A. (2003). Combining UCMs and Formal Methods for Representing and Checking the Validity of Scenarios as User Requirements. In *Proceedings of South African Institute of Computer Scientists and Information Technologists SAICSIT 2003*, 59-68. .

van Lamsweerde, A. (2000a). Requirements Engineering in the Year 00: A Research Perspective. In: *Proceedings of the 22nd International Conference on Software Engineering*, June 2000, 5-19.

van Lamsweerde, A. (2000b). Formal specification, In: *Proceedings of the conference on The future of Software engineering*. May 2000, 147-159.

van Welie, M. (2000). Task-Based User Interface Design. Ph.D. Thesis, Vrije University, Amsterdam .

van Welie, M.; van der Veer, G.; & Koster, A. (2000). Integrated representations for task modeling, In *Proceedings of the'Tenth European Conference on Cognitive Ergonomics*, Linkoping, Sweden, 129–138.

Vanlommel, E.; & de Brabander, B. (1975). The Organization of Electronic Data Processing (EDP) Activities and Computer Use. *Journal of Business, 48(2)*, 391-410.

Verplank et al., (1993).

Virzi, R.; Sokolov, J.; & Karis, D. (1996). Usability problem identification using both low- and high-fidelity prototypes. In *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground*.Vancouver, British Columbia, Canada. 236 – 243.

Vredenburg, K.; Mao, J.;, Smith, P.; & Carey, T. (2002). A Survey of User-Centered Design Practice. In *Proceedings of Computer Human Interaction CHI2002,* April 20-25, 2002, Minneapolis, Minnesota, USA, 471-478.

Weidenhaupt, K; Pohl, K; Jarke M; & Haumer P. (1998). CREWS team. Scenarios in system development: current practice. *IEEE Software 15(2)*, 34–45.

Wiegers, K. (1997). Use Cases: Listening to the Customer''s Voice. *Software Developmet 5*, 49-62.

Wixon, D.; Jones, S.; Tse, L.; & Casady, G. (1994). Inspections and design reviews: Framework, history, and reflection. In *Usability Inspection Methods*, J. Nielsen and R. Mack, eds. New York: John Wiley and Sons.

Woodcock, J.; & Davies, J. (1996). *Using Z: Specification, Refinement and Proof*. Prentice-Hall, London, UK.

Yourdon, E. (1994). *Object-Oriented Systems Design: An Integrated Approach*. Prentice Hall PTR, Yourdon Press Series, Englewood Cliffs, New Jersey.

Zhu, H.; & Jin, L. (2000). Scenario Analysis in an Automated Tool for Requirements Engineering. *Journal of Requirements Engineering 5*, 2-22.

Zowghi, D.; Gervasi, V.; & McRae, A. (2001). Using Default Reasoning to discover inconsistencies in Natural Language Requirements. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference*, Macau, China, December 2001.