Requirements Engineering with Use Cases – a Basis for Software Development

Björn Regnell

LUND UNIVERSITY

Department of Communication Systems
Lund Institute of Technology

ISSN 1101-3931
ISRN LUTEDX/TETS--1040--SE+225P
© Björn Regnell

Printed in Sweden
KF-Sigma
Lund 1999

to the memory of Ann-Christin

This thesis is submitted to Research Board II – Physics, Infomatics, Mathematics and Electrical Engineering (FIME) – at Lund Institute of Technology (LTH), Lund University, in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering.

Contact Information:

Björn Regnell
Department of Communication Systems
Lund University
P.O. Box 118
SE-221 00 LUND
Sweden

Tel.: +46 46 222 90 09
Fax: +46 46 14 58 23
email: Bjorn.Regnell@tts.lth.se
http://www.tts.lth.se/Personal/bjornr

Abstract

Successful development of software systems depends on the quality of the requirements engineering process. Use cases and scenarios are promising vehicles for eliciting, specifying and validating requirements. This thesis investigates the role of use case modelling in requirements engineering and its relation to system verification and validation. The thesis includes studies of concepts and representations in use case modelling. Semantic issues are discussed and notations based on natural and graphical languages are provided, which allow a hierarchical structure and enable representation at different abstraction levels.

Two different strategies for integrating use case modelling with system testing are presented and evaluated, showing how use cases can be a basis for test case generation and reliability assessment. An experiment on requirements validation using inspections with perspective-based reading is also reported, where one of the perspectives applies use case modelling. The results of the experiment indicate that a combination of multiple perspectives may not give higher defect detection rate compared to single perspective reading. Pilot studies of the transition from use case based requirements to high-level design are described, where use cases are successfully applied for documenting how functional requirements are distributed on architectural elements. The investigation of an industrial requirements engineering process improvement programme is also reported, where the introduction of a release-driven prioritisation method contributed to a measurable improvement in delivery precision and product quality.

The results presented in the thesis provide further support for how to successfully apply requirements engineering with use cases as an important basis for software development.

Acknowledgements

This work was financially supported by the National Board for Industrial and Technical Development, (NUTEK), Sweden, References 93-02850 and 1K1P-97-09690.

I am deeply indebted to many persons who have provided help, support and encouragement. First of all, I would like to thank my supervisor Prof. *Claes Wohlin* for his invaluable help and unselfish support throughout the preparation of this thesis. Thank you Claes, for your dedication and confidence in my work! Warm thanks also to Dr. *Per Runeson*, for our co-operation and fruitful discussions. I would also like to thank my other colleagues and dear friends in the Software Engineering Research Group: *Anders Wesslén*, *Martin Höst*, *Magnus C. Ohlsson, Thomas Thelin, Håkan Petersson, Lars Bratthall, Tomas Berling*, and *Enrico Johansson*. I would also like to thank the head of department, Prof. *Ulf Körner*, for his caring and optimistic leadership, and the department's head of teaching, Dr. *Lars Reneby*, for his encouragement and recognition of my teaching work, and all others at the department for providing such a nice working environment. I thank *Kristofer Kimbler* for our co-operation and for raising my interest in use cases, and *Daniel Søbirk* for study companionship at LTH.

A million thanks to my industrial partners for their fruitful co-operation, without which this thesis would have been much less interesting: *Per Beremark*, *Ola Eklundh*, *Mikael Andersson*, *Johan Bergstrand*, *Anders Ek*, and *Niklas Landin* at Telelogic, and to *Åke Davidson, Kristian Sandahl* and *Erik Linderoth-Olson* at Ericsson Software Technology. A very special thanks to Dr. *Joachim Karlsson* at Focal Point, for reviewing my work, for sharing his enormous requirements engineering wisdom, and for being such a good friend. I would also like to thank all fellow researchers around the world who I have met during conferences and workshops for many fruitful discussions and valuable input to my work, especially the participants at the excellent REFSQ workshops organised by Dr. *Klaus Pohl*, Dr. *Andreas Opdahl* and Dr. *Eric DuBois*. Many thanks to Dr. *Shailey Minocha* and Dr. *Neil Maiden* at City University for very interesting discussions.

Finally, I would like to express my warmest gratitude to my family and all my friends for making my life rich and joyful. Warm thanks to my oldest friend *Josefin* for always being there, and to my friend and study-mate in philosophy *Anna-Sofia* who also reviewed parts of this thesis. I thank my grand-mother *Anne-Marie*, and *Ann & Lasse* for their care and love. Thanks also to *Johan*, *Mats*, *Natasha*, and *Ingela*, for their invaluable love and companionship. I also express my heartfelt thanks to my dear father *Kai* for sharing his life-wisdom and giving crucial advise in difficult situations – without his support this thesis would never have been completed. I give my greatest love to *Susanne* for sharing her life with me.

Contents

List of Papers

The following papers are included in the thesis:

[I] **Improving the Use Case Driven Approach to Requirements Engineering**
Björn Regnell, Kristofer Kimbler and Anders Wesslén
Proceedings of *Second IEEE International Symposium on Requirements Engineering* (RE'95), York, UK, March 1995.

[II] **A Hierarchical Use Case Model with Graphical Representation**
Björn Regnell, Michael Andersson and Johan Bergstrand
Proceedings of *IEEE International Symposium and Workshop on Engineering of Computer-Based Systems* (ECBS'96), Friedrischshafen, Germany, March 1996.

[III] **Towards Integration of Use Case Modelling and Usage-Based Testing**
Björn Regnell, Per Runeson and Claes Wohlin
Accepted for publication in *Journal of Systems and Software*, Elsevier.

[IV] **Derivation of an Integrated Operational Profile and Use Case Model**
Per Runeson and Björn Regnell
Proceedings of *9th International Symposium on Software Reliability Engineering* (ISSRE'98), Paderborn, Germany, November 1998.

[V] **Are the Perspectives Really Different? – Further Experimentation on Scenario-Based Reading of Requirements**
Björn Regnell, Per Runeson, and Thomas Thelin
Technical Report CODEN:LUTEDX(TETS-7172)/1-40/1999 & local 4,
Submitted to the journal of *Empirical Software Engineering*, Kluwer.

[VI] **From Requirements to Design with Use Cases – Experiences from Industrial Pilot Projects**
Björn Regnell and Åke Davidson
Proceedings of *3rd International Workshop on Requirements Engineering - Foundation for Software Quality* (REFSQ'97), Barcelona, Spain, June 1997.

[VII] **A Market-Driven Requirements Engineering Process – Results from an Industrial Process Improvement Programme**
Björn Regnell, Per Beremark and Ola Eklundh
Journal of *Requirements Engineering* 3:121-129, Springer-Verlag, 1998.

Related Publications

The following publications are related but not included in the thesis:

[VIII] **User-Centred Software Engineering**
– A Comprehensive View of Software Development
Claes Wohlin, Björn Regnell, Anders Wesslén and Henrik Cosmo
Proceedings of *Nordic Seminar on Dependable Computing Systems* (NSDCS'94), Lyngby, Denmark, 1994.

[IX] **Combining Scenario-based Requirements with**
Static Verification and Dynamic Testing
Björn Regnell and Per Runeson
Proceedings of *4th International Workshop on Requirements Engineering - Foundation for Software Quality* (REFSQ'98), Pisa, Italy, June 1998.

[X] **An Evaluation of Methods for Prioritizing Software Requirements**
Joachim Karlsson, Claes Wohlin and Björn Regnell
Journal of *Information and Software Technology* 39(14-15):939-947, 1998.

[XI] **Reliability Certification of Software Components**
Claes Wohlin and Björn Regnell
Proceedings of *Fifth International Conference on Software Reuse* (ICSR'98), Victoria, Canada, June 1998.

[XII] **A Project Effort Estimation Study**
Magnus C. Ohlsson, Claes Wohlin and Björn Regnell
Journal of *Information and Software Technology* 40(14):831-839, 1998.

[XIII] **Achieving Industrial Relevance in Software Engineering Education**
Claes Wohlin and Björn Regnell
Proceedings of 12th *Conference on Software Engineering Education &Training* (CSEE&T'99), New Orleans, USA, March 1999.

Introduction

Software is intangible and immaterial. While physical constructions have properties that we can observe directly with our senses, software systems cannot be weighed, touched or smelled. A software system can only be observed through the linguistic representations that we make of it or through the effects it produces when it is used. In addition, many software systems of today are very “large” and of immense complexity. If we, for example, would print on paper the source code for the software that operates in a public telephone exchange, using the format of normal program listings, the printing would be several miles long. We should also bear in mind that such a complex system requires thousands of engineers involved in its development and subsequent modification.

A software system is, due to these facts, very difficult to conceptualize and communicate. Still, if we cannot define what we want from a software system in a precise and comprehensible manner, it is not very easy for the software engineers to know what to build and to verify that a system fulfils what was wanted from it. If we cannot capture the expectations on a system before it is implemented, it is very likely that the system will be a disappointment.

This thesis concentrates on how to elicit, specify and validate the requirements of a software system to be constructed. These activities are carried out within the discipline called *requirements engineering*. The research presented here aims at improving a particular technique in requirements engineering, called *use case modelling*, and to integrate this technique with other techniques used for the *testing* of software systems.

The thesis includes a collection of seven papers and is organized in the following four parts:

- **Introduction**. The introduction gives a background to the presented papers. Section 1 presents the research focus and states the research questions. Section 2 explains the research methods used in relation to the research questions and the presented papers. Section 3 gives an overview of related work. Section 4 summarises the main contributions of the research.
- **Concepts and representations of use cases**. This part includes two papers regarding semantical and syntactical issues in use case modelling. Paper I explains the need for adding structure to use cases and proposes a way to build an integrated model of use case fragments. Paper II presents a conceptual framework for use case modelling and proposes a graphical representation with support for different abstraction levels.
- **Integrating use cases with verification & validation**. This part includes three papers regarding the role use cases can play in testing and inspection. Paper III provides an investigation of how to utilize the information in use case models for statistical usage testing. Paper IV describes how use cases can be a basis for operational profile testing. Paper V reports on the results from an experiment with a specific inspection technique used for validating requirements documents, called Perspective Based Reading, in which use cases are applied as one way of finding requirements defects.
- **Industrial requirements engineering**. This part includes two papers regarding industrial process improvement efforts in requirements engineering. Paper VI reports on experiences from a sequel of industrial pilot projects where use cases were used in the transition from requirements to high-level design. Paper VII analyses the benefits of an industrial requirements engineering process, and outlines how some challenges can be addressed with use cases.

1. Research Focus

The research presented in this thesis is carried out within the discipline of *software engineering*, concerned with methods, tools, and techniques for developing and managing the process of creating and evolving software products (Sommerville, 1996). Within software engineering, a sub-discipline has grown during this decade, called *requirements engineering*, which focuses on the early phases of software development where decisions are made on *what* to implement by the software system and the foundation is laid for the later phases that determine *how* to implement it.

The software industry is perhaps the fastest growing and most profitable industry in the world and software has had a profound impact on our society. During the last decades, software systems have become more and more complex, and many vital functions of our society now depend on software systems. Development of large and complex software systems, however, is intrinsically difficult. It is unfortunately common that software systems are delivered with poor quality, too late, and over budget. The term “software crisis” was coined already in the late 1960’s (Naur & Randell, 1969), but even in the 1990’s, the crisis is still present (Gibbs, 1994). No single technique or method has yet been discovered that overcomes all the essential difficulties in the creative process of large-scale software development (Brooks, 1987). Thus, we have to apply a number of different techniques and methods in this process to get complex software systems with adequate quality delivered in a predictable time and to a predictable cost.

In large software development projects, it is neither feasible, nor desirable, to start directly with coding; several activities are needed before a solution can be implemented. After the implementation is ready, activities have to be carried out to assure and assess its quality and when the product has been delivered, it has to be maintained as all significant software systems are likely to need changes. New services are introduced into the system as its environment changes and its users require more functionality. In order to manage large software development projects, that either develop new systems from scratch or enhance existing systems, we need to follow a well defined process.

The waterfall model (Royce, 1970) is often used to illustrate the process of large-scale software development. It views the development as a stepwise transformation from the problem domain to the solution through a number of phases. The model has been criticized for being too simplistic and not taking iterations and other aspects into account (Boehm, 1988), but variants of the model are still used to illustrate the software development process. The major activities can be summarized as follows:

- *Requirements Engineering*. Requirements on the software system are elicited, specified and validated.
- *Design*. The architecture of the software system is determined, and the system is modularised into coherent subsystems (components).
- *Implementation*. Executable components are developed which implement the design.
- *Component Verification*. The components of the system are verified to ensure that they are implemented according to the design.
- *System Verification & Validation*. The system is verified to assure that it fulfils its specification, and validated against the users' and customers' needs and expectations. Quality aspects, such as reliability, are assessed.

This thesis concentrates on improvements to a specific technique for requirements engineering called *use case modelling*, and how this technique can be integrated with the verification & validation phases. The subsequent sections give a background to requirements engineering, use case modelling, and verification & validation respectively. Section 1 is concluded by stating the research questions investigated in this thesis.

1.1 Requirements Engineering

The two major objectives of requirements engineering are (1) to understand the problem that the intended system is supposed to solve, and (2) to select and document the requirements on the system and its development. The products of requirements engineering are the foundation for the whole subsequent development. In particular, we want to create a solid basis for:

- Planning and cost estimation,
- Design and architectural decisions,
- Verification and validation.

A major motive 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. This objective is justified by empirical evidence supporting the following hypotheses (Davis, 1993):

- Many requirements errors are being made.
- Many of these errors are detected late.
- Many of these errors can be detected early.
- Not detecting these errors may contribute to dramatic increase of software costs.

Experience shows that the cost of detecting and repairing errors increases dramatically as the development process proceeds. Table 1 shows a compilation of three empirical studies, indicating that it may be up to 200 times more expensive to detect and repair errors in the operation stage, compared to detecting and repairing them during the requirements stage (Davis, 1993).

With these figures in mind, it is reasonable to believe that efforts spent on improving requirements engineering will pay off.

Table 1. Relative cost of error repair in different development stages.

Stage	**Relative cost of error repair**
Requirements	0.1 - 0.2
Design	0.5
Implementation	1
Component Verification	2
System Validation	5
Operation	20

Webster's Dictionary (1989) defines a requirement as "something required; something wanted or needed". A more elaborated definition, specific to software systems, can be found in the IEEE Standard 610 (1990), where 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 possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

We can see that the word requirement can mean both a *desired* property and an *obligatory* property. This reflects the *decision process*, inherent to requirements engineering; we need not only to find all desires, we also have to decide which of the, possibly conflicting, desires to be implemented.

The documented representation of a requirement is also, for short, termed just requirement. This indicates the *specification process* within requirements engineering, as we need to document the selected requirements together with the reasons for our decision so that the rationale for the requirements can easily be understood.

Those that have interests in the process of requirements engineering are called *stakeholders*, including, for example, requirements holders (customers, procurers, sponsors, end-users, etc.) and developers (requirements engineers, designers, testers, managers etc.).

The *product* of requirements engineering is called *requirements specification*. This term is also used for the *process* of documenting requirements.

The process of requirements engineering is not a simple succession of demarcated activities. Instead, it is inherently iterative and consists of a number of interrelated subprocesses. Loucopoulos & Karakostas (1995) identify the following interrelated subprocesses:

- *Elicitation*: understand the problem, identify the stakeholders and capture the requirements.
- *Specification*: describe the problem and document the requirements.
- *Validation*: ensure that the specified requirements agree with the stakeholders' expectations.

Figure 1 gives an overview of the information flow between these processes. This picture does, however, not take into account other important subprocesses, such as the management of requirements changes or the prioritisation of requirements. Karlsson & Ryan (1998), for example, stress the need for incorporating a structured requirements selection process

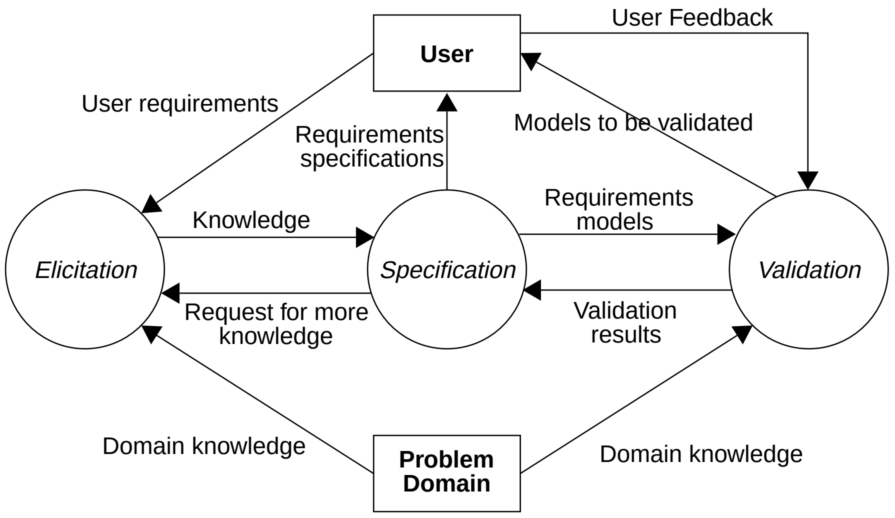


Figure 1. *Processes in requirements engineering (Loucopoulos & Karakostas,1995).*

into requirements engineering, where requirements may be prioritised based on pair-wise comparison of requirements, using the criteria of value and cost. Requirements are selected so that the system will give a high value to an acceptable cost.

A major challenge is to know when to stop requirements analysis, and unfortunately it is not easy to define an objective stop criterion. It is up to the judgement of the involved stakeholders to determine if the requirements documentation is good enough to proceed with design. The subsequent phases in the development process are likely to demand changes in the requirements document, and after delivery, the system is likely to need changes caused by changing requirements. Hence, the requirements engineering process must continue, in some form, throughout the entire lifetime of the system.

1.2 Use Case Modelling

The main concepts in use case modelling are actors and use cases. According to Jacobson et al. (1992), an *actor* represents a certain user type or a role played by users. A *use case* is a specific way of using the system, viewed as a set of related transactions performed by an actor and the system in dialogue.

As an example, consider an automated teller machine, where users can retrieve money and check how much money they have in their bank account. In this system we can identify the following actors:

- *Customer* – the normal user role with the goals of withdrawing cash and checking the account balance.
- *Supervisor* – the actor which supervises and maintains the machine, including money and receipt paper refilling.
- *Database* – the central banking system maintaining and updating bank account information.

The last actor is an example of a user type where the role is not played by humans, but instead by another software system.

Examples of use cases for the *customer* actor are *withdraw cash* and *check balance*. These use cases typically include events such as *insert card*, *display message*, *enter code* etc.

As another example, consider a telephony system. Here we may identify actors such as *subscriber* and *operator*, and the use cases *make call* and *add new subscriber*. The use case *make call* typically includes events such as *lift receiver*, *send dialtone*, *dial number*, etc.

Another example is an ordinary, single user word processor system, where the *writer* actor may, for example, perform the use case *spell checking*, including events such as *select paragraph*, *select dictionary*, *display misspelling*, *learn word*, etc.

Use cases may have different variants or alternative courses, depending on the different circumstances of system usage. For example, in the *withdraw cash* example use case there may be different courses depending on, for example, if the code is entered correctly or not. A single sequence of events representing such specific realisations of a use case is often termed a *scenario*. Hence, a use case is often said to cover a set of scenarios.

The term use case was coined by Jacobson et al. (1992), and introduced in an object-oriented context. Object orientation provides concepts for defining and relating objects. These objects can both be entities of the problem domain and entities existing as parts of a software system. Many object-oriented methods demarcate between analysis and design, where analysis is focused on problem domain objects and design is focused on system internal objects. Traditional object-oriented analysis, however, can only partially capture requirements, as object models, focused on inheritance and aggregation, mainly are static views of the system and its problem domain. Requirements are, however, to a large extent functional and not solely object-oriented, as they define a system's functional properties also with regard to dynamic issues. Several object-oriented methods have, in consequence, incorporated dynamic modelling based on the concept of use cases in order to bridge the gap between a functional view in requirements analysis and an object-oriented view in architectural analysis and design.

Although often combined with object orientation, use case modelling is a general technique that may be applicable as a front-end to any design method. The basic idea of modelling usage from an external point of view by describing different usage scenarios, is often practised within requirements engineering, sometimes under the name scenario-based requirements analysis (see e.g. Carroll, 1995).

Paper I and II concentrate on improvements to use case modelling with respect to their meaning and representation. In paper VI, an industrial case study is presented, where use cases are used as a vehicle in the transition from requirements to high-level design.

1.3 Verification & Validation

As the software development process is human intensive and error prone, it is important to continuously assure that the interim and final products of the development are of adequate quality. Verification & Validation (V&V) refer to activities that test the quality of a product for the purpose of finding and correcting defects and deciding if the development can continue or some parts must be subjected to rework. Verification refers to a narrow frame of testing, where a product is checked if it is correct in terms of previous products. An example of verification is when an implemented software component is executed using a set of test cases to see if it fulfils the component design. Validation is a broader type of evaluation, where the product is checked against the original stakeholders' views. An example of validation is when the final integrated system is tested in an acceptance test, to see if it corresponds to the original intentions.

V&V strategies may be divided into two main classes: (1) static V&V including V&V methods that do not execute the artifact under scrutiny, and (2) dynamic V&V including methods that exercise a software artifact by executing it with sample input data. These two classes can be broken down further into different types of methods. Figure 2 shows a partial[1] classification of V&V methods.

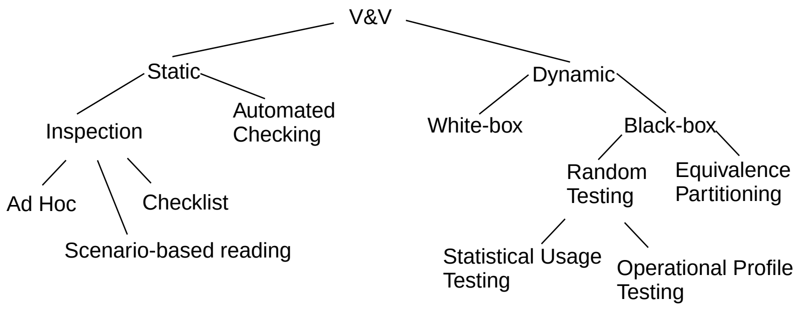


Figure 2. *Classification of verification & validation techniques.*

Static V&V in turn may be divided into automated checking and inspections. Automated static checking is based on formal languages and compiler technology and may reveal defects based on the syntax and semantics of the formal languages used.

Inspection, on the other hand, is a *manual* approach, where humans *read* the documentation of a software artifact with the objective of finding and understanding defects. Reading techniques are suitable for documents containing natural language, common in requirements documents, but also applicable when evaluating, for example, the quality of design and code. This thesis includes an investigation of a reading technique for requirements inspection that involves use cases (Paper V).

Dynamic testing can be divided into black-box and white-box techniques, where *black-box* refers to functional testing with an external perspective and *white-box* refers to structural testing with an internal

1. The figure only includes a few examples of each class. A more detailed survey of existing techniques is available in (Graham, 1994).

perspective. In white-box techniques, the test cases are defined based on structural aspects, such as path coverage.

In black-box techniques, test cases are derived from a system specification, and hence have a natural connection to requirements. One example of a black-box technique is equivalence partitioning (see e.g. Sommerville, 1996), where the test cases are based on classes of input and/or output values of a system.

Another example of a black-box V&V technique is random testing, where test cases are selected according to some probability distribution. This type of testing is motivated by the need for making test cases resemble system operation. The reliability of a system depends not only on the number of defects in a software product, but also on how it is executed in operation. This implies that reliability testing must resemble operational usage, i.e. test cases are selected according to a usage profile.

This thesis includes investigations of how usage models for reliability testing can be constructed based on use cases (Papers III, IV). The three papers III, IV & V are based on a general investigation of use cases and V&V integration opportunities provided in (Regnell & Runeson, 1998).

1.4 Research Questions

The presented thesis project started in November 1993. The main vision of the project was to work with methods and techniques which support a user-centred approach to software engineering, where the user perspective is in focus throughout the life-cycle (Wohlin et al., 1994). This vision aims at continuous visibility and evaluation by the customers and users, which in turn is assumed to help with keeping software development on track for the successful delivery of systems that comply with the original needs and expectations.

There are inherent connections between Requirements Engineering (RE) and Verification & Validation (V&V). Both RE and V&V view the system under development at a higher abstraction level compared to design and implementation, and both disciplines have an external view of the system (see Figure 3), where the system usage is in focus, rather than its internal structure. In both the RE and V&V research communities, there exist concepts related to system usage, namely the *use case* concept in RE and the concept of *usage testing* in V&V. At the beginning of this decade, little effort was paid to the integration of these concepts, and research questions were formulated for the investigation of such an integration.

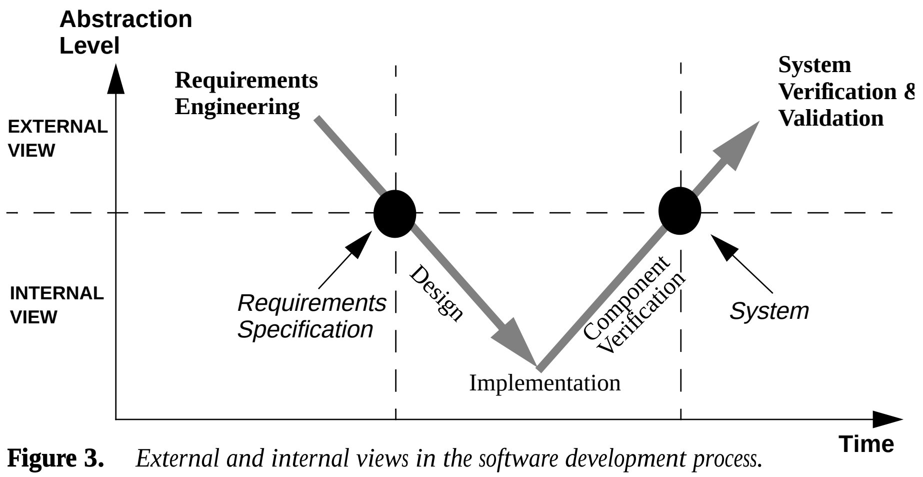


Figure 3. *External and internal views in the software development process.*

Before an integration between use case modelling and usage testing can be achieved, it is important to investigate the properties of each of these concepts in order to see which parts best fit together. The first two papers in this thesis are focused on research questions related to use case modelling itself, as the investigation presented by Wohlin et al. (1994) identified several open issues regarding how to apply use cases. It was assumed that a more structured approach to use case modelling was needed in order to facilitate an integration with usage testing.

The investigation of use case modelling is based on questions regarding the representation of use cases, the abstraction level of use case representations, and their understandability to laymen.

The thesis project has a broad interest in requirements engineering in general, and use case modelling in particular. Other requirements engineering related issues have been investigated, such as requirements validation through inspections and requirements process improvements.

In summary, four major research questions have formed the basis for the presented work:

RQ1. How can the semantics and syntax of use case representations be improved in order to allow different degrees of formality, different levels of abstraction, and easy comprehension?

RQ2. How can use case modelling be integrated with system testing, so that the usage information contained in use case models can be utilised for test case creation and assessment of reliability?

RQ3. How does the user perspective perform in requirements validation through inspection?

RQ4. What have been the outcomes of requirements engineering improvement efforts in industry?

2. Research Methods

Software engineering has a short history and is still maturing as a research discipline. It is therefore not surprising that the research community is debating how to conduct research in software engineering and requirements engineering (Potts, 1993; Fenton, et al., 1994; Glass, 1994; Ryan, 1995). Software engineering emerged as a complement to computer science, which in turn has grown out of a sub-discipline of mathematics. Although software engineering research deals with real world problems involving humans and organisations, a great deal of software engineering research is, perhaps for historical reasons, conducted in the analytical tradition of mathematics, where formal (mathematical) reasoning is the main starting point. The research presented in this thesis is, however, less based on formal mathematics, and more centred around the investigation of informal and semi-formal techniques aimed at solving practical problems faced by the software industry.

The humans involved in and affected by software development are central to software engineering. Hence, it may be wise to investigate the research approaches used in behavioural sciences which have a longer tradition of research with humans as subjects. Robson (1993) gives a classification of behavioural science research approaches on an ordinal scale from basic research to applied investigations:

- A. *The traditional 'science only' approach*, including (1) basic research where application to problem-solving in the real world is not usually seen as an objective; (2) less basic, but still 'pure' or 'theoretical' where application is not a high priority; (3) research on practical problems where application is seen as possible but not a necessary outcome.
- B. *Building bridges between researcher and user*, where (1) the researcher believes that the work has practical implications and should be used and seeks to disseminate results widely and in an accessible language; (2) the researcher obtains client collaboration on researcher-designed projects and would like the client to be influenced by the outcome of the research; (3) in addition to B2 the researcher takes steps to give the client regular feedback on progress, problems and outcomes and during this feedback, the client has an

opportunity to check on interim findings and contribute with the client's own analysis and interpretation and the researcher helps in the practical implementation of the research results.

- C. *Researcher-client equality.* (1) The researcher and client together discuss problem areas and jointly formulate the research design and actively collaborate involving some measure of control on part of the client, including implementation of results; (2) As in C1, but the initiative is taken by the client who identifies the problem and the researcher consider whether there are other issues which should receive primary attention; (3) As in C2, but the problem identified by the client is not questioned and research proceeds on that basis with the researcher paying most attention to implementation.
- D. *Client-professional exploration*. A client with a problem requests help from a researcher. Collection of new data (if any) is minimal. Advice or recommendation is based on the researcher's past experience and knowledge of the field. If this takes place in an organisation, then training or organisation development is a frequent outcome.
- E. *Client-dominated quest*. A client requests help from a specialist who examines the problem, interprets 'best current knowledge', makes a diagnosis and suggests a line of action.

The work presented in this thesis has, according to the above classification, been conducted with approaches in the classes from A3 to C1, indicating a clearly applied focus with the researcher in main control of the research design.

A classification of research methods in software engineering was proposed at a Dagstuhl Workshop (Adrion, 1993) and discussed by Glass (1994) and Zelkowitz & Wallace (1998). This classification identifies the following four research methods:

- *Scientific method*. Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis. An example is building a simulation model of a software development process based on mathematical relations between measurable entities and validate results from the simulations with empirical data from real projects; if the simulation model is valid it may be used for prediction purposes (see e.g. Donzelli & Iazeolla, 1996).

- *Engineering method*. In this evolutionary approach, engineers observe existing solutions, propose improvements and analyse and measure the improvements proposals; this scheme is repeated until the solutions do not need further improvements. An example is proposing improvements to a requirements engineering method and trying it out to evaluate the results; the evaluation is input to further improvement efforts (see e.g. Potts et al., 1994).

- *Empirical method*. Based on hypotheses, design a study, collect data, and test the hypotheses with quantitative (statistical) methods. Unlike the scientific method, there is not necessarily a formal model or theory describing the hypotheses. Empirical methods can be further categorised into three types (Robson, 1993):
  – *Experiment*: measuring the effect of manipulating one variable over another variable. An example is comparing the effectiveness of two different requirements inspection techniques (see e.g. Basili et al., 1996)
  – *Survey*: collection of information in standardized form from groups of people. An example is investigating current practice in requirements engineering through interviews with a number of companies (see e.g. Weidenhaupt et al., 1998)
  – *Case study*: development of detailed, intensive knowledge about a single case, or a small number of related cases. An example is studying a specific requirements engineering technique used in a real project (see e.g. Gough et al., 1995)

- *Analytical method*. A formal theory is developed, and results derived from that theory can be compared with empirical observations. An example is using formal reasoning in requirements specification (see e. g. Du Bois et al., 1997; Desharnais et al., 1998).

The work presented in papers I, II, III, and IV represents the engineering research method, as new improvement proposals on existing techniques are investigated. Paper V represents the empirical research method, in which a controlled experiment is performed. Paper VI and VII represents the empirical research method as case studies of specific industrial requirements engineering methods are investigated.

The approaches and methods used in this thesis are summarised in Table 2.

Table 2. Mapping between research questions, papers, approaches and methods.

Research Questions	Papers	Approaches	Methods
RQ1	I, II	B1, B2	Engineering
RQ2	III, IV	B1	Engineering
RQ3	V	A3	Empirical: experiment
RQ4	VI, VII	C1	Empirical: case study

The work presented here is conducted by engineers with a strong urge to measure and quantify. Although, the search for objective and quantifiable facts is considered as essential, there is in engineering also a pragmatic attitude to the problems in focus; if a method works, then use it, even if there is no rigorous scientific explanation to why it works or any certain quantification of how effective the method is. The use of qualitative methods in software engineering is often not as structured as in the more mature behavioural sciences, but qualitative reasoning is often used by engineers.

During the presented research project, there have been many occasions where a clear cut, objective answer in quantifiable terms has not been found. Other parts of the problem space have been investigated with more scientific rigour. Still, all research results presented in the thesis are founded in the dedicated quest for solutions to industrial software engineering problems.

3. Related Work

The research contributions presented in this thesis are enhancements of previous work in the area of requirements engineering and verification & validation. This section puts the presented work into context, points out some important publications in the field, and identifies a number of sources that have been major inputs to the study of the research questions.

This thesis represents an effort of combining and integrating different areas of software engineering through the use case concept, as the research questions RQ2 & RQ3 in Section 1.4 suggest. The presentation of related work is consequently divided into three subsections, describing use cases, testing, and inspections, respectively.

3.1 Use cases and Scenarios

The work in this thesis was initially based on the use case concept coined by Jacobson et al. (1992). In the papers of this thesis we use the terminology that stems from Jacobson, where a “scenario” is specialisation of a “use case”. Now, the term “scenario” has gained widespread acceptance as a general concept denoting a sub-discipline of requirements engineering rather than a specific, well-defined technique.

The number of research publications in the area has grown and industry is applying scenarios and use cases in a plurality of ways. A large EU-funded basic research action started in 1996 called CREWS (An ESPRIT 4th Framework Programme, No. 21.903), which stands for “Cooperative Requirements Engineering With Scenarios”, and which have had a major impact on the requirements engineering community. An important concept-building CREWS report is the Scenario Classification Framework (Rolland et al., 1998) – subsequently denoted CREWS-SCF – which also includes an extensive literature study and the classification of 12 scenario approaches according to this framework. A scenario approach is classified in terms of its contents, form, purpose and role in the requirements engineering life cycle. An overview of the CREWS-SCF is given in Figure 4.

Paper I is classified by Rolland et al. (1998) according to the CREWS-SCF, and it is concluded that the approach in Paper I differs from Jacobson et al. (1992) with respect to form, contents and purpose, as more structure to use case modelling is applied, a more advanced graphical notation is provided, and the purpose is primarily focused on requirements specification rather than identification of design objects. Paper II is

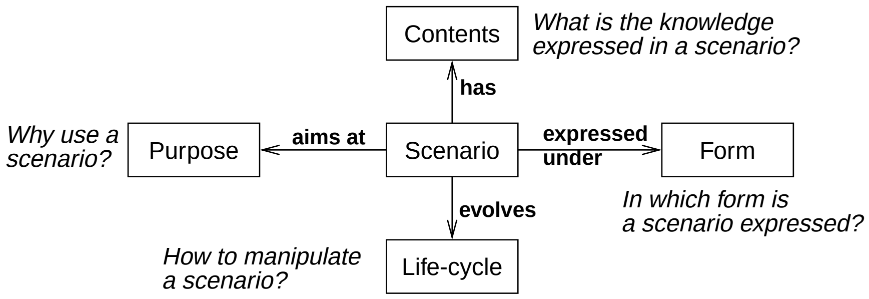

Contents
What is the knowledge expressed in a scenario?
has
Why use a scenario?
Purpose
aims at
Scenario
expressed under
Form
evolves
In which form is a scenario expressed?
How to manipulate a scenario?
Life-cycle

Figure 4. *Overview of the CREWS Scenario Classification Framework (Rolland et al., 1998).*

in essence similar to Paper I in terms of the CREWS-SCF, although the graphical representation is more developed and extensions are proposed in relation to the standard language of Message Sequence Chart (MSC, 1993) ITU Recommendation Z.120. Many of the ideas in Paper II are now incorporated in the latest version of MSC (1996).

The CREWS-SCF shows that the views of scenarios may vary to a large extent and that many different ways of using scenarios are possible. For example, Gough et al. (1995) present a use case approach where use cases are expressed using visual representations with multimedia and hypertext animations, while Hsia et al. (1994) present a formal but non-executable, grammar-based approach to scenario generation and analysis. Potts et al. (1994) propose textual scenarios and tabular notations and integrate scenarios with an inquiry cycle where open questions, responses and arguments are tracked together with change requests.

The plurality of scenario applications is further emphasised by the survey of current industrial practice described in another CREWS report (Weidenhaupt et al., 1998), where 15 projects in 4 European countries are classified according to the CREWS-SCF. In this survey it is concluded that inspections in conjunction with scenarios are commonly used, which is a motivation from practice for the study in Paper V.

Another conclusion from the survey is that nearly all developers mention the need to base system tests on scenarios, but current practice rarely satisfies this demand as most projects lack a systematic approach for defining test cases based on scenarios. Paper III & IV provide an investigation of such systematic approaches, providing methods to integrate use case modelling with testing.

In addition, four types of scenario evolution where encountered by the survey: (1) top-down decomposition of scenarios, (2) from black-box to white-box scenarios, (3) from informal to formal scenario definitions, and (4) incremental scenario development. The first two evolution types are discussed in Paper VI. The third evolution type is discussed in Paper I & II. Leite et al. (1997) discuss the management of incremental scenario development in conjunction with a requirements baseline, which in turn may fit well to the use case approach in Paper I & II.

Several object-oriented methods have incorporated dynamic object modelling with use cases, as a complement to the static view in traditional object models. One of the early publications on dynamic object behaviour analysis is Rubin & Goldberg (1992), who propose object scripts in tabular form which resemble use cases. Paper II presents a comparison of object-oriented use case approaches including OOSE (Jacobson et al., 1992), OMT (Rumbaugh et al., 1991), the Booch method (Booch, 1994), and ROOM (Selic et al., 1994). The Jacobson, Rumbaugh, and Booch approaches are now integrated in the Unified Modelling Language (UML).

In Paper II, the goal concept (Dardenne et al., 1993) is pointed out as useful in the identification and definition of use cases. Several authors agree with the importance of combining use cases and goals (see e.g. Cockburn, 1997a; Cockburn, 1997b; van Lamsweerde & Willemet, 1998). Anton et al. (1994) investigate the derivation of goals from scenarios.

Paper I & II are mainly interested in functional requirements, while Paper III & IV focus on one non-functional requirement, namely reliability. Sutcliffe & Minocha (1998) provide a general investigation of how scenarios can be utilised in the analysis of non-functional requirements.

3.2 Usage-based Testing

Non-functional requirements are an essential part of requirements specifications. In particular, the reliability requirements are often regarded as one of the most important non-functional requirements. Software *reliability* is in the IEEE Standard 610 (1990) defined as “the ability of a system or component to perform its required functions under stated conditions for a specified period of time”. As the definition indicates, reliability is dependent on the circumstances of operation. When certifying that a system fulfils a specified reliability requirement, it is in consequence necessary to specify under which circumstances this reliability requirement is to be met. Thus, there is a need to model system usage and to quantify the probability of different usages in order to generate test cases that resemble the circumstances in the reliability requirement. As in all testing, the testing budget is limited, making it impossible to test all possible cases, which in turn implies that a subset of all test cases must be selected. A common criterion for selecting test cases is their expected frequency of usage.

It can thus be concluded that both the objective of prioritising among possible test cases and the objective of certifying reliability, results in the need for a system usage model. Usage-based testing implies a focus on detecting the faults that cause the most frequent failures, hence maximising the growth in reliability.

There are two main approaches on probabilistic models of system usage: (1) Statistical Usage Testing (SUT) based on Markov models as introduced by Mills et. al. (1987) and further developed by Whittaker & Thomason, where test cases are generated from a state-machine model with probabilities attached to transitions between externally observable system states; and (2) Operational Profile Testing (OPT) as proposed by Musa (1993), where test cases are sorted in expected frequency order based on a functional profile combined with an operational profile including quantification of usage frequencies. A further development of SUT is presented by Runeson & Wohlin (1992, 1995), introducing the state-hierarchy (SHY) model in order to tackle the problem of scalability of Markov models.

Common to the mentioned techniques is that the approaches include two basic parts of usage specification: (1) a *usage model* including the *structural* aspects of usage in terms of externally observable states or system functions, and (2) a *usage profile* including the *statistical* aspects of usage in terms of frequency or probabilities.

Paper III is based on the SHY model, and shows how SUT can be combined with use case modelling, while Paper IV is based on OPT and presents ways of creating an operational profile from use cases. Another approach for generating operational profiles from use cases is proposed by Denney (1998). This approach uses the regular expression based use case notation from the Fusion object-oriented method (Coleman et al., 1994), complemented with statistical information.

3.3 Requirements Inspections

Tool support for requirements validation based on scenarios has been proposed by, for example, Maiden et al. (1998) and Sutcliffe et al. (1998). The validation of requirements documents is, however, often done manually, as requirements normally include informal representations, frequently based on natural language.

A commonly used technique for manual validation of software documents is inspections, proposed by Fagan (1976). Inspections can be carried out in different ways and used throughout the software development process for (1) understanding, (2) finding defects, and (3) as a basis for making decisions. Inspections can be used to find defects *early* in the development process, and have shown to be cost effective (e.g. Doolan, 1992).

A central part of the inspection process is the *defect detection* carried out by an individual reviewer reading the document and recording defects (a part of preparation, see Humphrey, 1989). Three techniques for defect detection are Ad Hoc, Checklist and Scenario-based reading (Porter, 1995). Ad Hoc detection denotes an unstructured technique, providing no guidance and the reviewers detect defects based on their personal knowledge and experience. The checklist detection technique provides a list of issues and questions, capturing the knowledge of previous inspections, helping the reviewers to focus their reading.

In the scenario-based approach, different reviewers have different responsibilities and are guided in their reading by specific scenarios which aim at constructing a model, instead of just passive reading. A scenario in this context denotes a script or procedure that the reviewer should follow.

There is a considerable risk for terminology confusion here, as the term *scenario* also is used within requirements engineering in general, and in this thesis in particular, to denote a sequence of events involved in an envisaged usage situation of the system under development, and a *use case* is often said to cover a set of related (system usage) scenarios. The term scenario in a scenario-based reading context is, however, a meta-level concept which denotes a procedure that a reader of a document should follow during inspection.

Two variants of scenario-based reading have been proposed: Defect-Based Reading (Porter, 1995) and Perspective-Based Reading (Basili, 1996). The former concentrates on specific defect classes, while the latter focuses on the points of view of the users of a document.

Another part of the inspection process is the *compilation of defects* into a consolidated defect list where all individual reviewers' defect lists are combined. This step may include the removal of false positives (reported defects that where not considered to be actual defects) as well as the detection of new defects. This step is often done in a structured *inspection meeting* in which the *team* of reviewers participates. The effectiveness of a team meeting has been questioned and studied empirically by Votta (1993) and Johnson (1998).

Paper V describes research on scenario-based reading with a perspective-based approach. The research method is empirical and includes a formal factorial experiment in an academic environment. The presented experiment is a partial replication of previous experiments in the area and focuses on refined hypotheses regarding the differences between the perspectives in perspective-based reading.

4. Research Results

The main results of the presented work include findings related to both use case modelling itself, the combination with verification & validation, as well as industrial application of requirements engineering and use cases. This chapter summarises the main contributions, and gives a guide to the reader on the contents of each included paper. Finally, an extensive list of further research is presented.

4.1 Main contributions

The main research contributions are summarised below. The results are grouped in relation to the research questions of Section 1.4.

- *Proposals on use case syntax and semantics* (RQ1). A conceptual study of use case modelling is provided including additional concepts, such as goal, service, episode, context and event. Definitions of and relations between the concepts are given. Proposals on how to give use cases more structure are given, and representation suggestions regarding both natural language and graphical descriptions are illustrated with examples. A hierarchical approach to use case modelling is presented as a means for allowing descriptions at different levels of abstraction. Extensions to the Message Sequence Chart language are provided, making it more suitable for graphical representation of requirements-level use cases.
- *Methods for applying use cases as a basis for testing* (RQ2). An investigation of the integration of use case modelling and usage-based testing is provided, and two specific methods of integration is proposed: extension and transformation. The two approaches are investigated in combination with both Statistical Usage Testing using the State Hierarchy model, and Operational Profile Testing. The results include a conceptual analysis of the terms from each domain and methods for deriving test models from use cases.
- *Evaluation of use cases in requirements inspection* (RQ3). An experiment is conducted, which evaluates perspective-based reading of requirements documents with three perspectives, including the user perspective applying use case modelling. The assumption of difference between perspectives is analysed and the results of the data

analysis show that (1) there is no significant difference between the three perspectives in terms of defect detection rate and number of defects found per hour, (2) there is no significant difference in the defect coverage of the three perspectives, and (3) PhD students with a checklist approach find significantly more defects per hour and have a significantly higher detection rate than MSc students with a PBR approach.

- *Experiences from industrial applications* (RQ4). Case studies in two different industrial environments are presented. The first case study includes a sequel of pilot studies where use cases are applied in the modelling of requirements distribution on architectural components. Use case modelling was found valuable as a tool in the transition from requirements engineering to design. A central observation is that the use cases are not decomposed in a strict top down manner and that the architectural decomposition is mainly carried out based on trade-offs between non-functional requirements. The second case study reports on a requirement process improvement programme in a market driven context, concluding that requirements prioritisation in combination with cost estimations and release planning have contributed to significant improvements of delivery precision and product quality. The evaluation of the case study includes the observation of a remaining challenge of how to implement a product strategy for a plurality of market segments, which possibly can be achieved using a use case based approach in combination with prioritisation based on pair-wise comparison.

4.2 Summary of Papers

The work presented in Paper I represents the earliest views of the research project. This paper is written in the bold spirit of providing a step towards the grand Method for use case modelling. As the investigation of use case modelling progressed, it became obvious that use cases can be applied in a number of ways depending on, for example, the application domain and the purpose, and that no modelling strategy is optimal in all situations. Hence, despite some rather categorical statements in Paper I, a more pragmatic view on formalisation in general and the use of specific constructs in particular is represented by the later papers.

The abstracts of each paper are provided below.

PAPER I: Improving the Use Case Driven Approach to Requirements Engineering

Björn Regnell, Kristofer Kimbler and Anders Wesslén

Proceedings of *Second IEEE International Symposium on Requirements Engineering* (RE'95), York, UK, March 1995.

This paper presents the idea of Usage-Oriented Requirements Engineering, an extension of use case driven analysis. The main objective is to achieve a requirements engineering process resulting in a model which captures both functional requirements and system usage aspects in a comprehensive manner. The paper presents the basic concepts and the process of Usage-Oriented Requirements Engineering, and the Synthesized Usage Model resulting from this process. The role of this model in system development, and its potential applications are also discussed.

PAPER II: A Hierarchical Use Case Model with Graphical Representation

Björn Regnell, Michael Andersson and Johan Bergstrand

Proceedings of *IEEE International Symposium and Workshop on Engineering of Computer-Based Systems* (ECBS'96), Friedrischshafen, Germany, March 1996.

This paper presents a conceptual framework for use case modelling and a new use case model with graphical representation, including support for different abstraction levels and mechanisms for managing large use case models. Current application of use cases in requirements engineering is discussed, as well as conceptual and methodological issues related to use case modelling.

PAPER III: Towards Integration of Use Case Modelling and Usage-Based Testing

Björn Regnell, Per Runeson and Claes Wohlin

Accepted for publication in *Journal of Systems and Software*, Elsevier.

This paper focuses on usage modelling as a basis for both requirements engineering and testing, and investigates the possibility of integrating the two disciplines of use case modelling and statistical usage testing. The paper investigates the conceptual framework for each discipline, and discusses how they can be integrated to form a seamless transition from requirements models to test models for reliability certification. Two approaches for such an integration are identified: integration by model transformation and integration by model extension. The integration approaches are illustrated through an example, and advantages as well as disadvantages of each approach are discussed. Based on the fact that the two disciplines have models with common information and similar structure, it is argued that an integration may result in coordination benefits and reduced costs. Several areas of further research are identified.

PAPER IV: Derivation of an Integrated Operational Profile and Use Case Model

Per Runeson and Björn Regnell

Proceedings of *9th International Symposium on Software Reliability Engineering* (ISSRE'98), Paderborn, Germany, November 1998.

Requirements engineering and software reliability engineering both involve model building related to the usage of the intended system; requirements models and test case models respectively are built. Use case modelling for requirements engineering and operational profile testing for software reliability engineering are techniques which are evolving into software engineering practice. In this paper, approaches towards integration of the use case model and the operational profile model are proposed. By integrating the derivation of the models, effort may be saved in both development and maintenance of software artifacts. Two integration approaches are discussed: transformation and extension. It is concluded that the use case model structure can be transformed into an operational profile model adding the profile information. As a next step, the use case model can be extended to include the information necessary for the operational profile. Through both approaches, modelling and maintenance effort as well as risks for inconsistencies can be reduced. A positive spin-off effect is that quantitative information on usage frequencies is available in the requirements, enabling planning and prioritizing based on that information.

PAPER V: Are the Perspectives Really Different? – Further Experimentation on Scenario-Based Reading of Requirements

Björn Regnell, Per Runeson, and Thomas Thelin

Technical Report CODEN:LUTEDX(TETS-7172)/1-40/1999 & local 4,
Submitted to the journal of *Empirical Software Engineering*, Kluwer.

Perspective-Based Reading (PBR) is a scenario-based inspection technique where several inspectors read a document from different perspectives (e.g. user, designer, tester). The reading is made according to a special scenario, specific for each perspective. The basic assumption behind PBR is that the perspectives find different defects and a combination of several perspectives detects more defects compared to the same amount of reading with a single perspective. This paper presents a study which analyses the differences in the perspectives. The study is a partial replication of previous studies. It is conducted in an academic environment using MSc and PhD students as subjects. Each perspective applies a specific modelling technique: use case modelling for the user perspective, equivalence partitioning for the tester perspective and structured analysis for the design perspective. A total of 30 MSc students were divided into 3 groups, giving 10 subjects per perspective. A control group of 9 PhD students used a checklist reading technique. The analysis results show that (1) there is no significant difference between the three perspectives in terms of defect detection rate and number of defects found per hour, (2) there is no significant difference in the defect coverage of the three perspectives, and (3) PhD students with a checklist approach find significantly more defects per hour and have a significantly higher detection rate than MSc students with a PBR approach. The results suggest that a combination of multiple perspectives may not give higher coverage of the defects compared to single-perspective reading. It is also indicated that individual abilities and motivation are more important than the reading technique used.

PAPER VI: From Requirements to Design with Use Cases – Experiences from Industrial Pilot Projects

Björn Regnell and Åke Davidson

Proceedings of *3rd International Workshop on Requirements Engineering - Foundation for Software Quality* (REFSQ'97), Barcelona, Spain, June 1997.

In systems evolution, new requirements are distributed on existing architectures. This paper describes a method for modelling how new requirements are distributed on a hierarchy of existing system components. The method applies use case modelling in the transition from requirements to design, with focus on requirements traceability and dynamic system behaviour modelling. The method is based on a recursive process where functionality specification and distribution activities are applied at different abstraction levels in the component hierarchy. The method has been evaluated in three realistic projects, concerned with the evolution of a complex real-time cellular switching system. The subjective conclusions from these evaluations suggest that use case modelling is useful in requirements analysis and distribution within the studied domain.

PAPER VII: A Market-Driven Requirements Engineering Process – Results from an Industrial Process Improvement Programme

Björn Regnell, Per Beremark, Ola Eklundh

Journal of *Requirements Engineering* 3:121-129, Springer-Verlag, 1998.

In market-driven software evolution, the objectives of a requirements engineering process include the envisioning and fostering of new requirements on existing packaged software products in a way that ensures competitiveness in the market place. This paper describes an industrial, market-driven requirements engineering process which incorporates continuous requirements elicitation and prioritisation together with expert cost estimation as a basis for release planning. The company has gained a measurable improvement in delivery precision and product quality of their packaged software. The described process will act as a baseline against which new promising techniques can be evaluated in the continuation of the improvement programme.

4.3 Further Research

One important outcome of research is the formulation of new research issues to be investigated in the striving for more knowledge. Several areas of further research in use case modelling are identified in each of the papers of this thesis. Below, some examples of further research areas are outlined.

- *Use cases and market-driven requirements engineering*. Most of the published requirements engineering practices and process models have been oriented towards customer driven projects. Requirements engineering for market-driven projects poses quite different problems, regarding, for example, how to invent requirements based on foreseen end-user needs and select a set of requirements resulting in a software product which can compete on the market. Requirements prioritisation based on use cases may be a valuable tool, when designing requirements for a packaged software product.
- *Use cases and software procurement*. The market for commercial off-the-shelf software components is taking off, and system providers may construct their systems using components that they buy from specialised software vendors. Software procurers are often in a delicate situation, as they need to be experts in several specific component domains in order to be able to state the right requirements of each component. There are considerable risks of either being too specific, or too general, which may result in a component that does not solve the actual problem. Use cases may play an important role in the requirements models used for specifying component requirements in a procurement situation.
- *Usage-based inspection*. Given that a use case model is part of the requirements document rather than created afterwards, this can be a starting-point for a new type of inspection technique (Regnell, 1998), where the focusing of inspection effort is based on use cases annotated with priority information. Techniques based on pair-wise comparison may be used here (Karlsson, 1997). By comparing pairs of use cases, we may prioritise them according to criteria such as frequency and criticality. With the methods in (Karlsson, 1998) it is possible to derive the relative priority of each use case. Based on this, it may be possible to conduct usage-based reading using the following scheme of *effort partitioning*: (1) Prioritise the use cases;

(2) Decide on the total time to be spent on reading of an artifact; (3) Assign a fraction of the total time to each use case based on the priority; (4) For each use case, inspect the artifact for the assigned time fraction by "walking through" the events of the use case and decide if the artifact is correct. By using the priority criterion of usage frequencies to focus reading by use cases, we get a static verification that corresponds to the expected operational conditions of the system. If we also record the reading time between found defects, we may use these measures to derive an estimate of reliability based on the mean time between defects. The performance of such an inspection technique is an interesting area of further empirical research.

- *Use cases and usage-based testing.* The two proposed strategies of integrating use cases with testing (transformation and extension) require validation in real situations to investigate under which conditions which strategy is best. Further study of stochastic semantics of use case models in the extension approach is needed. Interesting topics are how time can be introduced in stochastic use case models, and how to construct user simulators based on use case models.
- *Use cases and usability engineering*. In usability engineering the user interface of a system is optimized in order provide a user friendly and efficient interaction between humans and computer-based systems. Analysis of user tasks in terms of their contained operations and their frequency is hence an important step in usability engineering. Scenarios have been used in the Human Computer Interaction (HCI) community, and the relation between and integration of HCI methods with use case based RE is an interesting topic for further studies.
- *Requirements engineering process simulation based on "meta use cases"*. In Paper VII, requirements engineering process enactment scenarios are described using Message Sequence Charts in a similar way as use cases are described in Paper II. These meta-level "process use cases" may be a valuable input to the creation of requirements engineering process simulation models, which may be used to investigate process optimization problems, such as how to partition the effort spent on different requirements engineering activities in order to get the highest possible throughput in the analysis and specification of requirements.

- *Formal methods and use case modelling*. In this thesis, the representations of use cases are semi-formal in the sense that they are not based on a mathematical formalism that allow formal reasoning. Although there is in an inherent conflict between the understandability of requirements models and their formalisation, it is still interesting to define and investigate the formal semantics of use cases, using, for example, the Z language (Spivey, 1992) or process algebra (Hoare, 1985), to see how the deductions made possible by such a formal language can help in the analysis of use case based requirements.
- *Use cases as a basis for effort estimation*. When planning a software development project, a difficult task is to estimate the efforts needed to implement the requirements. Given that the requirements include structured use case models, it may be possible to define measurements that can be used in the creation of use case based effort prediction models. Combined with experiences from previous projects, use case metrics may play a key role in effort estimation. Developing and validating such metrics and prediction models is an interesting area of further research.
- *Tool support for use case modelling*. If use cases are represented using natural language without restrictions, it may be suitable to use a common word processor as a tool for creating and maintaining use cases. However, a structured approach to use case modelling may provide the opportunities of effective tool support, where text-based requirements entities are integrated with use case descriptions in, for example, graphical language. Tools may support the analysis of use case syntax and semantics and provide means for relating use cases to other artefacts such as test cases and design models.
- *Empirical studies of use case approaches*. Weidenhaupt et al. (1998) provide an interesting survey of current practice in 15 European software development projects. Further surveys of use case practice and the effectiveness of use cases and scenarios are important. Specific case studies on new use case methodology proposals are also of certain interest. It is very important to gain further understanding of when and how use cases and scenarios can be effectively and efficiently applied in different contexts.

5. References

Adrion, W. R. (1993) “Research Methodology in Software Engineering”, in “Summary of the Dagstuhl Workshop on Future Directions in Software Engineering” Ed. Tichy, Habermann, and Prechelt, *ACM Software Engineering Notes*, SIGSoft, 18(1):36-37.

Anton, A. I., McCracken, W. M., Potts, C. (1994) “Goal Decomposition and Scenario Analysis in Business Process Reengineering”, Proc. *6th Conference on Advanced Information Systems Engineering*, pp. 94-104.

Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., and Zelkowitz, M. 1996. “The Empirical Investigation of Perspective-Based Reading”, *Empirical Software Engineering*, 1(2): 133-164.

Boehm, B. (1988) “A Spiral Model of Software Development and Enhancement”, *IEEE Computer*, May 1988, pp. 61-72.

Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*, 2nd Ed., Benjamin-Cummings Publ.

Brooks, F. P. Jr. (1987) “No Silver Bullet - Essence and Accidents of Software Engineering”, *Computer*, April 1987.

Carroll, J. M., ed. (1995) *Scenario-based Design: Envisioning Work and Technology in System Development*, Wiley, New York, 1995.

Cockburn, A. (1997a) “Goals and Use Cases”, *Journal of Object-Oriented Programming*, 10(5):35-40.

Cockburn, A. (1997b) “Using Goal-Based Use Cases”, *Journal of Object-Oriented Programming*, 10(7):56-62.

Dardenne A., van Lamsweerde A., and Fickas S. (1993) “Goal-Directed Requirements Acquisition”, *Science of Computer Programming*, Vol. 20:3-50.

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

Denney, R. (1998) “Generating Operational Profiles from Workflow Descriptions”, Proc. *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany.

Desharnais, J., Frappier, M., Khédri, R., and Mili, A. (1998) “Integration of Sequential Scenarios”, *IEEE Transactions on Software Engineering*, 24(9):695-708.

Donzelli, P. and Iazeolla, G. (1996) “Performance Modelling of Software Development Processes”, *Proceedings of 8th European Simulation Symposium*, Genoa, Italy.

Doolan, E. P. (1992) “Experiences with Fagan’s Inspection Method” *Software Practice and Experience*, 22(2):173-182.

Du Bois, P., Dubois, E., and Zeippen, J. (1997) “On the Use of a Formal Requirements Engineering Language: The Generalized Railroad Crossing Problem”, *Requirements Engineering Journal*, 2(4):171-183.

Fagan, M. E. (1976) “Design and Code Inspections to Reduce Errors in Program Development” *IBM System Journal*, 15(3):182-211.

Fenton, N., Pfleeger, S., and Glass, R. (1994) “Science and Substance: A Challenge to Software Engineers”, *IEEE Software*, July 1994, pp. 86-95.

Gibbs, W. (1994) “Software’s Chronic Crisis”, *Scientific American*, September 1994.

Glass, R. L. (1994) “The Software Research Crisis”, *IEEE Software*, November 1994, pp. 42-47.

Gough, P., Fodemski, F., Higgins, S., and Ray, S. (1995) “Scenarios - an Industrial Case Study and Hypermedia Enhancements”, Proc. *Second IEEE International Symposium on Requirements Engineering* (RE’95), York, UK, March 1995, IEEE Computer Society Press, pp. 10-17.

Graham, D. R. (1994) “Testing”, *Software Engineering Encyclopedia*, Ed. Marciniak, Vol. 2, pp. 1330–1353, John Wiley & Sons.

Hoare, C. A. R. (1985) *Communication Sequential Processes*, Prentice-Hall.

Hsia, P., Samuel J., Gao J., and Kung, D. (1994) “Formal Approach to Scenario Analysis”, *IEEE Software*, March 1994, pp. 33-41.

Humphrey, W. S. (1989) *Managing the Software Process*. Addison-Wesley.

IEEE (1990) *Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Standard 610.12, New York, USA.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley.

Johnson, P. M., and Tjahjono, D. (1998) “Does Every Inspection Really Need a Meeting?”, *Empirical Software Engineering*, 3(1): 9-35.

Karlsson, J., and Ryan, K. (1997) “A Cost-Value Approach for Prioritizing Requirements”, *IEEE Software*, September/October, pp. 67–74.

Karlsson, J., Wohlin, C., and Regnell, B. (1998) “An Evaluation of Methods for Prioritizing Software Requirements”, *Journal of Information and Software Technology*, 39(14-15).

van Lamsweerde, A., and Willemet, L. (1998) “Inferring Declarative Requirements Specifications from Operational Scenarios”, *IEEE Transaction on Software Engineering*, 24(12):1089-1114.

Leite, J. C. S., Rossi, G., Balaguer, F., Maiorana, V., Kaplan, G., Hadad, G., and Oliveros, A. (1997) “Enhancing a Requirements Baseline with Scenarios”, *Requirements Engineering Journal*, 2(4):184-198.

Loucopoulos, P., and Karakostas, V. (1995) *System Requirements Engineering*, McGraw-Hill Book Company, UK.

Maiden, N., Cisse, M., Perez, H. and Manuel, D. “CREWS Validation Frames: Patterns for Validating Systems Requirements”, Proc. *4th Intl. Workshop on Requirements Engineering - Foundation for Software Quality* (REFSQ'98), Pisa, Italy, pp. 167-180.

Mills, H. D., Dyer, M., and Linger, R. C. (1987) “Cleanroom Software Engineering”, *IEEE Software*, September 1987, pp. 19-24.

MSC, (1993, 1996), Message Sequence Chart. *ITU-T Recommendation Z.120*, International Telecommunication Union. http://www.itu.int

Musa, J. D. (1993) “Operational Profiles in Software Reliability Engineering”, *IEEE Software*, March 1993, pp. 14-32,.

Naur, P., and Randell, B. Eds. (1969) *Software Engineering Conference Report*, Scientific Affairs Division, NATO, Belgium, January 1969.

Porter, A., Votta, L., and Basili, V. R. (1995) “Comparing Detection Methods for Software Requirements Inspection: A Replicated Experiment” *IEEE Transactions on Software Engineering*, 21(6):563-575.

Potts, C. (1993) “Software Engineering Research Revisited”, *IEEE Software*. September 1993, pp. 19-28.

Potts, C., Takahashi, K., and Anton, A. (1994) “Inquiry-Based Requirements Analysis”, *IEEE Software*, March 1994, pp. 21-32.

Robson, C. (1993) *Real World Research*. Blackwell.

Royce, W. (1970) “Managing the Development of Large Software Systems”, *Proc. IEEE Wescon*, New York, August 1970.

Regnell, B., and Runeson, P. (1998) “Combining Scenario-based Requirements with Static Verification and Dynamic Testing”, Proceedings of *4th Intl Workshop on Requirements Engineering - Foundation for Software Quality* (REFSQ'98), Pisa, Italy, pp. 195-206.

Rubin, K., and Goldberg, a. (1992) “Object Behavior Analysis”, *Communications of the ACM*, 35(9):48-62.

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

Runeson, P., and Wohlin, C. (1992) “Usage Modelling: The Basis for Statistical Quality Control”, *Proc. 10th Annual Software Reliability Symposium*, pp. 77-84, Denver, Colorado, USA.

Runeson, P., and Wohlin, C. (1995) “Statistical Usage Testing for Software Reliability Control”, *Informatica*, Vol. 19, No. 2.

Ryan, K. (1995) “Let’s Have More Experimentation in Requirements Engineering”, Proc. *Second IEEE International Symposium on Requirements Engineering* (RE’95), York, UK, March 1995, IEEE Computer Society Press, p. 66.

Selic B., Gullekson G., and Ward P. T. (1994) *Real-Time Object-Oriented Modelling*, Wiley & Sons.

Sommerville, I. (1996) *Software Engineering*, 5th edn., Addison-Wesley.

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

Sutcliffe, A., Maiden, N., Minocha, S., and Manuel, D. (1998) “Supporting Scenario-Based Requirements Engineering”, *IEEE Transaction on Software Engineering*, 24(12):1072-1088.

Sutcliffe, A., and Minocha, S. (1998) “Scenario-based Analysis of Non-Functional Requirements”, Proceedings of *4th Intl Workshop on Requirements Engineering - Foundation for Software Quality* (REFSQ'98), Pisa, Italy, pp. 219-233.

Sutcliffe, A. (1998) “Scenario-Based Requirements Analysis”, *Requirements Engineering Journal*, 3(1):48-65.

Votta, L. G. 1993. “Does Every Inspection Need a Meeting?” *Proceedings of the ACM SIGSOFT 1993 Symposium on Foundations of Software Engineering, ACM Software Engineering Notes*, 18(5):107-114.Webster’s Ninth New Collegiate Dictionary (1989), Springfield, Massachusets, USA, G. & C. Merriam.

Whittaker, J. A., and Thomason, M. G. (1994) “A Markov Chain Model for Statistical Software Testing”, *IEEE Transactions on Software Engineering*, 20(10):812–824.

Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. (1998) “Scenarios in System Development: Current Practice”, *IEEE Software*, March/April 1998, pp. 34-45.

Wohlin, C., Regnell, B., Wesslén, A., and Cosmo, H. (1994) “User-Centred Software Engineering – A Comprehensive View of Software Development”, Proceedings of *Nordic Seminar on Dependable Computing Systems* (NSDCS’94), Lyngby, Denmark.

Zelkowitz, M. V. and Wallace, D. R. (1998) “Experimental Models for Validating Technology”, *IEEE Computer*, May 1998, pp. 23-31.

I

PAPER I:

Improving the Use Case Driven Approach to Requirements Engineering

Björn Regnell, Kristofer Kimbler and Anders Wesslén

Published in Proceedings of RE'95, *Second IEEE International Symposium on Requirements Engineering*, York, UK, March 1995

Abstract

This paper presents the idea of Usage-Oriented Requirements Engineering, an extension of Use Case Driven Analysis. The main objective is to achieve a requirements engineering process resulting in a model which captures both functional requirements and system usage aspects in a comprehensive manner. The paper presents the basic concepts and the process of Usage-Oriented Requirements Engineering, and the Synthesized Usage Model resulting from this process. The role of this model in system development, and its potential applications are also discussed.

1. Introduction

When dealing with complex systems, it does not seem feasible to go directly, in one step, from an informal requirements description provided by the customer to a formal requirements specification. Too rapid formalization of requirements may have several negative consequences, such as a substantial semantic gap between the requirements description and the requirements specification or incompleteness of the latter. It is also very difficult to produce a formal specification without a deep understanding of what the customer and the end users expect from the system, and how they intend to employ it in practice. This kind of information is rarely provided at the outset of system development.

This paper presents Usage-Oriented Requirements Engineering (UORE) which tries to address the above issues in a structured and systematic way. The concept of UORE originates from Use Case Driven Analysis (UCDA), a key contribution of Object-Oriented Software Engineering (OOSE) [1].

Our objective is to improve the original UCDA by extending it with a *synthesis phase* where separate use cases are integrated into a Synthesized Usage Model (SUM). This model captures both functional requirements and system usage aspects. To facilitate this integration, UORE introduces a formal, graphical representation of use cases and abstraction mechanisms for representing user and system actions.

Abbreviations

The following abbreviations are used throughout this paper:

AIO	Abstract Interface Object
AUS	Abstract Usage Scenario
SUM	Synthesized Usage Model
UCDA	Use Case Driven Analysis
UCM	Use Case Model
UCS	Use Case Specification
UORE	Usage-Oriented Requirements Engineering
OOSE	Object-Oriented Software Engineering, according to [1]

2. Use Case Driven Analysis

This section presents and discusses UCDA as defined in OOSE [1]. The basic concepts of UCDA are actors and use cases. An *actor* is a specific role played by a system user, and represents a category of users that demonstrate similar behaviour when using the system. By users we mean both human beings, and other external systems or devices communicating with the system. An actor is regarded as a class, and users as instances of this class. One user may appear as several instances of different actors depending on the context.

A *use case* is a system usage scenario characteristic of a specific actor. During the analysis we try to identify and describe a number of typical use cases for every actor. Use cases are expressed in natural language with terms from the problem domain. The descriptions of actors and use cases form the Use Case Model (UCM).

Advantages. UCDA helps to cope with the complexity of the requirements analysis process. By identifying and then independently analysing different use cases we may focus on one, narrow aspect of the system usage at a time.

Since the idea of UCDA is simple, and the use case descriptions are based on natural concepts that can be found in the problem domain, the customers and the end users can actively participate in requirements analysis. In consequence, the developers can learn about the potential users, their actual needs, and their typical behaviour.

Disadvantages. The lack of synthesis is probably the main drawback of UCDA. The Use Case Model that we get from UCDA is just a loose collection of use cases. In the subsequent phases of OOSE, these use cases are directly used to create the so-called Analysis Model. This model describes the structure of the system and is a step towards design. What we really would like to get from requirements analysis, is a model which captures the functional requirements and system usage, without any design aspects.

Although use cases are perfect material for creating test cases, the UCM resulting from UCDA cannot be used for automatic generation of test cases. This limits its applicability as a reference model for validation and verification.

There are also several problems with the interpretation of the actor and use case concepts, as defined in [1]. No clear definition of the semantics of use cases, and no consistent guidelines on how the use cases should be described are provided. It is not clear what kind of events we should concentrate on while describing use cases; external stimuli-responses only, or internal system activities as well. In [1], use cases are treated as classes with inheritance-like relations, but, at the same time, they are seen as sequences of events. Object-orientation purists tend to treat everything as objects, but here we find the class interpretation rather artificial and confusing.

In [1], every use case is associated with a specific actor, but, at the same time, allows use case descriptions in which several actors are involved. Moreover, an actor is defined as a specific role played by a user. This means that, in extreme, one physical user can appear as different actors in a single use case. These uncertainties leave too much room for free interpretation of the actor and use case concepts, and may cause a lot of confusion.

The number of use cases may be very large in cases of complex systems. Since produced independently, there might be inconsistencies between use case descriptions. Moreover, use cases might be contradictory, as they express goals of different actors. In [1] there is no support for resolving such problems.

A specific use case can not occur in every situation. What we need for each use case is a specification of the context in which it can be triggered and successfully accomplished. This issue is not addressed by UCDA.

In general, UCDA, as defined in [1], does not fully address the following issues:

- Use cases are not independent. They may overlap, occur simultaneously, or influence each other.
- Use cases occur under specific conditions. They have invocation and termination contexts.
- The level of abstraction of use cases and their length are matters of arbitrary choice.
- The use cases can, in practice, guarantee only partial coverage of all possible system usage scenarios.

3. Usage-Oriented Requirements Engineering

The proposed UORE process aims at removing some of the weaknesses of UCDA stated in the previous section. UCDA is extended with a synthesis phase, where use cases are formalized and integrated into a Synthesized Usage Model. The SUM captures functional requirements and system usage in a more formal way than the UCM.

The SUM is intended to be a part of requirements specification, and a reference model for validation and verification. The SUM captures the following related aspects:

- Categories of system users and their objectives,
- Domain objects, their attributes, and operations,
- Stimuli and responses of user-system communication,
- User and system actions, their possible combinations and usage contexts,
- Scenarios of system usage, their flows of events, and trigger conditions.

The process of UORE consists of two phases, *analysis* and *synthesis*, as shown in fig. 1. The analysis phase has an informal requirements description as input, and produces the use case model containing descriptions of actors and use cases. This model, in turn, is used as input to the synthesis phase which formalizes the use cases, integrates them, and creates the synthesized usage model

3.1 Analysis phase

The analysis phase of UORE resembles the original UCDA, and consists of two interrelated activities:

1. Identification of use cases and actors,
2. Unification of terminology.

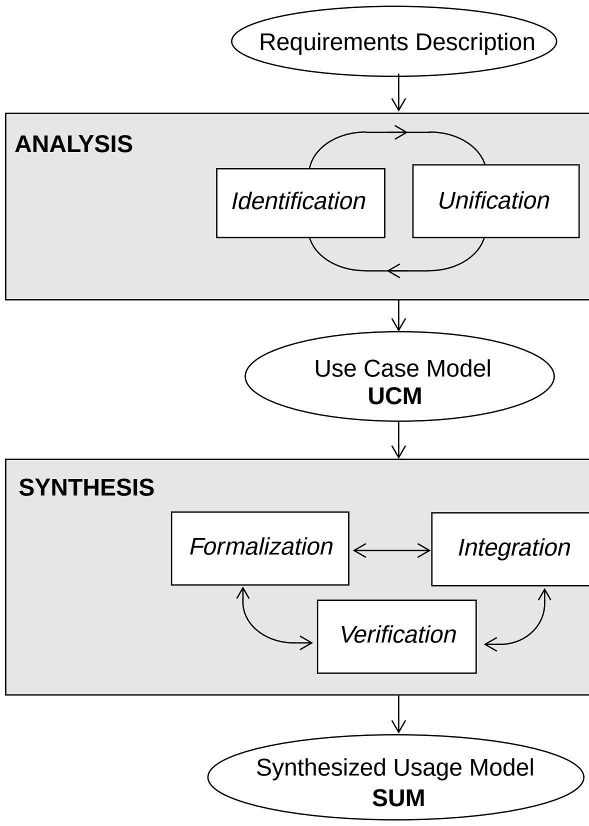


Figure 1. *The process of UORE.*

The first activity aims at finding and describing actors and use cases. The second activity unifies the terminology of these descriptions. For this purpose the problem domain objects and their attributes are identified and described in a *data dictionary*. The focus is on entities manipulated by the actors, externally observable system operations, and elements of the user interface.

The unification of terminology is important, especially as different use cases may be described by separate persons or groups. The terminology is gradually extended and revised as more and more use cases are identified. The unified terminology is enforced by inspections. The two activities of the analysis phase are performed iteratively.

To illustrate UORE we will use a well-known example of an Automated Teller Machine (ATM) [2]. ATM offers basically two services: *cash withdrawal* and *account control*. In fig. 2 we show examples of actors and use cases.

Actors:
ATM customer – uses the ATM to withdraw cash or control the account balance.
ATM supervisor – supervises and maintains the operation of the ATM.
ATM database – the external system maintaining account information.

1. ***Withdraw Cash, normal case***
Actor: “*ATM customer*”

1.IC	**Invocation Conditions:**
1.IC.1	The system is ready for *transactions*.
1.FC	**Flow Conditions:**
1.FC.1	The user’s **card** *is valid*.
1.FC.2	The user enters a *valid* **code**.
1.FC.3	The user enters a *valid* **amount**.
1.FC.4	The machine has the *required amount of* **cash**.
1.FE	**Flow of Events:**
1.FE.1	The user inserts the **card**.
1.FE.2	The system *checks if the* **card** *is valid*.
1.FE.3	A *prompt for the* **code** is given.
1.FE.4	The user enters the **code**.
1.FE.5	The system *checks if the* **code** *is valid*.
1.FE.6	A *prompt “enter amount or select balance”* is given.
1.FE.7	The user enters the *amount*.
1.FE.8	The system *checks if the amount is valid*.
1.FE.9	The system *collects the* **cash**.
1.FE.10	The **cash** *is ejected*.
1.FE.11	A *prompt “take cash”* is given.
1.FE.12	The user takes the **cash**.
1.FE.13	The **card** *is ejected*.
1.FE.14	A *prompt “take card”* is given.
1.FE.15	The user takes the **card**.
1.FE.16	The system *collects* **receipt** *information*.
1.FE.17	The **receipt** *is printed*.
1.FE.18	A *prompt “take receipt”* is given.
1.FE.19	The user takes the **receipt**.
1.TC	**Termination condition:**
1.TC.1	The system is ready for *transactions*.

2. ***Withdraw Cash, amount invalid***
Actor: “*ATM customer*”

2.IC	**Invocation Conditions:**
2.IC.1	Same as 1.IC.1.
2.FC	**Flow Conditions:**
2.FC.1	Same as 1.FC.1 - 1.FC.2.
2.FC.2	The user enters an *invalid* **amount**.
2.FE	**Flow of Events:**
2.FE.1	Same as 1.FE.1 - 1.FE.8
2.FE.2	The *“invalid amount” message* is given.
2.FE.3	A *prompt for “retry”* is given.
2.FE.4	The user *aborts the transaction*.
2.FE.5	Same as 1.FE.13 - 1.FE.15
2.TC	**Termination condition:**
2.TC.1	Same as 1.TC.1.

3. ***Account Control, normal case***
Actor: “*ATM customer*”

3.IC	**Invocation Conditions:**
3.IC.1	Same as 1.IC.1.
3.FC	**Flow Conditions:**
3.FC.1	Same as 1.FC.1 - 1.FC.2.
3.FE	**Flow of Events:**
3.FE.1	Same as 1.FE.1 - 1.FE.6.
3.FE.2	The user *selects “balance”*.
3.FE.3	The system *collects balance information*.
3.FE.4	The *balance is displayed*.
3.FE.5	Same as 1.FE.13 - 1.FE.19
3.TC	**Termination condition:**
3.TC.1	Same as 1.TC.1.

Problem domain objects in **bold** face.
Defined and unified terminology in *italics*.

Figure 2. *Use case description examples.*

Differences

As mentioned above, the analysis phase of UORE resembles the OOSE version of UCDA. There are, however, a number of issues that make our approach different:

- Changed semantics of actors and use cases,
- Identification of use case contexts,
- Strict application of the single-actor view,
- Explicit unification of terminology,
- Structured description of use cases.

In UORE, an actor represents a user (a person or an external system) that belongs to a set of users with common behaviour and goals. An UORE actor does not necessarily model a *single* role played by a user, as in [1]. In our opinion, the single-role semantics of actors may lead to use cases which address too narrow aspects of system usage. This, in turn, disables analysis of how different system operations interact. (Some systems may allow a user to play multiple roles at the same time.)

Unlike [1], which treats the use cases as classes, we regard them just as examples of system usage. We consider use cases as “experimental material” which will be further investigated in the synthesis phase.

In UORE, each use case describes the system behaviour, as seen by one actor only. This *single-actor-view* approach makes the use case concept simpler. We assume that the actor involved in a use case communicates with other actors through the system. No situations with direct actor-to-actor communication are modelled. In other words, the narration of the use cases distinguishes only between the actor and the “rest”. If a system usage scenario involves several actors, this scenario should be modelled by several use cases, one for each involved actor. This provides a clear criterion for constructing use case descriptions and reduces their complexity. To conclude, we can say that the UORE principle for use case definition is: “*multiple roles, yes; multiple actors, no*”.

In UORE, the description of each use case contains a list of conditions defining a context in which the specific flow of events of the use case can occur. The *invocation conditions* and *termination conditions* define the system state before and after the use case, while the *flow conditions* state the assumptions about the user and system behaviour during the use case. A flow condition is not necessarily true at the invocation of the use case, but it becomes true at some point in the use case. A flow condition is thus a temporal assertion that implicitly refers to a “future” point in the flow of events of the use case. These different conditions are an important aid in the synthesis phase for finding relations between use cases.

In order to avoid some typical problems with natural language descriptions, all the use cases should use the same terminology and format. The terminology of these descriptions is unified across different use cases, as discussed above. The examples in fig. 2 show a possible structure of use case descriptions. A systematic numbering of events supports traceability within and between the models of the analysis and synthesis phases. Furthermore, when describing a use case, we can use such numbers to refer to identical conditions and sequences of events in other use cases, in order to make the description shorter.

3.2 Synthesis phase

The synthesis phase formalizes the use cases, integrates them, and creates the Synthesized Usage Model. The synthesis phase consists of three activities:

1. Formalization of use cases,
2. Integration of use cases,
3. Verification.

These three activities are carried out in an iterative manner, until an agreement upon the correctness and completeness of the SUM is reached. In the following sections we will describe each activity and the concepts they use.

3.3 Formalization activity

The formalization activity aims at producing a formal Use Case Specification (UCS) for each use case identified in the analysis phase. The product of this activity is a collection of UCS’s, represented in the formal, graphic language of message sequence charts (an extension of [3]). Each UCS expresses the temporal ordering of user *stimuli*, system *responses*, and *atomic operations*.

The formalization activity has the following steps:

1. Identification of abstract interface objects,
2. Identification of atomic operations,
3. Creation of one UCS for every use case.

The concepts used in the formal representation of user-system communication, and the steps necessary in the creation of a UCS are explained below.

Abstract interface objects

The user never communicates directly with a software system. Some sort of interface is always involved in this communication. The interface transforms the user's *stimuli* into *messages* (software events) and, messages from the system into *responses* comprehensible to the user. This transformation is not necessarily straightforward. The three basic elements of user-system communication; the *user*, the *interface* and the *system*, are inherently parts of system usage, consequently they can be found in use case descriptions.

The entities that form the nature of user-system communication will be called Abstract Interface Objects (AIO). They are abstract in the sense that they do not necessarily represent concrete interface objects. Instead, they model responsibilities (see [4]) that can be mapped to one or more real interface objects. The intention is to avoid any design decisions at this stage.

Identification of abstract interface objects is achieved by examining all the use cases and the problem domain terminology, and searching for entities that take part in the actor-system communication. An AIO is characterized by its sets of stimuli, responses, messages, and states.

Atomic operations

On a conceptual level we can describe the elements of the system's capabilities by atomic operations. They are operations performed by the system, and have effect on the users. A system operation is atomic from an actor's point of view, if it does not require any communication with this actor during its execution. However, other actors may see the same operation as a combination of other atomic operations and communication protocols. For example, the operation *card validation* is atomic from the *ATM Customer* actor's point of view, although from the *ATM Database* actor's point of view it is a sequence of operations and communications.

The atomic operations are identified from the use cases by focusing on system operations that do not require interaction with the actor involved in the use case. Every system action is described and given a unique name to be used uniformly in all use case specifications. We will not elaborate here on the specification of atomic operations.

Formal use case specification

The formalization activity produces formal use case specifications. After identifying all abstract interface objects and atomic operations, we transform the flow of events of every use case into a UCS that models the temporal relations between AIO stimuli/responses/states and atomic operations.

We illustrate the notation of UCS by our ATM example. The UCS corresponding to the use case "withdraw cash, normal case" is shown in fig. 3. The left-most time axis of fig. 3 represents the specified actor. The right-most time axis represents the system. Between the actor and the system we have the different AIO's involved in this use case. The AIO states are drawn as diamonds on the AIO time axis, and the atomic operations are drawn as boxes on the system's time axis.

3.4 Integration activity

The integration activity aims at merging different use case specifications and producing a Synthesized Usage Model. The SUM consists of a collection of *usage views*, one for each actor. The integration activity consists of the following three steps:

1. Identification of user and system actions,
2. Creation of abstract usage scenarios,
3. Integration of abstract usage scenarios.

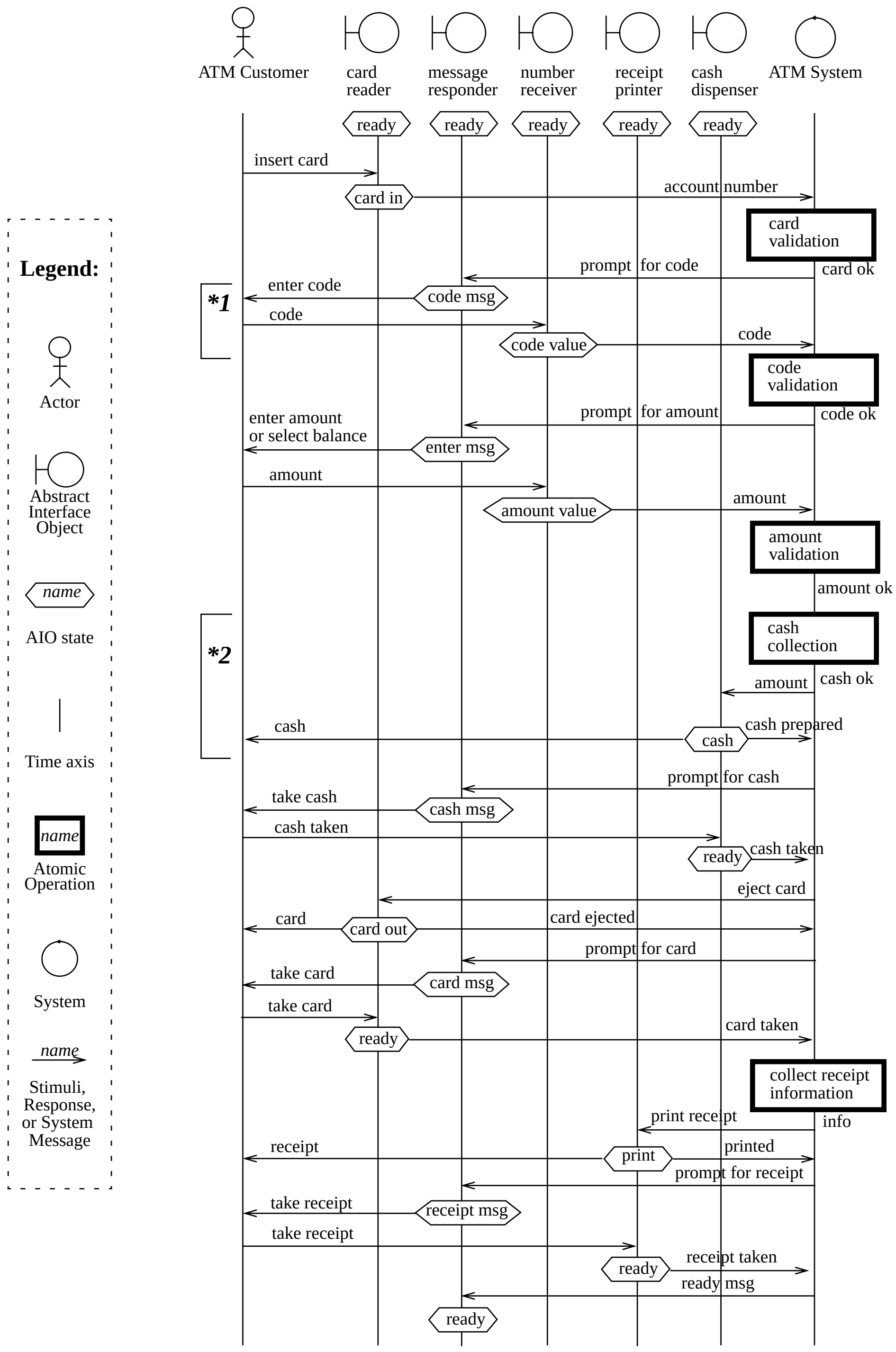
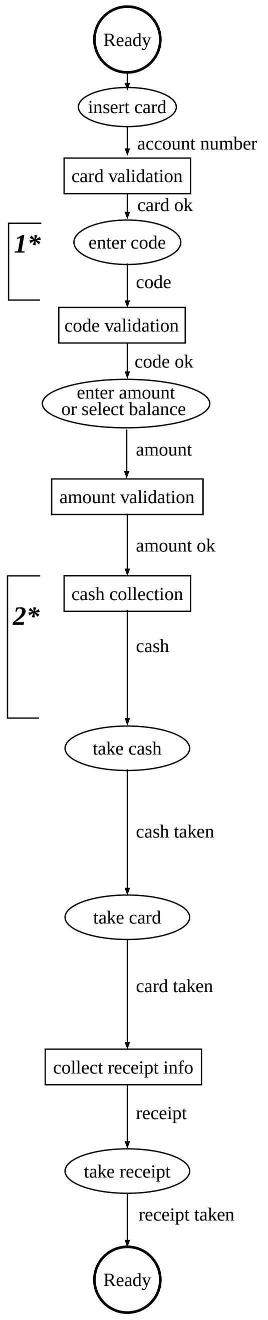


Figure 3. *Use Case Specification* ***Withdraw cash - normal case***.



Note: For the example in 3.4, two UCS parts are marked **1* and **2*, and correspond to the actions *1** and *2**.

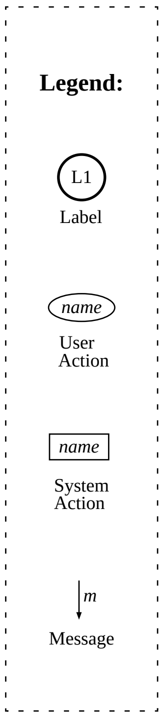


Figure 4. *Abstract Usage Scenario.*

User and system actions

In a use case, the control shifts between the user and the system. When we formally represent system usage we would like to have an abstraction mechanism that conceals the detailed protocol of the interaction during the user-controlled parts and the system-controlled parts of a use case. We use the terms *user actions* for protocols where the user is in control, and *system actions* for protocols where the system is in control.

The first step of the integration activity aims at extracting such abstract protocols. Hence, in several UCS's we can identify actions such as “enter code” and “cash collection”, which form a demarcated protocol with a sequence of related events, resulting in a single message. All such UCS parts are uniformly defined with a name and description.

To illustrate this, in fig. 3 a UCS part is marked with ***1**, which corresponds in fig. 4 to the user action **1*** “enter code” and the resulting message “code”. Similarly, the UCS part denoted ***2** corresponds to the system action “cash collection” and message “cash”, marked with **2***.

An action can have different outcomes. For example, the system action “code validation” may result in the events “code OK” or “code invalid”. An action can thus represent a collection of similar protocols, with different outcomes. An action can also be seen as a state where the user or the system tries to accomplish some specific task. The user and system actions could be described internally by finite state machines, as proposed in [5]. This possibility is, however, not yet incorporated into UORE.

Abstract usage scenarios

Using the abstraction mechanisms of user and system actions, use case specifications can be expressed in a more condensed way. Every UCS is transformed into an Abstract Usage Scenario (AUS), drawn as a sequence of user actions (bubbles) and system actions (boxes) interconnected with transitions (arrows) that represent the resulting messages of each action. The invocation and termination context of an AUS is indicated by labels (circles). A label denotes an external system state, i.e. a subset of the carthesian product of all AIO states. In fig. 4 a sample AUS is shown.

The main purpose of creating AUS's, is to make the synthesis feasible even if we have a very large number of use cases. By raising the abstraction level we hide information, to make “clean threads” and then “weave them together” in the last step of integration.

Synthesized Usage Model

The SUM consists of one usage view per actor. A usage view is synthesized from all Abstract Usage Scenarios produced for one specific actor. A usage view is created by finding similar parts of Abstract Usage Scenarios and merging them. The result is a directed graphs with three types of nodes: *user actions*, *system actions*, and *labels*. These nodes have the same meaning as in Abstract Usage Scenarios. Labels are used to maintain traceability between usage views and AUS's. Additional labels can be introduced to divide large graphs into separate sub-diagrams, thus promoting scalability. An example of a usage view is given in fig. 3. (This usage view is an integration of more use cases than shown in the examples in fig. 3.)

In summary, the SUM contains descriptions of the following:

1. Actors,
2. Usage views,
3. Use case specifications,
4. Abstract interface objects,
5. User actions and system actions,
6. Data dictionary with problem domain objects.

3.5 Verification activity

The purpose of the verification activity is to obtain a consistent and complete SUM. There are two verification steps related to the formalization activity and integration activity respectively:

1. Verification of UCS,
2. Verification of SUM.

The verification of a UCS is performed as a rigorous inspection where the UCS is compared with the corresponding use case in the UCM. The reviewers check that the UCS is a correct transformation of the informal use case description, meaning that everything in the use case is contained in the UCS and that the objects in all UCS's are consistently defined.

The second verification step aims at ensuring that the SUM completely covers every UCS. Here is a great potential for automatic verifica-

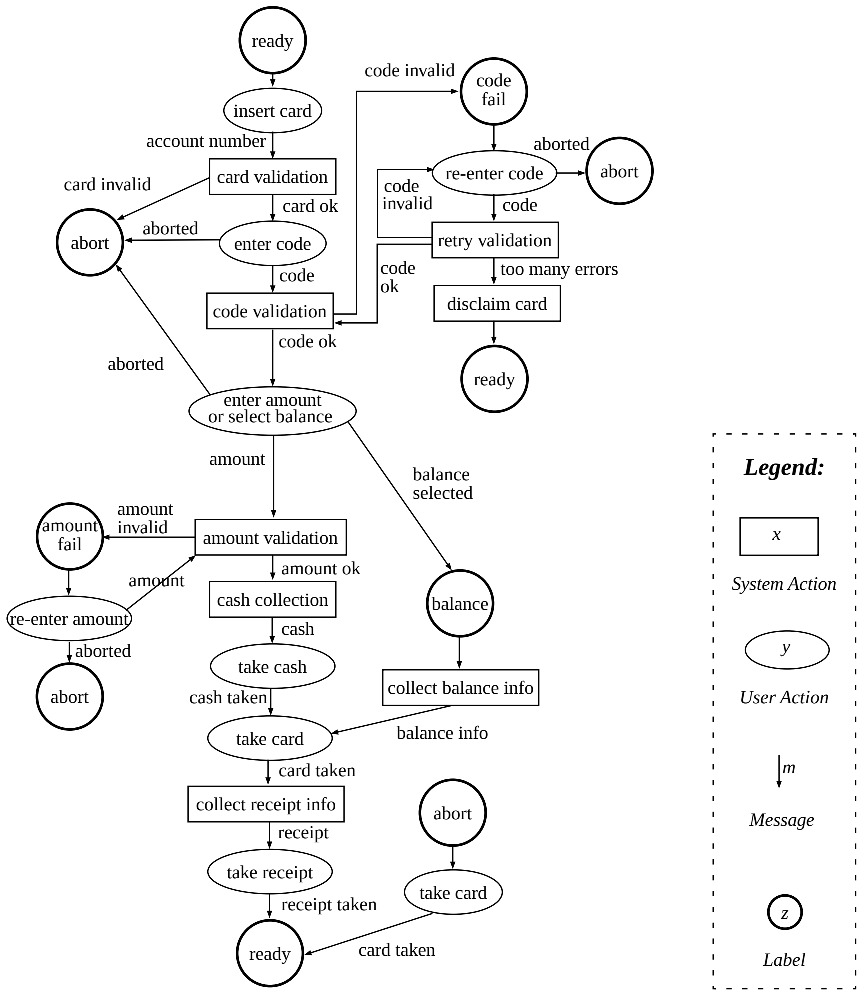


Figure 5. *The usage view for "ATM Customer".*

tion, where a tool could check that every AUS is a possible path in the corresponding usage view. It is possible that, during the synthesis phase, new user and system actions are discovered and incorporated in a usage view, thus making more usage scenarios valid in addition to the defined AUS's. In the verification of the SUM, such additional usage scenarios can be created by traversing the graphs of the usage views. In this way, the "experimental material" of use cases is used to build a model that enables the discovery of yet unidentified scenarios, and thus making the SUM a model that covers *more* than the initial experiments.

4. Applications of SUM

The Synthesized Usage Model is designed to be used as a *reference model* for the remaining phases of system development. The SUM captures not only functional requirements, but also system usage. The SUM is a source of information about *what* the system is supposed to do, and *how* it should behave from the user's point of view in different usage contexts. Therefore, the SUM can form a backbone for the whole development process including system design, verification, and validation. This role of a formal usage model in system development is discussed in [6]. In the following sections we discuss the potential benefits of SUM in system design as well as in verification and validation. A report on practical experiences in the field of telecommunication is also given.

4.1 System design

The SUM captures functional and behavioural aspects of the system that are important for system design. The user and system actions are abstractions of user-system communication protocols that produce system stimuli and responses.

The semantics of an action is defined by the different contexts in which it can occur, and the set of abstract interface objects it encapsulates. This information can be directly applied in the *external design*, where the mapping of AIO's to actual interface objects, and the concrete shape of the user interface is determined. The SUM can also be used for creating a prototype of the user interface.

The set of atomic system operations and their usage contexts is a valuable information for *internal design*. Here we have to consider the fact that some system operations can be atomic for one actor, but not for another. This information can be useful for finding a robust object structure of the system, and for allocating functionality to objects, as suggested in [1].

4.2 System verification and validation

In order to ensure the correctness of the system implementation and requirements traceability, the system can be verified against the SUM by means of testing. The possibility of *automatic generation of test cases* is one of the most important properties of the SUM. Each usage view of the SUM can be used to generate test cases in the form of “re-created” usage scenarios. These scenarios contain both stimuli to the system and the expected system responses, thus enabling automatic verification of the test results. Though a strategy of test case selection is beyond the scope of this paper, in the next paragraph we briefly discuss the possibility of using the SUM for statistical usage testing.

Statistical usage testing

In statistical usage testing [7], test cases are derived from a usage model. This model describes both functional and statistical properties of system usage. Experiences with the so-called *state hierarchy model* [8, 9], shows that it is feasible to generate test cases automatically from a model of system usage. These test cases are samples of the expected system usage and have the necessary statistical properties that enable certification of the system’s reliability.

Statistical usage testing is a black-box testing technique, as it derives the usage model from the requirements specification. Unfortunately, there is a substantial gap between the usage model required for statistical usage testing and the traditional requirements specification. What we need from requirements analysis is an explicit description of system usage. By using the SUM as an element of requirements specification, we can possibly bridge this gap and make test preparation easier. By this approach, test preparation can concentrate on modelling statistical properties by adding probabilities to the SUM.

4.3 Practical experiences

Though UORE, as described in this paper, has been used only in minor case studies, the key elements of the method (synthesis of use cases, single-actor views, and SUM) have already been applied in practice. In the analysis of interactions between pan-European telecom services, these elements of UORE have yielded positive results [10, 11].

The problem of undesired feature interactions is a major threat to the rapid deployment of new telecom services. An interaction occurs when one service feature changes or disables the operation of another feature. One of the approaches to this problem is to detect and resolve interactions during requirements analysis.

By applying the use case driven approach to the analysis of the pan-European candidate services, and synthesizing the use cases, a behavioural model of these services and their features was obtained. This model corresponds to one usage view of the SUM, i.e. it represents the behavioural aspects of the services as seen by one actor, *service user*. This model was analysed by a custom-designed tool that automatically detected a large number of potential feature interactions. The tool used the possibility of re-creating service usage scenarios (in this case different scenarios of telephone calls) from the model, as described in [12] and [10].

5. Conclusions

The ideas introduced in this paper clarify and formalize several important aspects of UCDA. It is our belief that UORE is a significant improvement of UCDA, by its criteria for finding, describing, formalizing, and synthesizing use cases. However, it is still untried on the large scale, and it remains to be proven that UORE is easier to use and gives a better support in requirements engineering than the original UCDA. By empirical studies we hope to prove the benefits of the SUM as a system reference model.

In summary, the main contributions are:

- The improvement of the actor and use case concepts,
- The formalization of use case descriptions,
- The idea of use case synthesis,
- The Synthesized Usage Model,
- The process of Usage-Oriented Requirements Engineering.

There are still a number of issues to be addressed in future research, for example:

- Formal description of the invocation, termination, and flow conditions of use cases,
- Formal description of procedural and non-procedural properties of user and system actions,
- Further refinement of use case synthesis – integration of different usage views in the SUM,
- Formal definition of the syntax and semantics of SUM,
- Transformation of the SUM into a test model suitable for statistical usage testing,
- Automation of verification and validation of the test results by using SUM as a reference model of system behaviour.

Acknowledgements

We would like to thank the following persons for reviewing this paper: Claes Wohlin, Daniel Söbirk, Henrik Cosmo, Helen Sheppard, and Anna-Sofia Maurin.

References

[1] Jacobson, I., et al. *Object-Oriented Software Engineering, A Use Case Driven Approach*, Addison-Wesley, 1992.

[2] Sommerville I., *Software Engineering*, fourth edition, Addison-Wesley, 1992.

[3] ITU-T Recommendation Z.120. *Message Sequence Chart (MSC)*, Telecommunication Standardization Sector of International Telecommunication Union, 1993.

[4] Wirfs-Brock, R., et al. *Designing Object-Oriented Software*, Prentice Hall, 1990.

[5] Zave P. “Feature Interaction and Formal Specification in Telecommunications”, *IEEE Computer*, August 1993.

[6] Wohlin C., Regnell B., Wesslén A. and Cosmo H. “User-Centred Software Engineering - A Comprehensive View of Software Development”, *Proceedings of Nordic Seminar on Dependable Computing Systems*, Denmark, August 1994.

[7] Mills, H. D., Dyer, M. and Linger, R. C., “Cleanroom Software Engineering”, *IEEE Software*, pp. 19-24, September 1987.

[8] Runeson, P. and Wohlin, C., “Usage Modelling: The Basis for Statistical Quality Control”, *Proceedings of 10th Annual Software Reliability Symposium*, pp. 77-84, Denver, Colorado, USA, 1992.

[9] Wohlin, C. and Runeson, P. “Certification of Software Components”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp 494-499, 1994.

[10]EURESCOM Project EU-P230, *Enabling pan-European services by cooperation of PNO’s IN platforms*, Deliverable 4, EURESCOM, Heidelberg, December 1994.

[11]Kimbler K., Kuisch E. and Muller J., “Feature Interactions among Pan-European Services”, *Feature Interactions in Telecommunications Systems*, IOS Press, Netherlands, 1994.

[12]Kimbler K. and Söbirk D., “Use Case Driven Analysis of Feature Interactions”, *Feature Interactions in Telecommunications Systems*, IOS Press, Netherlands, 1994.

PAPER II:

A Hierarchical Use Case Model with Graphical Representation

II

Björn Regnell, Michael Andersson and Johan Bergstrand

Published in Proceedings of ECBS'96, *IEEE International Symposium and Workshop on Engineering of Computer-Based Systems*, March 1996

Abstract

Use case modelling is gaining increasing interest in computer-based systems engineering, especially in the earliest stages of system development, where requirements are elicited, documented and validated. This paper presents a conceptual framework for use case modelling and a new use case model with graphical representation, including support for different abstraction levels and mechanisms for managing large use case models. Current application of use cases in requirements engineering is discussed, as well as ontological and methodological issues related to use case modelling.

1. Introduction

The elicitation, analysis and documentation of requirements on complex computer-based systems is a crucial and non-trivial task [1]. Well defined concepts and methods are needed when constructing formal, agreed upon, specifications that represent requirements in an unambiguous, consistent, and complete manner. It is also important to have representations of requirements models that are easily understood by the different stakeholders that take part in requirements analysis [2].

Use cases (also called scenarios), introduced in OOSE [3], have been given increasing attention in other object-oriented development methods, e.g. [4, 5]. In requirements engineering use cases are of special interest, as this concept has proven to be valuable in elicitation, analysis and documentation of requirements [6, 7, 8]. Use cases also provide traceability of requirements throughout the design, implementation and verification and validation phases [9]. This paper presents ongoing research in use case based requirements engineering, with focus on representation. The presented results are extensions of the work described in [10, 11].

A major objective of use case driven analysis is to model a system's functional requirements by describing different scenarios of system usage. The basic concepts of these descriptions are actors and use cases. An *actor* is a specific role played by a system user, and represents a category of users that demonstrate similar behaviour when using the system. The way an actor uses the system is described by *use cases*. The actors and use cases form the use case model. An important advantage of use case driven analysis is that it helps to manage complexity, as it focuses on one specific aspect of usage at a time. It also provides means for customers and users to actively participate in requirements analysis, as use cases are expressed in terms familiar to them.

An important question in use case based requirements engineering research is: How should use cases be represented? In [6], one of the conclusions from an industrial case study on use cases in requirements specification was that textual representation is insufficient and that graphical representation is desirable. This paper presents a new hierarchical use case model with a graphical representation based on an extension of Message Sequence Charts (MSC), an ITU-T standardized language [12].

The main hypothesis behind this work is that use case modelling benefits from graphical representations that have support for descriptions at different levels of abstraction. The results presented here include such a representation.

Section 2 discusses the state of the art in uses case modelling. Section 3 provides a conceptual framework for a use case ontology and discusses methodological issues. The presented conceptual framework forms the basis for Section 4, where a new hierarchical use case model with graphical notation is presented. Section 5 provides conclusions and covers some issues of further research and tool support.

2. Use Cases in Current Methods

This section discusses how use cases are currently applied in four development methods; OOSE [3, 13], OMT [14, 5], Booch [4], and ROOM [15]. The main focus here is on the role of use cases in requirements elicitation and analysis.

Table 1 shows an overview of these four methods with respect to concepts, notation, process and methodology. The overview is not complete, but highlights some similarities and differences between the methods.

	OOSE	**OMT**	**Booch**	**ROOM**
Concepts	use case, actor, exception, extends, uses	use case, actor, scenario, pre- & post condition, exception, adds	use case, scenario, initiator, pre- & post condition	scenario, initiator, participant, package, alternative, exception, high level scenario
Notation in requirements analysis phase	Natural language describes use cases. Arrows and ovals describe relations between use cases.	Natural language with some structure guidelines.	Natural language.	Natural language and Message Sequence Charts.
Role in the development process	Drives the whole process. Used for finding objects. Used for finding robust design.	Used to strengthen req. analysis. Used for finding objects.	Used for finding objects. Used in design. Used for release planning.	Used for expressing functional requirements; to make initial design; for incremental development.
Methodology for creating use case model	Some heuristics are provided. Questions to answer for each actor help identify use cases.	Step-wise action list is provided. Scenarios are combined/ generalized into use cases.	Prescribes scenario planning as an activity and provides a few guidelines.	A few guidelines are provided.

Table 1. *Overview of use cases in current object-oriented methods.*

Below we give a few reflections on these methods.

- OOSE has the most thorough definition of the use case model and the most methodological support for constructing the model. In OOSE, use cases play a central role, as the whole development process is driven by the use case model.
- OMT [5] has adopted many of the use case related concepts in OOSE and added pre- and post conditions to use cases.

- The Booch method includes scenarios, similar to use cases, but the initiator concept does not fully correspond to the actor concept of OOSE.
- In ROOM, scenarios are grouped into packages that may be prioritized. This prioritization is used when planning incremental development. ROOM has an initiator concept similar to Booch.

The semantics of the use case related concepts in the different methods are not corresponding, and thus there is a significant inconsistency between the methods with respect to how the concepts are interpreted. Consequently the use case concept as such becomes fuzzy and several issues need to be clarified [16]. The inventors of OOSE, OMT and Booch are currently striving at a unified method, and in the future we will hopefully see a convergence between the different sets of concepts. (In the next section we will define the concepts that will be used later on in this paper. It remains to be seen how these concepts correspond to use case related concepts in the emerging unified method.)

Natural language is the main tool, in most current methods, for describing use cases at the requirements level. Graphical representation, if present, is only used in the design phase. An exception is ROOM, that uses Message Sequence Charts for describing linear scenarios already during requirements specification. Natural language gives freedom and expressive power, but little support for visualization and automated syntactic and semantic checking.

In all methods, use cases are mainly a support for design, and here use cases are used for finding objects and determine the system structure. Although ROOM mentions scenarios as a means for expressing functional requirements, no methods have, in our opinion, fully exploited use cases in pre-requirements specification activities.

In general, the methods focus on internal rather than external system behaviour, thus making their use case approach less suitable for requirements engineering.

Furthermore, no modularisation concepts are given to manage large use case models. The *uses* and *extends* concepts in OOSE, and the *adds* concept in OMT [5], are presented as a means to provide extensibility, but no rigorous semantics are provided for these concepts.

A general disadvantage of current use case approaches is that we get a loose collection of use cases which are separate, partial models, addressing narrow aspects of the system requirements [11]. When we divide the system requirements into different partial models of usage we face the problem of relating these views and keeping them consistent with each other [17].

Another general problem with use case modelling is *granularity.* How detailed should we be when describing use cases? How large should the scope of each use case be? The current methods give little support to tackle this problem.

3. Ontological and Methodological Issues

In order to create a method that gives sufficient support in model creation, it is vital that the conceptual framework applied by the method is well defined and easy to interpret. This section is aimed at providing clear definitions of the concepts that will form a basis for the hierarchical use case model presented in the next section. In addition to the concepts of actor and use case, this section will extend the conceptual framework of use case analysis with a number of additional concepts, such as *service*, *goal* [18], and *episode* [7]. The concepts *scenario* and *use case* have often been used synonymously [6], but within the presented framework, they will be distinguished.

3.1 Concepts

We start with the demarcation of what is inside and what is outside the system to be built. The intended system is called **target system** and the environment in which the system will operate is called the **host system**. The **users** of a target system belong to the host system and can be either humans or software/hardware based systems. Inside the target system we have a number of services. A **service** is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have.

Users can be of different types, called **actors**. A user is thus an instance of one actor (or possibly many actors, if user types are defined as overlapping and thus allowing for multiple classifications). An actor (also called *user type*) represents a set of users that have some common characteristics with respect to why and how they use the target system. Each actor has a set of **goals**, reflecting such common characteristics. In other words; *goals* *are objectives that users have when using the services of a target system; goals are used to categorize users into actors.*

A **use case** models a usage situation where one or more services of the target system are used by one or more users with the aim to accomplish one or more goals. A use case may either model a successful or an unsuccessful accomplishment of goal(s). Every use case has a **context** that demarcates the scope of the use case and defines its **preconditions** (properties of the host and target system that need to be fulfilled in order to invoke the use case) and **postconditions** (properties of the host and target system at use case termination).

A use case may be divided into coherent parts, called **episodes**. The same episode can occur in many use cases. Each episode consists of **events** of three kinds: **stimuli** (messages from users to the target system), **responses** (messages from the target system to users), and **actions** (target system intrinsic events with no communication between the target system and the users that participate in the use case). Stimuli and responses can have **parameters** that carry data to and from the target system. Actions may have input and output parameters.

These concepts build up our conceptual framework for use case analysis. Some of the concepts and their relations are illustrated in figure 1.

A use case can be described at different levels of abstraction. Three levels can be identified. At the **environment level**, the environment of each use case is described by associating it with related actors, services and goals. At the **structure level** the episode structure of each use case is described by defining *sequencing*, *alternatives*, *repetitions*, *exceptions* and *interrupts*. At the **event level** the episodes are described in further detail in terms of the events that occur in each episode. The event level orders the events by the same means as episodes are ordered at the structure level.

A **scenario** is a realisation of a use case described as a sequence of a limited number of events with linear time order. A use case may cover an unlimited number of scenarios as it may include undetermined repetitions and interrupts that may occur at any time, whereas a scenario is a specific and bound realisation of a use case. The structure of a scenario is flat, with repetitions and alternatives evaluated.

It is possible to have different degrees of scenario instantiation [7]; a completely instantiated scenario corresponds to a system usage trace, where the sequence of events is totally ordered and every parameter has a specific value. A scenario may also be on a slightly higher level, having symbolic names instead of specific parameter values.

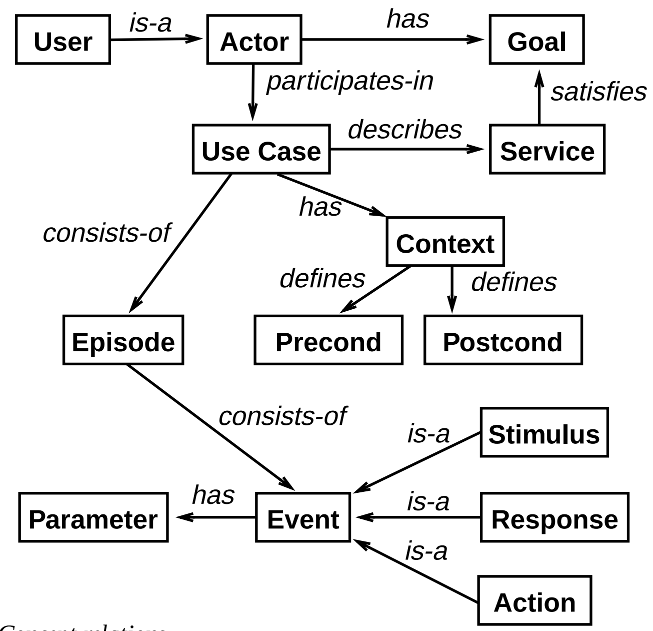


Figure 1. *Concept relations.*

3.2 Model Creation

An important question when applying use case modelling is: What criteria should we have for creating and describing use cases? This question relates to the problem of *granularity*: How large should the scope of a use case be, and at what level of detail should use cases be modelled? By the introduction of goals and services into our conceptual framework we can use them to form a lower bounding criterion for use case granularity: *The scope of one use case is at minimum covering how one goal is satisfied or unsatisfied by the usage of one service*. When it is relevant to model that users have multiple goals to be realized simultaneously, and use many services in combination, the scope has to be extended to cover these cases.

When creating use case models it is possible to apply two principally different approaches; top-down and bottom-up. A top-down approach starts with eliciting a number of use cases that are further refined with respect to their structural properties. A bottom-up approach starts with concrete examples of usage scenarios that are further generalized and synthesised into use cases that encompass these scenarios and more [11]. We believe that the two approaches can complement each other, and that they, together with an appropriate representation, can be incorporated as iteratively performed activities in the same method. The top-down and bottom-up approaches are illustrated in figure 2.

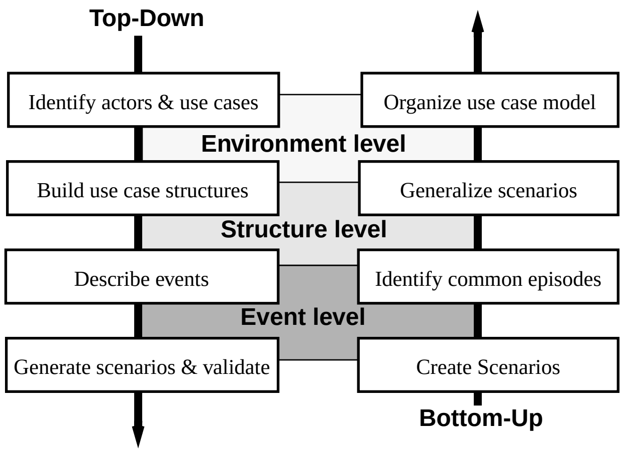


Figure 2. *Model creation approaches.*

3.3 Managing Complexity

When dealing with complex systems, the structure of the use case model is critical. It is vital to have modularisation constructs for dividing the use case model into manageable units. Another common way to cope with complexity is to use hierarchical models, allowing successive refinement of entities into descriptions with more details.

In large use case models we can, at the environment level, organize use cases into packages depending on what services they describe. Use cases that describe usage of a single service may be packaged together into a "service usage package". Special "service combination packages" can be created, when it is possible to use services in combination with each other. Such combination packages include use cases that describe how two or more services are used together. Thus the concept of services allows for dividing large use case models into manageable units.

The concept of episodes supports reuse of use case parts, and thus avoids duplication of use case specification work. This will be of help in managing large use case models by avoiding rework and giving a more compact use case model.

In addition, the hierarchical nature of the use case model helps to manage complexity, as the use cases can be viewed on different levels of detail. The environment level provides an overview, without cluttering the picture with details. The detailed descriptions on the event level are organized by the structure level.

4. Representation

Given the ontological and methodological background, this section presents a hierarchical use case model with graphical representation, after a brief discussion on some different possible representation alternatives.

4.1 Representation alternatives

Different application domains may need different notations for documenting use cases. It is also possible that in the same method, different notations are used at different stages. There are numerous alternative languages for representing use cases. Some of them are listed below.

1. Natural language without restrictions,
2. Structured natural language,
3. Pseudo code,
4. Data flow diagrams,
5. State machine based notations,
6. Event based (e.g. MSC).

Natural language provide expressive power but little support for automation. Some problems can be tackled by structuring natural language specifications into enumerated sections with defined contents [11]. The third alternative is a mix between structured natural language and some programming language and thus gives weak support for graphical visualisation. Alt. 4, although graphical, has the disadvantage of being biased towards internal, rather than external system behaviour. Alt. 5 we believe is rather difficult for the average end user to comprehend.

To accomplish a step-wise formalisation of use cases that seamlessly introduces formality, we recommend to start with structured natural language descriptions that are subsequently formalised into an unambiguous graphical language with event based semantics. Such a language needs to include constructs for managing complexity, e.g. mechanisms for hierarchical decomposition, as discussed earlier.

4.2 Hierarchical Event Based Representation

In this and the subsequent sections we describe a graphical notation for use cases, based on an extension of Message Sequence Chart (MSC) [12]. In the domain of distributed real-time systems (e.g. telecommunication systems) MSC is a common and well known language for describing temporal relations between events. For use case modelling, the MSC language needs to be extended with further constructs to encompass, for example, alternative flows and iteration.The presented graphical language includes such constructs[1].

Figure 3 gives an overview of the hierarchical use case model. Each level is described in more detail in the following sections. The grey arrows indicate refinement links from higher level entities into more detailed level descriptions, and are not part of the graphical syntax

In the subsequent sections, each level is illustrated by a small example system, called “Access control”, which controls a number of card and code terminals and locks on doors in an office building, where access is restricted. The system can open selected doors after an employee has put in her card and given the correct code. The system also keeps track of when employees have entered and exited the building. Visitors can via an entrance door terminal call the security section to ask for access. Some administrative features are also provided, such as daily printouts of entering and exits.

We will not go into detail on all aspects of this system, and not discuss how the system’s use case model was created from a methodological point of view. We only show some parts of the model to exemplify the notation.

1. The presented MSC extensions are under consideration by ITU-T Study Group 10, for incorporation in the comming ITU-T MSC Standard

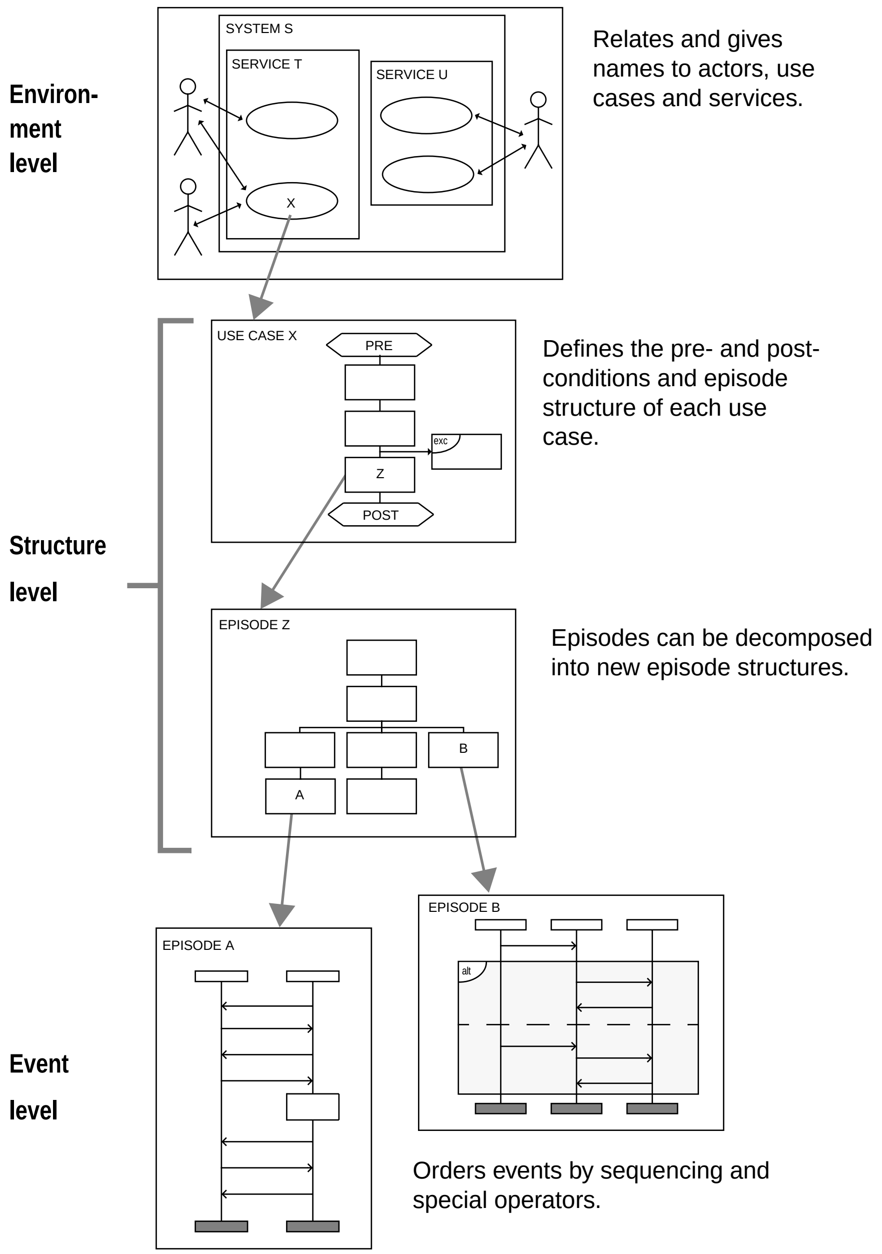

Environ-
ment
level
SYSTEM S
SERVICE T
SERVICE U
X
Relates and gives names to actors, use cases and services.
Structure
level
USE CASE X
PRE
exc
Z
POST
Defines the pre- and post-conditions and episode structure of each use case.
EPISODE Z
B
A
Episodes can be decomposed into new episode structures.
Event
level
EPISODE A
EPISODE B
alt
Orders events by sequencing and special operators.

Figure 3. *Overview of hierarchical use case model.*

4.3 Environment Level

Figure 4 shows the environment level of the example system, with its actors, use cases and services. The notation here is similar to the notation used in [3], but we have added *services*, allowing use cases to be encapsulated into packages, representing a demarcated functional entity.

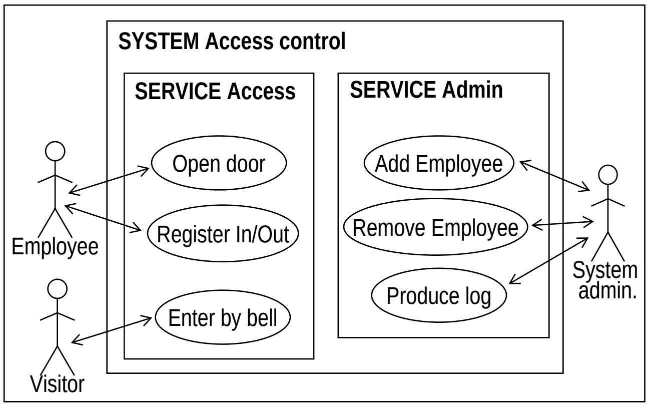
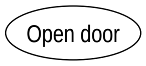
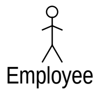


Figure 4. *Environment level example.*

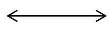
These symbols are used on the environment level:



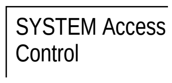
Use case. Oval with a unique name, describing the use case from the main actor's point of view.



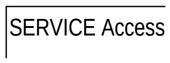
Actor. Icon with a name denoting the user type.



Interaction. Arrow that shows interaction between one use case and its environment (the actors in the host system).



System. The name of the system and the boundary between the system and its environment. Contains services.



Service. The name of the service. Contains the use cases of this service.

4.4 Structure Level

The structure level describes each use case as a graph of episodes. Each episode is a named use case part, representing a demarcated and coherent flow of events. Pre- and postconditions are defined at this level. Operators can be used to express alternative, repetition, exception and interrupt. An *exceptional episode* is a part of a use case that may occur at a specific point (i.e. either zero or one time). If an exception occurs, the use case is terminated. An *interrupting episode* can occur at any point in a use case and may either terminate the use case or resume it at the point where the interrupt occurred.

An episode can be expanded into new episode structures in a hierarchical manner. A leaf episode in this hierarchy is defined on the event level (see next section) as a message sequence chart. Episode decomposition is used to make use cases easier to understand and of manageable size, but also for allowing reuse of episodes across many use cases.

Figure 5 shows a use case in our example system, where we have used pre- and postconditions (drawn as named hexagons). Episodes are drawn as named rectangles.

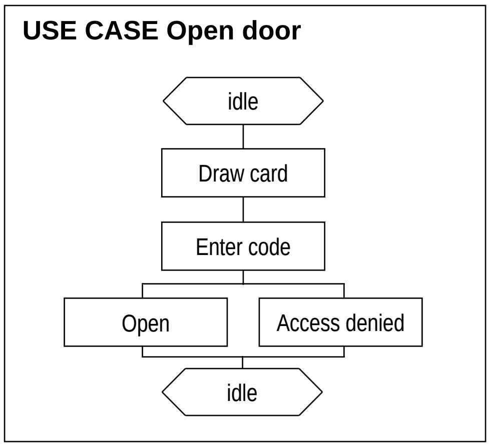


Figure 5. *Structure level example.*

Time progresses downwards. The flow lines that connect episodes indicate sequencing. Alternative (also called choice) is indicated by a flow line that is split and then connected to two or more episodes.

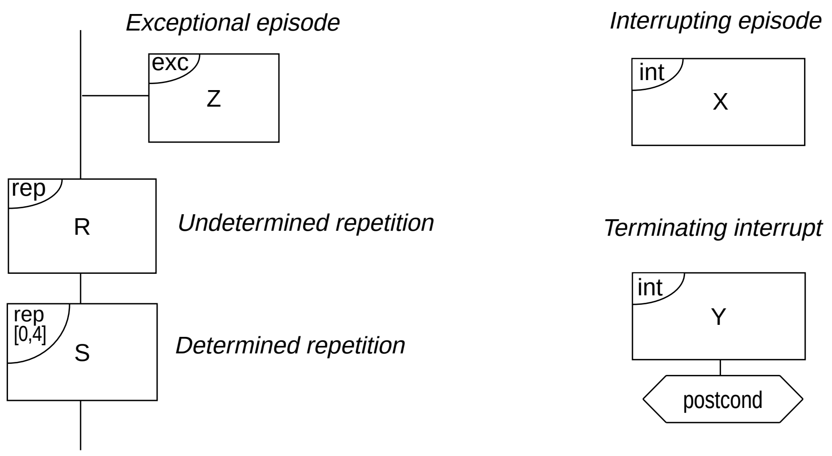


Figure 6. *Exception, repetition and interrupt.*

Figure 6 shows interrupts, exceptions and repetitions. Interrupts are not connected with the flow of other episodes, as they can occur anywhere. If an interrupt terminates the use case, it is connected to a hexagon labelled with the name of the postcondition.

Exceptions are attached to a flow line, indicating where it may occur. Repetitions can have brackets indicating upper and/or lower bounds. A fully undetermined repetition means that the episode occurs at least one time, i.e. there is no upper limit on the number of repetitions.

The int, exc, and rep operators can be used on multiple episodes by using an extra operator frame around them, as shown in figure 7.

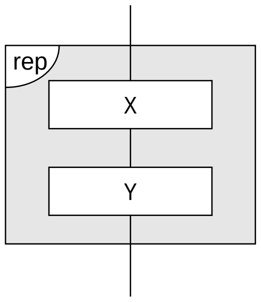


Figure 7. *Multi-episode operator.*

4.5 Event Level

On the event level the detailed flow of events in each use case episode is described as an MSC. There are three kinds of events: stimuli, responses and system actions. An MSC expresses the temporal relation between these events. Operators for alternative, repetition, exception and interrupt are also used at this level. Operators can be nested. Actors on the event level are explicitly instantiated, to allow for multiple instances of the same actor to participate in the same event flow. Figure 8 shows an example of an event flow.

The “Enter code” episode shows the nesting of a repetition and an exception operator. Shading indicate level of operator nesting. Timers are used to keep track of time.

The episode starts with a response DisplayMsg and then repeats for four times the stimuli Key. At the beginning of the repetition, a timer is

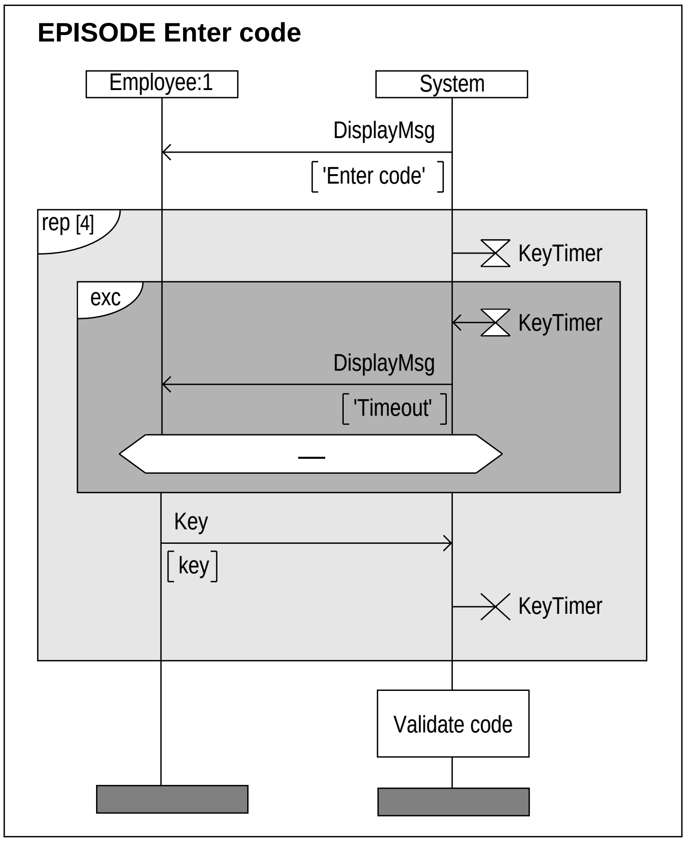


Figure 8. *Event level example.*

set, and normally the Key stimulus comes before the timer expires. If the exceptional case occurs, where the employee waits too long between keys, the timer expires and the use case is terminated after a DisplayMsg response. If all four keys are entered in time, the system action “Validate code” is performed.

The following symbols are used at the event level:

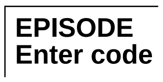
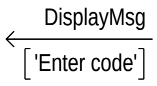
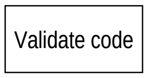


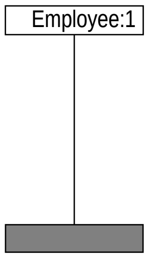
Diagram name. “EPISODE <name>”, where <name> corresponds to the episode name on the structure level.



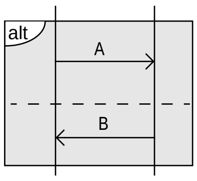
Stimulus/Response. An asynchronous message is drawn as an arrow. It has a name and optional parameters inside brackets. The parameter could either be a specific value or a name.



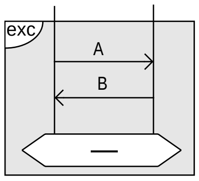
System action. An intrinsic operation that involves no communication with the actor instances of the episode.



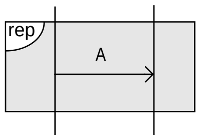
Instance. An instance represents a participant of the use case (actor instances or the system). It contains an instance head, an axis and an end. The instance head contains the instance name, which is either “System“ or an actor name followed by a colon and an instance number. The instance axis orders its events in time. The axis is ended with a filled rectangle indicating the end of the episode.



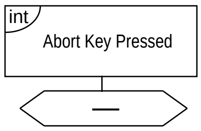
Alternative. The alt operator indicates a choice between two or more alternative flows separated by dashed lines. The example here means that either the stimulus A, or the response B, will happen.



Exception. The exc operator indicates exceptional events. If the exception occurs, the use case is terminated. This is indicated by a condition symbol with a dash that refers to the postcondition label of the use case.



Repetition. The rep operator indicates repetition. If no interval in brackets are specified, the events inside the rectangle are repeated one or more times.



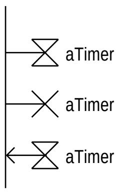
Interrupt. If interrupts are to be expressed on the event level, this is done by referring to another episode. An episode rectangle with an int operator is drawn beside the instances, similar to the structure level notation.

Timer. A timer has a name and may be associated with a specific time duration. A timer has three possible events:

Set. The timer is activated.

Reset. The timer is deactivated.

Timeout. The timer expires.



The current MSC standard from 1993 [12] does not include the presented operators alt, rep, int, and exc. There are also some constructs in [12] that we have not included in this presentation, such as co-regions and instance decomposition, although they might be of interest in some modelling situations. There are also some additional constructs in [10] and [11] that are not included in this short presentation.

5. Conclusions and Further Research

The presented hierarchical use case model with graphical representation has been verified on relatively small examples. From these examples we can conclude that a graphical representation of use cases with well defined syntax and semantics helps to visualize functional requirements, and also gives a more structured, and less ambiguous use case model, compared to natural language representations. To fully assess the virtues of the presented model, more extensive case studies, that cover larger systems, are needed.

We also plan to investigate constructs for supporting *extensibility.* When extending an existing system with new services that reuse or redefine existing services, it is preferable to have mechanisms that prevent a restructuring or duplication of existing services. This relates to the *uses* and *extends* concepts in OOSE, and the *adds* concept in OMT. Well defined extensibility mechanisms are needed, not only on the environment level, but also on the structure and event levels.

Another area of further research is *goal modelling* [18] and its relation to use case modelling. We here see the opportunity to further strengthen the methodological support for use case model creation by incorporating a semiformal goal model that enables expression of inter-goal relations and relations between goals and use cases.

In the presented model, system intrinsic actions are represented as named entities with temporal relations to other events. Other requirements on actions, such as temporal constraints or relations between their inputs and outputs can be expressed e.g. by comments in natural language attached to actions. In further development of the model, it could be useful to include a formal language for system action specification.

In order to provide *tool support*, not only for diagram drawing, but also for automated analysis and checking, we need to formally define the syntax and semantics of the presented use case model in some meta language, e.g. process algebra.

Finally, we want to stress the importance of relating the use case model to other requirements documentation. In a tool environment it would, for instance, be valuable to have the possibility of creating links between requirements in a requirements management database, stated as interrelated “shall-statements”, and use cases in a use case model, in order to be able to express mappings between requirements and usage models and thus supporting traceability between these models.

References

[1] Bubenko, J. A., “Challenges in Requirements Engineering”, *Proceedings of Second International Symposium on Requirements Engineering*, York, UK, March 1995.

[2] Pohl, K., “The Three Dimensions of Requirements Engineering”, *Proceedings of 5th International Conference on Advanced Information Systems Engineering*, Springer-Verlag, 1993.

[3] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., *Object-Oriented Software Engineering- A Use Case Driven Approach*, Addison-Wesley, 1992.

[4] Booch, G., *Object-Oriented Analysis and Design with Applications*, 2nd Ed., Benjamin/Cummings Publ., 1994.

[5] Rumbaugh, J.,“Getting Started - Using use cases to capture requirements”, *Journal of Object-Oriented Programming*, September 1994.

[6] Gough, P., Fodemski, F., Higgins, S., Ray, S., “Scenarios - an Industrial Case Study and Hypermedia Enhancements”, *Proceedings of Second International Symposium on Requirements Engineering*, York, UK, March 1995.

[7] Potts, C., Takahashi, K., Anton, A., “Inquiry-Based Requirements Analysis”, *IEEE Software*, March 1994.

[8] Hsia, P., Samuel J., Gao J., Kung, D., “Formal Approach to Scenario Analysis”, *IEEE Software*, March 1994.

[9] Lindvall, M., *A Study of Traceability in Object-Oriented Systems Development*, Licentiate Thesis 462, Dep. of Computer and Information Science, Linköping University, Sweden 1994.

[10]Andersson, M., Bergstrand, J., *Formalizing Use Cases with Message Sequence Charts*, Technical Report CODEN: LUTEDX(TETS-5125)/1-108/(1995)&local 8, 108 p., Dept. of Communication Systems, Lund University, Sweden, 1995.

[11]Regnell, B., Kimbler, K., Wesslén, A., “Improving the Use Case Driven Approach to Requirements Engineering”, *Proceedings of Second International Symposium on Requirements Engineering*, York, UK, March 1995.

[12]Message Sequence Chart (MSC), *ITU-T Recommendation Z.120*, International Telecommunication Union, 1993.

[13]Jacobson, I., Christerson, M., “A growing consensus on use cases”, *Journal of Object-Oriented Programming*, March-April 1995.

[14]Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[15]Selic B., Gullekson G., Ward P. T., *Real-Time Object-Oriented Modeling*, Wiley & Sons, 1994.

[16]Wohlin C., Regnell B., Wesslén A., and Cosmo H., “User-Centred Software Engineering - A Comprehensive View of Software Development”, *Proceedings of Nordic Seminar on Dependable Computing Systems*, Denmark, August 1994.

[17]Easterbrook, S., Nuseibeh, B., “Managing Inconsistencies in an Evolving Specification”, *Proceedings of Second International Symposium on Requirements Engineering*,York, UK, March 1995.

[18]Dardenne A., van Lamsweerde A., Fickas S., “Goal-directed Requirements Acquisition”, *Science of Computer Programming*, Vol. 20, 1993.

PAPER III:

Towards Integration of Use Case Modelling and Usage-Based Testing

Björn Regnell, Per Runeson and Claes Wohlin

Accepted for publication in *Journal of Systems and Software*.

III

Abstract

This paper focuses on usage modelling as a basis for both requirements engineering and testing, and investigates the possibility of integrating the two disciplines of use case modelling and statistical usage testing. The paper investigates the conceptual framework for each discipline, and discusses how they can be integrated to form a seamless transition from requirements models to test models for reliability certification. Two approaches for such an integration are identified: integration by model transformation and integration by model extension. The integration approaches are illustrated through an example, and advantages as well as disadvantages of each approach are discussed. Based on the fact that the two disciplines have models with common information and similar structure, it is argued that an integration may result in coordination benefits and reduced costs. Several areas of further research are identified.

1. Introduction

Over the last decades, much effort has been devoted to software implementation issues. However, as complexity grows, software development needs more than just programming. Design paradigms, such as object orientation, have entered the scene claiming to provide robust architectures and reusable components. Recently, the focus of the software research and practice has also approached issues related to requirements specification and reliability certification.

This paper presents recent research related to both these areas. The basic idea behind the presented work is to combine and integrate two different approaches that focus on the modelling of usage:

- **Use Case Modelling,** UCM, (Jacobson et al., 1992) and
- **Statistical Usage Testing**, SUT, (Mills et al., 1987).

Both UCM and SUT address phenomena related to the modelling of anticipated system usage, although with different background and terminology. UCM focuses on requirements analysis and usage modelling as a tool for describing and understanding requirements, while SUT focuses on usage modelling to enable test case generation for reliability estimation and certification.

In a survey of industrial software projects (Weidenhaupt et al., 1997), it is concluded that there is an industrial need to base system tests on use cases and scenarios. The studied projects, however, rarely satisfied this demand, as most projects lacked a systematic approach for defining test cases based on use cases.

UCM was introduced in the object-oriented paradigm (Jacobson et al., 1992) to complement traditional static object models with dynamic aspects. The work on use case modelling has in an object-oriented context primarily been focused on the transition from use case based requirements to high-level design, and how use cases can be used to find good object structures (Jacobson, 1995; Buhr and Casselman, 1996).

SUT, on the other hand, has focused on how to create a usage model that allows for generation of test suites which resemble operational conditions, by capturing the dynamic behaviour of the anticipated users. SUT research has concentrated on how such a model can be made scalable (Runeson and Wohlin, 1992; Wohlin and Runeson, 1994), but the main focus has been on the usage model itself from a testing perspective and not on the process of creating it from a requirements perspective. Hence, we see the need to study the common denominator of UCM and SUT in the perspective of requirements engineering, in search for an integrated framework for usage modelling.

There is an intimate relation between requirements specification and system validation; the major goal of validation is to show, for example through testing, that a system correctly fulfils its requirements. This fact is the main motivation behind combining and integrating use case modelling and statistical usage testing. Modelling effort related to the specification of system usage hopefully can be minimised if it can be used for both purposes (Wohlin et al., 1994). Software developers want to avoid modelling the same thing twice.

The main challenges in this work are:

- What concepts in use case modelling on the requirements level can be utilised in usage-based testing for reliability certification?
- How can we create a seamless transition between usage models for requirements specification and usage models for testing?

The presented work includes a conceptual study of each of the approaches, together with some preliminary results on how the approaches can be combined. Future work on an integrated usage modelling approach is also discussed. Chapter 2 presents the major motivations to integration and gives a general overview of integration approaches. Chapter 3 focuses on concepts in usage analysis for requirements specification with use cases, and Chapter 4 focuses on concepts in usage analysis for system validation with statistical usage testing. Chapter 5 presents two approaches to integrated usage modelling in more detail. Some conclusions are presented in Chapter 6.

2. An Integrated View on Usage Modelling

A schematic picture of the software development process is shown in Figure 1. The figure focuses on two dimensions: time and abstraction level. System development progresses from requirements elicitation and analysis, through design, implementation and testing, to validation and certification before delivery of a new system release. During this progress of activities, the focus changes with respect to abstraction. Development starts with an external view of the system, continues during design and

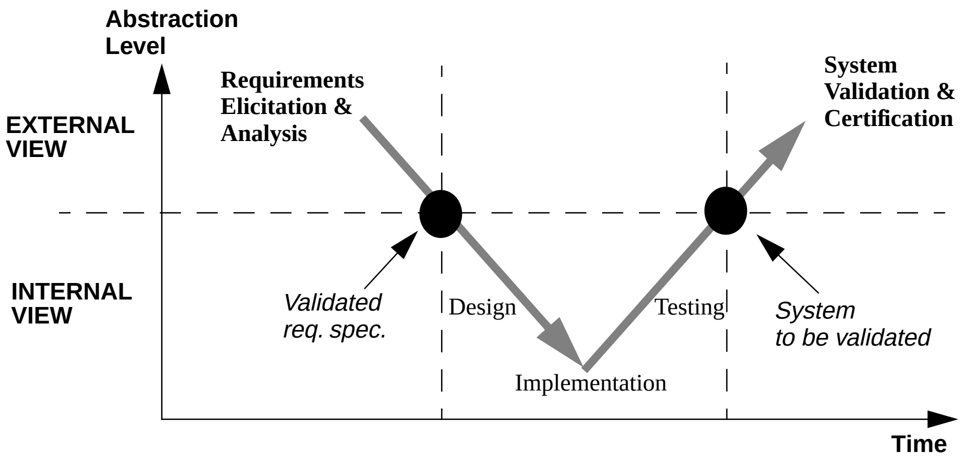


Figure 1. *Different abstraction levels over time.*

implementation with a more detailed focus on system internal issues, before returning to an external “black-box” view. The external view focuses on externally observable functionality as seen by the users of the system, while the internal view focuses on system architecture and the structural and behavioural properties of objects within the system.

Usage modelling refers to the analysis and representation of system usage from the external viewpoint and thus fits well with both the early and late phases in Figure 1. This external view enables usage modelling on a high abstraction level, preventing models to be cluttered with internal details of the system.

Usage modelling deals with the dynamic relations between the events that take place when the system is used by its users. Thus, we are interested in stimuli to the system and its responses from the system actions issued by user stimuli. From a *requirements perspective* we want to capture what events should take place and in what order. Use case modelling, as presented in Chapter 3, is focused on the modelling of such dynamic aspects. A use case model describes a collection of use cases. Each use case covers a set of scenarios. The use cases determine the order of the events and define the possible alternatives in the flow of events.

When the system has been implemented, its function is determined by its program. System usage is, however, due to the free will of its human users, non-deterministic. The flow of events in system usage includes points where the next event is determined by a user action. A use case model represents the different usage possibilities and thus allows for non-deterministic choices of the users. From a *testing perspective*, we want to be able to capture this non-determinism in statistical terms by quantifying the probabilities (frequencies) of different alternative usages, in order to make the testing conditions resemble the foreseen operating conditions. Statistical usage testing, as presented in Chapter 4, is focused on both the usage dynamics and usage statistics, and quantifies the usage probabilities in, so called, usage profiles or operational profiles.

The statistical properties of system usage are also interesting from a requirements perspective. Information on, for example, how often a certain service is used is important input to the process of prioritizing requirements (Karlsson, et al., 1998). The probabilities of combining certain services are vital information when analysing how service combinations can interfere (Kimbler and Wohlin, 1995). Usage frequencies may also be used to optimize user interfaces.

Both Requirements Engineering (RE) and the Verification & Validation (V&V) have the same challenge of completeness and coverage. Have we covered all the essential requirements? Does the set of test cases cover adequately the requirements? Limited resources may require both requirements models and test models to be partial, giving the challenge of finding a level of coverage that is a good approximation of the complete system usage.

In summary, the *commonalities* between RE and V&V include:

1. Both areas desire models of system usage.
2. Both areas strive at an external view of the system.
3. Both areas benefit from quantification of usage frequencies.
4. Both areas have the challenge of adequate coverage.

Besides these commonalities, the motivation for integrating usage models in RE with usage models in V&V is based on the following *expected benefits*:

1. Modelling effort is reduced, as the same information is used for many purposes.
2. Traceability from requirements to test is promoted, which can be assumed to lead to less expensive maintenance.

Both RE and V&V have a variety of approaches proposed for usage modelling. In (Rolland et al., 1998) a survey of existing literature on scenarios and use cases in RE shows a great span of available methods. In (Jarke, et al., 1998) a survey of industrial practise revealed a great diversity in the ways that scenarios and use cases are applied. In (Graham, 1994) a number of different black-box testing techniques are outlined.

In this exploration of the possibilities of integration, we have chosen to focus only on two particular approaches: an extended version of Jacobson's use case modelling (Regnell, 1996), and an extended version of Whittaker's state-based markov model (Whittaker and Thomason, 1994; Runeson and Wohlin, 1995). These approaches are summarised in Chapter 3 and 4 respectively. Other specific approaches to use case modelling and usage-based testing can of course be combined in a number of ways. The specific integration approaches presented in Chapter 5 may be used as input to further research on the integration of other specific RE and V&V approaches to usage modelling.

In general, two different *integration strategies* can be identified, as illustrated in Figure 2.

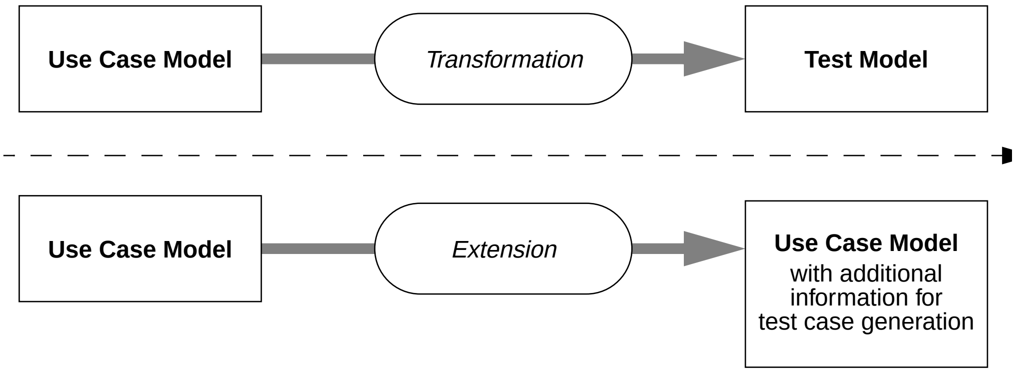

Use Case Model
Transformation
Test Model
Use Case Model
Extension
Use Case Model with additional information for test case generation

Figure 2. *Two ways of integrating usage modelling.*

The first integration strategy, **model transformation**, is based on the assumption that two different usage models are used: one for RE and one for V&V. The integration strategy requires guidelines for how to transform the information in the use case model in combination with additional test-specific information in the process of test model building.

The second integration strategy, **model extension**, is based on the assumption that a tailored use case model can be used directly for V&V, if it is extended with additional information necessary for testing.

The presented work represents an initial study on each of these strategies, but hard evidence on which strategy fits best with which context requires further research. There are, however, some basic reflections on the different strategies:

1. Model extension requires only one model, which can be assumed to imply less modelling effort and less expensive maintenance, compared to model transformation.
2. Model transformation may be more appropriate if the models differ greatly between requirements and test. By creating two models tailored for their special purposes, no compromise is needed. The common information may still be utilised through transformation rules.

Before we continue the investigation of the two proposed integration approaches, we present a conceptual study of each of the specific methods for use case modelling (Chapter 3) and statistical usage testing (Chapter 4) respectively.

3. Functional Requirements Specification with Use Cases

The elicitation, analysis and documentation of requirements on software systems is a crucial and non-trivial task (Loucopoulos and Karakostas, 1995; Bubenko, 1995). Well defined concepts and methods are needed when constructing specifications that represent requirements in an unambiguous, consistent, and complete manner. It is also important to have representations of requirements that are easily understood by the different stakeholders that take part in requirements analysis (Pohl, 1993). This Chapter concentrates on use case modelling for eliciting, analysing and documenting functional requirements. The use case concept has gained widespread acceptance within methods and notations such as OOSE (Jacobson, 1992), OMT (Rumbaugh et al., 1991; Rumbaugh 1994), the Booch method (Booch, 1994), ROOM (Selic et al., 1994), Fusion (Coleman, et al., 1994), and UML (Fowler and Scott, 1997).

There are many different possibilities of applying use cases and scenarios in requirements engineering. A survey of european software projects (Weidenhaupt, 1997) concluded that about two thirds of 15 visited projects used the OOSE (Jacobson, 1992) approach extended in various ways. Here we concentrate on one such extension (Regnell, 1996; Regnell et al., 1995; Regnell et al., 1996).

The main purpose of the presentation of the use case modelling concepts in this Chapter, and the statistical usage testing concepts in Chapter 4, is to provide a background to our objective of combining the two disciplines into an integrated framework, as discussed in Chapter 5.

In the subsequent sections, an example from the domain of telecommunication will be used as illustration. This example is a simplification of results from a case study conducted as a prestudy for the presented work, and involves a simplified Private Branch Exchange (PBX) with some common telephony services, such as unconditional call forwarding (CFU).

3.1 A Conceptual Framework for Use Case Modelling

The main idea behind use case modelling is to elicit and document requirements by discussing and defining specific contexts of system usage as they are anticipated by the different stakeholders in the requirements engineering process. The conceptual framework for the presented use case modelling approach (Regnell et al., 1996) and their relations are illustrated in Figure 3. These concepts are described in the subsequent sections.

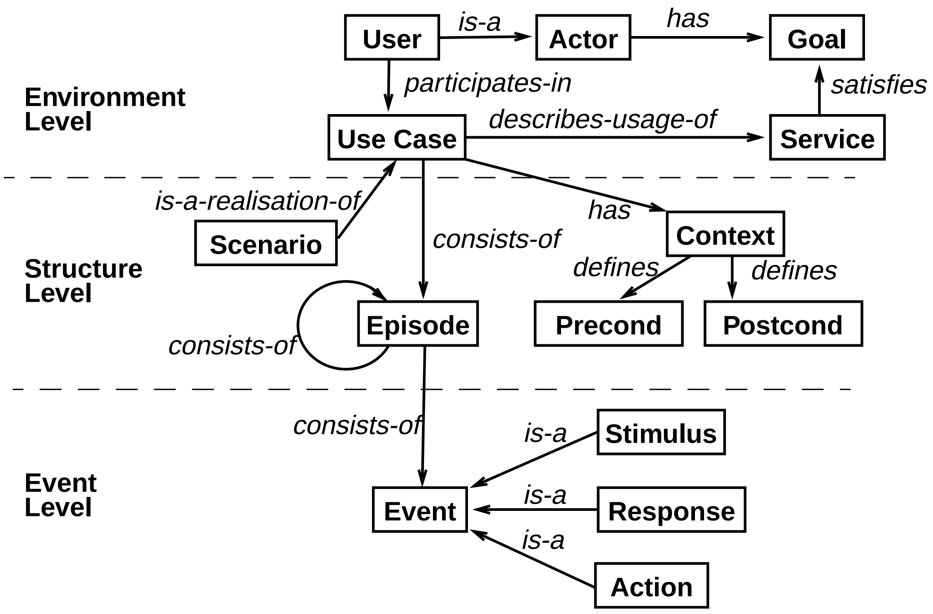


Figure 3. *Concept relations and levels of abstraction.*

Use cases can be viewed on different abstraction levels. At the **environment level**, the use case is related to the entities external to intended system. On this level, a use case is viewed as an entity representing a usage situation. At the **structure level**, the internal structure of a use case is revealed together with its different variants and parts. The **event level** represents a lower abstraction level where the individual events are characterized.

3.2 Environment Level

The **users** belong to the intended target system's environment and can be either humans or other software/hardware based systems. Inside the target system we have a number of services. A **service** is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. Table 1, includes the services of our PBX example system.

Table 1. Services in the PBX example.

Service	Description
NCC	Normal Call with Charging
CFU	Call Forward Unconditional
RMR	Read Markings and Reset

Users can be of different types, called actors. A user is thus an instance of an actor. An actor (also called *user type* or *agent*) represents a set of users that have some common characteristics with respect to why and how they use the target system. Each actor has a set of goals, reflecting such common characteristics. Goals are objectives that users have when using the services of a target system. Thus, goals are used to categorize users into actors. Table 2 shows the goals of the two actors *subscriber* and *operator* in the PBX example.

The goals are described as patterns using general temporal operators such as *achieve*, *cease*, and *maintain* (Dardenne and van Lamswerde, 1993).

A **use case** represents a usage situation where one or more services of the target system are used by one or more actors with the aim to accomplish one or more goals. Table 3 shows the use cases of the PBX example, and their relation to actors, goals, and services.

Table 2. Actors and their goals in the PBX example.

Actors	Goals
Subscriber	GS1 To *achieve* communication with another subscriber
	GS2 To *cease* communication with another subscriber
	GS3 To *achieve* reachability at another destination
	GS4 To *cease* reachability at another destination
Operator	GO1 To *maintain* markings information representing call duration
	GO2 To *achieve* a printout of the number of markings for each subscriber
	GO3 To *achieve* a resetting of the number of markings for each subscriber

Table 3. Uses cases in the PBX example.

Use Cases	Actors	Goals	Services
Normal Call	Subscriber	GS1, GS2, GO1	NCC
Activate CFU	Subscriber	GS3	CFU
Deactivate CFU	Subscriber	GS4	CFU
CFU Call	Subscriber	GS1, GS2, GS3, GO1	CFU
Read Markings	Operator	GO2	RMR
Reset Markings	Operator	GO3	RMR

3.3 Structure Level

The structure level includes concepts that relates to the internal structure of use cases, such as different variants and parts of a use case.

A **scenario**[1] is a specific realisation of a use case described as a sequence of a finite number of events. A scenario may either model a successful or an unsuccessful accomplishment of one or more goals. A use case may cover an unlimited number of scenarios as it may include alternatives and repetitions. A scenario, however, is a specific and bound realisation of a use case, with all choices determined to one specific path. Table 4 shows a number of scenarios identified for the use case *normal call*.

The standardised language of Message Sequence Chart (MSC) (ITU-T, 1996; Regnell et al., 1996) may be used to express the structure level of a use case graphically. A High Level Message Sequence Chart (HMSC) is

1. Some authors use the terms scenarios and use cases as synonyms, but here we distinguish between them, to differentiate between type level and instance level.

Table 4. Scenarios for the use case “Normal Call”

Scenario	Description
Reply	Call to idle subscriber that replies
Busy Subscriber	Call to busy subscriber
No Reply	Call to idle subscriber that does not reply
Non-Existent	Call to non-existent subscriber
Timeout	Subscriber waits too long after offHook

illustrated in Figure 4. Each scenario of the use case “normal call” is represented as an alternative.

Each box with rounded corners refers to either another HMSC at a sub-structure level, or an MSC at the event level

Every use case (and scenario) has a **context** that demarcates the scope of the use case and defines its **pre-conditions** (properties of the environment and the target system that need to be fulfilled in order to invoke the use case) and **post-conditions** (properties of the environment and the target system at use case termination). An example of a pre-condition for the “CFU Call” use case is that “the CFU service has been activated”. An example of a post-condition for the “normal call” use case is that “the caller is idle”. Pre- and post-conditions are shown in Figure 4 as diamond symbols.

It is possible to have different degrees of scenario instantiation (Potts et al., 1994); a completely instantiated scenario corresponds to a system usage trace, where the sequence of events is totally ordered and every parameter has a specific value. A scenario may also be on a slightly higher level, having symbolic names instead of specific parameter values.

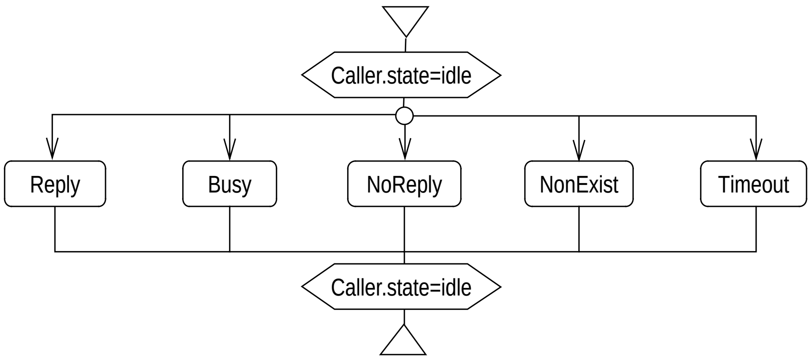


Figure 4. *High Level Message Sequence Chart.*

In use cases and scenarios it may be possible to identify coherent parts, called **episodes**. Similar event sequences may occur in several use cases, and episodes can be used as a modularisation mechanism to encapsulate use case parts and create a hierarchical use case model. We will not go into detail on episodes here, for more information see (Potts et al., 1994; Regnell et al., 1996).

3.4 Event Level

The lower abstraction level of uses cases, scenarios, and episodes includes **events** of three kinds: **stimuli** (messages from users to the target system), **responses** (messages from the target system to users), and **actions** (target system intrinsic events which are atomic in the sense that there is no communication between the target system and the users that participate in the use case).

Stimuli and responses can have **parameters** that carry data to and from the target system. In order to express parameters, and also conditions on data, a use case model may be complemented by a data model. A simple representation of a data model for the PBX example is given in Figure 5.

Given a data model, we may express conditions on the data model that always should be true. Figure 6 shows such invariants for our PBX example.

When describing the information exchange between actors and the target system, it may be useful to define unique names for messages (stimuli and responses) together with information on the data types of their parameters. Table 5 presents the identified messages for our PBX example.

```
toneType =
 (dialTone, ringSignal, ringTone, busyTone, errorTone, infoTone);
maxNumberOfSubscribers: Natural;
SubscriberType: record (
    state: (idle, busy, off);
    telNumber: TelNbrType;
    CFU_active: Boolean;
    CFU_number: TelNbrType;
    markings: Natural;
    talkingTo: SubscriberType);
SubscriberSet:
    Set (1..maxNumberOfSubscribers) Of SubscriberType;
```

Figure 5. *A data model for the PBX example.*

The telephone number is unique:

```
    For-all X in SubscriberSet:
        For-all Y in SubscriberSet:
          if X<>Y then X.telNumber<>Y.telNumber;
```

If state is idle, the subscriber is not connected to another subscriber.

```
    For-all X in SubscriberSet:
       (if X.state=idle then X.talkingTo=nil)
```

If subscriber X is talking to Y then subscriber Y is talking to X:

```
    For-all X in SubscriberSet:
       if X.talkingTo <> nil then
          X. state = busy and
          Exist Y in Subscriberset:
             Y.state=busy and Y.talkingTo=X and
             X.talkingTo=Y;
```

Figure 6. *Some invariants in the PBX example.*

Table 5. Messages in the PBX example.

Message	Description
offHook	From Subscriber when lifting receiver
onHook	From Subscriber when hanging up the receiver
number(telNbrType)	From Subscriber when dialling a Number
activateCFU(telNbrType)	From Subscriber when activating CFU
deactivateCFU	From Subscriber when deactivating CFU
startTone(toneType)	To Subscriber when a tone is given
stopTone(toneType)	To Subscriber when a tone is stopped
readMarkings	From Operator when issuing a reading of markings
resetMarkings	From Operator when issuing a reset of markings
markings(markingListType)	To Operator when reporting the markings

Based on the data model, the message definitions and natural language descriptions of scenarios, MSC can be used to describe graphically the event level as shown in Figure 7.

Each actor instance and the system are represented by a vertical time-axis, with time progressing downwards. Stimuli and responses are represented by arrows between actors and the system. Conditions are expressed as assertions on the data model in Figure 5.

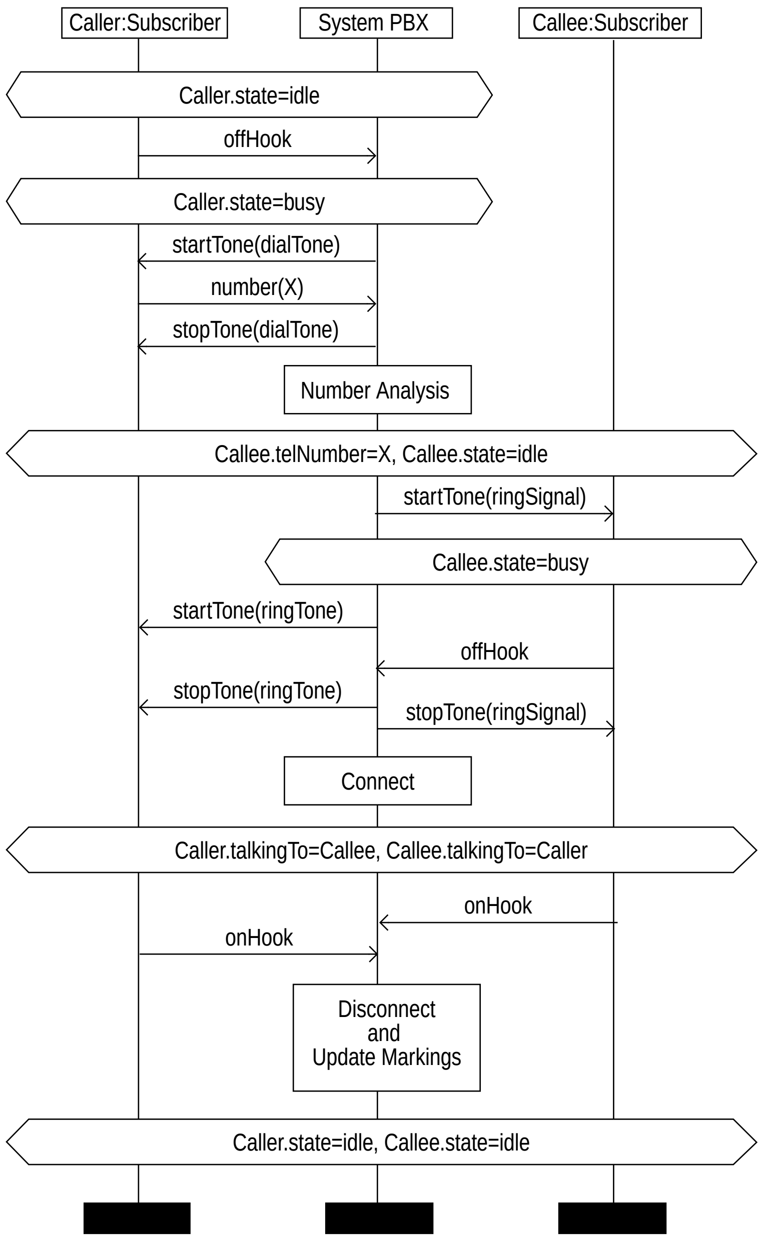


Figure 7. *MSC for the scenario “Reply” of use case “Normal Call”.*

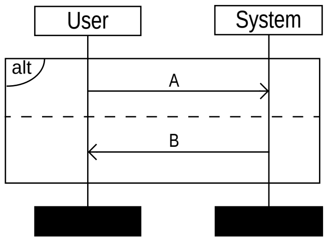


Figure 8. *Example of operator notation at the event level.*

It is also possible to describe alternatives and repetitions on the event level. Figure 8 shows a simple example of an alternative operator on the event level, expressing a choice between either stimuli A or response B. For more details on operators for ordering events, see (Regnell et al., 1996).

4. Reliability Requirements Specification and Certification

Non-functional requirements are an essential part of requirements specifications. In particular, the reliability requirements are often regarded as one of the most important non-functional requirements. The reliability requirements cannot be formulated as a single figure (e.g. probability for failure-free execution or mean time between failures), since more information is needed. The reliability depends not only on system properties, for example correctness, but also on the system environment, i.e. how the system is used. It is necessary to take the anticipated usage into account as the reliability of the system is dependent on the usage; the usage for which the requirement is valid must be stated together with the requirements of the system.

Usage-based testing with reliability certification is a means for validating reliability requirements. Functional requirements are validated at the same time. Thus, usage-based testing allows for both functional requirements validation and reliability certification.

This chapter describes the concepts and representations of a particular usage-based testing approach, and Chapter 5 provides examples of usage-based testing models (using the PBX system of Chapter 3), and discusses how to integrate usage-based testing with use case modelling.

4.1 Usage-based Testing

There are two major approaches to testing: *black-box testing* and *white-box testing*. Black-box testing techniques take an external view of the system and test cases are generated without knowledge of the interior of the system. White-box testing techniques take an internal view and aim at covering all paths in the code or all lines in the code or maximising some other coverage measure. The main objective of all testing techniques is to validate that the system fulfils the requirements; mostly the focus is on functional requirements, but test cases can also address quality issues. For example, they can either be derived with the objective to locate as many faults as possible or to certify the reliability level of the software.

Usage-based testing implies a focus on detecting the faults that cause the most frequent failures, hence maximising the growth in reliability. This paper focuses on black-box testing and in particular on usage-based testing, which can be used to certify a particular reliability level and, of course, to validate the functional requirements.

The ability to certify software during testing is based on a user-oriented approach. This requires a model of the anticipated usage of the software and quantification of the expected usage as the software is released. Several approaches have been investigated and used in this area. Musa (1993), for example, advocates operational profile testing, Mills et al. (1987) discuss random testing based on the operational profile and Runeson and Wohlin (1992; 1995) present an approach with user-state dependent random testing based on the operational profile. The focus in this paper is on the latter approach, i.e. statistical usage testing based on a *state hierarchy model* (Wohlin and Runeson, 1994; Runeson and Wohlin, 1998). The subsequent sections present the conceptual framework for this approach and discuss how the concepts are represented in the state hierarchy model.

4.2 A Conceptual Framework for Usage-based Testing

A system consists of a number of services provided to the system users. These services are implemented by objects. The objective here is to provide a framework for modelling usage and to enable reliability certification of objects that are parts of a system, as well as certification of an entire system. In Figure 9, the relations between target system and environment concepts are illustrated. These concepts form the basis for creat-

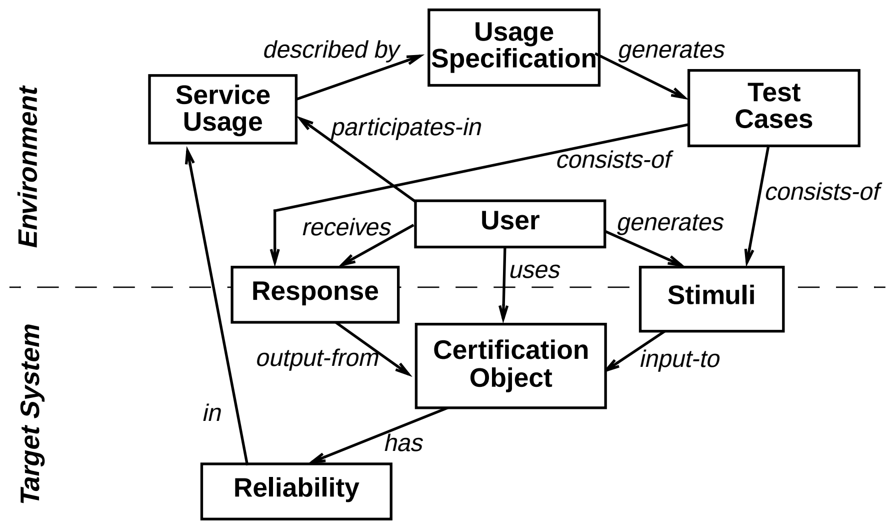


Figure 9. *Certification concepts and their relationships.*

ing a model of the usage of the certification object and also for quantifying the anticipated usage. The concepts of the usage specification test model are further discussed below and shown in Figure 10.

The concepts in Figure 9 can be defined as follows. The software to be certified is referred to as a the **certification object**. A certification object has a certain **reliability**, which is the probability that the object works as intended for a specified time and in a specified **service usage** environment. The certification object is used by one or many users, which can be either human users or other systems. The communication between the user in the **environment** and the **certification object** is made through **stimuli** generated by the user and **responses** sent by the object. The user of the certification object participates in service usage, which is described by the **usage specification**. From the usage specification, **test cases** are generated including stimuli and responses to/from the certification object.

To enable certification, the environment must be modelled to allow for generation of test cases which resemble the anticipated behaviour in the operational phase. Thus, modelling concepts capturing the environment are needed. Depending on the type of testing being applied, different test models have to be derived. The focus here is on usage-based testing, which means that the test model is a **usage specification**, see Figure 10. The usage specification consists of a **usage model**, which describes the

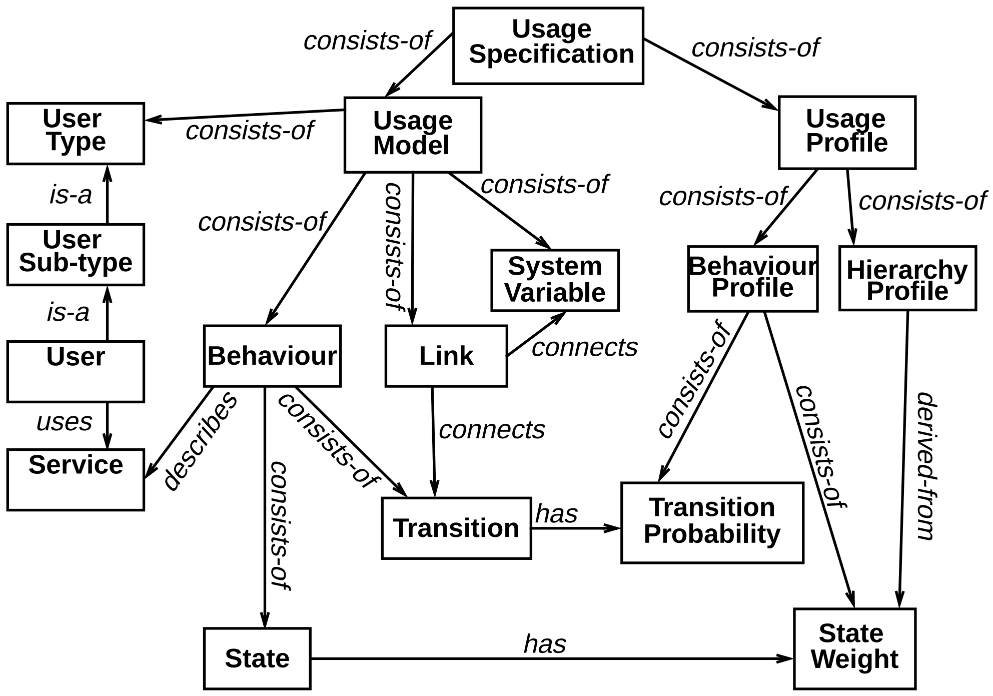


Figure 10. *The usage test model and its usage-oriented modelling concepts.*

possible behaviour of the users, and the **usage profile**, which quantifies the actual usage in terms of probabilities for different user behaviour.

The usage model is described through a hierarchy which defines the **users** and their relations (**user types** and **user sub-types**), the **behaviour** models which define the user **states** and **transitions** between user states, the **system variables** which capture important assets of the system state and the **links** which define connections between different behaviour models and system variables.

The usage profile is divided into a **hierarchy profile**, which describes the probabilities for choosing one specific user in the environment, and the **behaviour profile**, which models the behaviour of a single user, while using the available services.

The hierarchy is a tree structure where the nodes in the tree represent groups of users, based on their usage models and usage profiles. A **user type** is defined as the collection of users having the same possible behaviour (normally equivalent to that the users have the same goal, cf. actor), i.e. they have exactly the same behaviour models. A **user sub-type** is a further division of the users into a group where all users also have the same behaviour profile, hence having a similar statistical behaviour. The **users** are instances of a user sub-type and each user has access to a set of **services**, which usage is described by behaviour models.

4.3 Hierarchical Representation of User Behaviour and Usage Profiles

The usage model has to two main parts, the hierarchy and the behaviour parts. The two model parts are illustrated through a small example in Figure 11 and Figure 12, which show a part of the PBX example.

For each instance of a service, there is a behaviour model, which consists of states and transitions. The services in Figure 11, has been divided into states, and the possible transitions among the states are also shown.

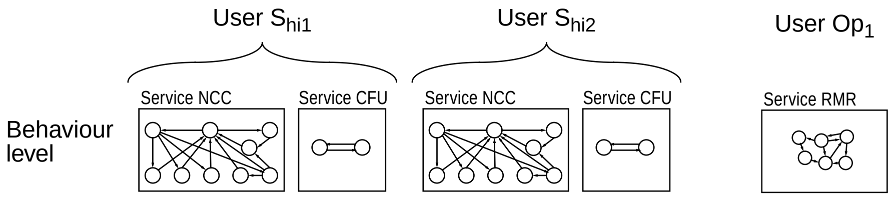


Figure 11. *Behaviour model for a part of the PBX example.*

It should be noted that the states are external states, i.e. **user states**, which are only a subset of the possible system states. User states describe the externally visible states of the system.

The total state of the usage model is a vector T of states for all behaviour models:

$$T_{\text{Behaviour model}} = \left[\ldots \; t_{S_{hi1}\, NCC} \quad t_{S_{hi1}\, CFU} \quad t_{S_{hi2}\, NCC} \quad t_{S_{hi2}\, CFU} \quad t_{Op_1\, RMR} \right]$$

where t_{ij} means the state of service j for user i.

Probabilities are assigned to each arc in the instances of the behaviour models, hence taking different profiles into account. Furthermore, state weights are assigned to each state (denoted W_k for service k), which reflect the overall stimulus frequency of the user being in that state.

In Figure 12, the hierarchy is shown, which breaks down the usage of the certification object into individual users and their services.

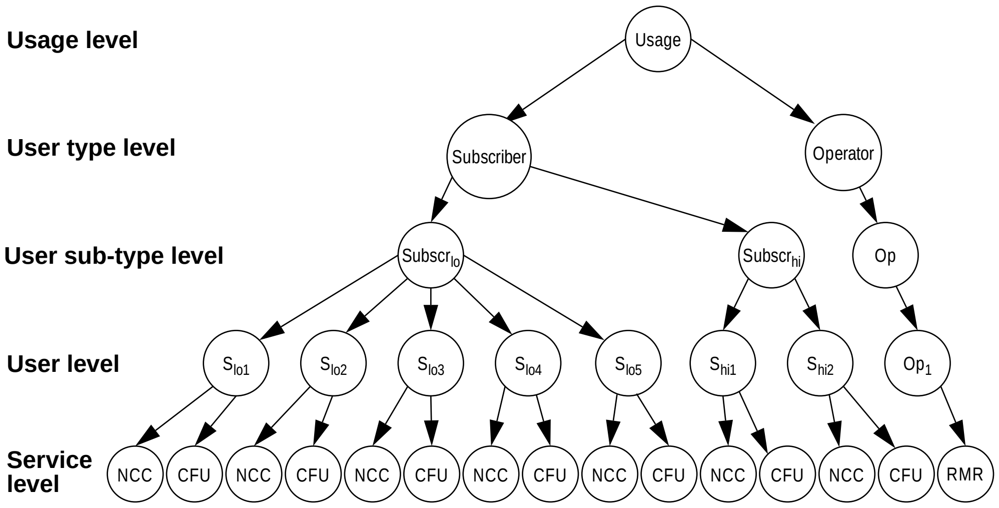


Figure 12. *The hierarchy part of the usage model, using the PBX example*

Two user types have been identified, which implies, for example, that all users of user type *Subscriber* must have the same behaviour model. *Subscriber* is divided into two user sub-types, hence modelling differences in the behaviour profiles, i.e. users S_{lo1} through S_{lo5} do not have the same behaviour profiles as users S_{hi1} and S_{hi2}. These users use the services NCC and CFU.

The *Operator* user type only consists of one user sub-type Op, and only one user exists of this sub-type, i.e. user Op_1. This user uses a single service, i.e. service RMR. (This example is further elaborated in Section 5.1.)

The hierarchy profile is a little bit more complicated. The complication arises as it is reasonable to change the probabilities in the hierarchical profile based on the state of the users. The probability for the selection of a service (denoted p_k for service k) equals the current state weight of the service, divided by the sum of the current state weights for all services.

$$p_k = \frac{W_k}{\sum_j W_j} \text{ if at least one } W_j \neq 0$$

For each transition in a behaviour model, the hierarchy profile is updated. The update algorithm is described in more detail in (Runeson and Wohlin, 1998).

The usage specification is run through, using random numbers. To generate test cases, the tester is supposed to act as the system and provide the expected responses of the certification object using the requirements specification as a basis. The stimuli generated from the usage specification and the responses from the tester are stored on a test file. The test generation procedure is further described in (Wesslén and Wohlin, 1995).

5. Two Approaches to Integrated Usage Modelling

The combination of use cases and usage-based testing provides a comprehensive view of the software development process from a user perspective (Wohlin et al., 1994). The user does not have to bother about internal technical solutions. Instead the user can focus on the external view and the actual use of the system. However, to make the combination seamless, there is a need for bridging the conceptual gap between the two approaches. This Chapter presents two approaches to integration.

As stated in Chapter 1, use case modelling and usage based testing are sprung out of different traditions and have different objectives. It should be noted, however, that both usage-based testing and use case modelling need similar information, which means that the information is not collected solely for either requirements specification or testing purposes; use case models contain much information that can be used for system validation. Although there are many conceptual similarities between use case modelling, as presented in Chapter 3, and usage-based testing, as presented in Chapter 4, there does not exist a simple one-to-one mapping between the concepts in the two disciplines. The use case modelling concepts do, for example, not cover the stochastic semantics of usage profiles, and the state hierarchy model does, for example, not cover the concepts of actor goals or pre- and post-conditions.

When trying to combine the concepts of Section 3 and Section 4 to form an integrated approach to usage modelling, two integration approaches can be identified: (1) we could try to *transform* a use case model into a state hierarchy model by translating the concepts in the former to the concepts in the latter according to some concept mapping rules, or (2) we could try to *extend* the use case model to incorporate stochastic semantics and the necessary information for test case generation.

Section 5.1 and Section 5.2 proposes a working procedure for each of these integration approaches, and the PBX example presented in

Section 3 is followed in both approaches. Section 5.3 discusses their advantages and disadvantages.

5.1 Approach 1: Integration by Transformation

The first approach to integration is based on the observation that many of the concepts in use case modelling and statistical usage testing have similar semantics. For such similar concepts it may be possible to use simple translation guidelines, and together with the necessary additional information on the stochastic properties of users we can create a state hierarchy model by transforming the use case model. This transformational approach is sketched in Figure 13.

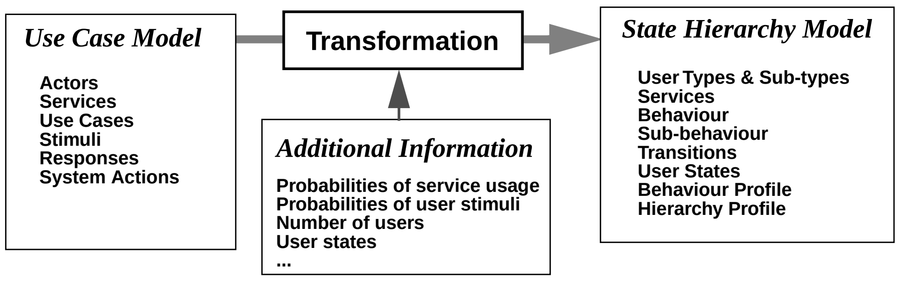


Figure 13. *Transforming a use case model into a state hierarchy model.*

In general, the transformation includes moving from an event based representation to a state-based representation and adding other necessary information.

When creating the state hierarchy model, it may be suitable to follow the method outlined below (Runeson and Wohlin, 1998). The transformation activity uses the concepts captured by the use case model and additional information of usage probabilities and quantities.

1. Identify services
2. Define user types
3. Define user sub-types and instantiate users
4. Create behaviour models
5. Define the behaviour usage profile
6. Define the hierarchy usage profile

This method shall not be seen as completely defined steps; iterations are performed when needed. Below the steps are presented.

Identify services. The services can be used directly as defined in the use case model, see Table 1. In our example the resulting service list is: NCC (Normal Call with Charging), CFU (Call Forward Unconditional) and RMR (Read Markings and Reset).

Define user types and sub-types. The actors in the use case model are the basis for the upper levels of the hierarchy model. Each actor becomes a user type and user sub-types are added if there are different probability profiles for a user type. Additional information on the estimated number of instances of each user sub-type determines the user level in the hierarchy model.

The example has two actors in the use case model, *Subscriber* and *Operator* (see Table 2), which constitute user types in the usage model. The *Subscriber* user type has access to two of the services, NCC and CFU, and the *Operator* user type has the RMR service.

In addition to the use case information, quantitative information is gathered from other sources. There are two variants of the Subscriber type, one with high frequency usage and one with low frequency; each constituting a user sub-type. For each of the user sub-types, the number of instances are defined as well. There are 5 low-frequency subscribers, 2 high-frequency subscribers and 1 operator. These steps result in the hierarchy model as presented in Figure 12.

Create behaviour and sub-behaviour models. The information for the behaviour models are not directly available in the use case model as the information for the hierarchy model are. However, there are parts of the information in the use cases and the scenarios (see Table 3 and Table 4 respectively) which can be integrated to a behaviour model for the service in question. Furthermore, the messages (see Table 5) constitute the interface between the system and its users and will hence appear in the usage model as well.

The behaviour model is a state-transition diagram in which use cases and scenarios constitute parts. The state information can in parts be collected from the pre- and postconditions for the use cases. The messages are attached to the transitions in the behaviour model, as stimuli to the system.

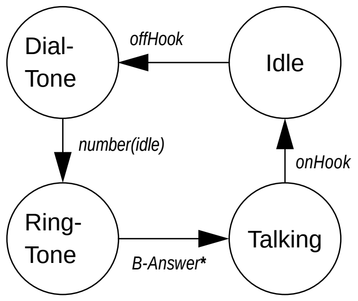


Figure 14. *Behaviour model for service NCC, scenario Reply.*

In our example, the NCC service behaviour model is further elaborated. The Normal Call use case and its five scenarios (Reply, Busy Subscriber, No Reply, Non-Existent and Timeout) is the starting point for the behaviour modelling. The first state to define is the starting state, *Idle*, when no actions haave taken place, see Figure 14. Then we follow the reply scenario, see Figure 7. The first stimulus that can be generated from the subscriber is *offHook*, resulting in the *DialTone* state. Next step is to enter a number and the subscriber state moves into *RingTone*. The called part (called B-part) answers the call and the state is moved into *Talking*. Finally when they close the call with *onHook*, the subscriber is back to the *Idle* state.

All the other scenarios are taken into account in the model, resulting in the model in Figure 15. It can be noted that there are a few new labels on the transitions in addition to the messages in Table 5. Timeouts are modelled as stimuli. There are also transitions labelled B-Answer and B-Calling which involve another behaviour model, denoted with asterisk in the figure. These are replaced with links, meaning that transitions in the other behaviour model causes a transition in the current model.

Define the behaviour usage profile. When the behaviour models are ready, the behaviour profile can be defined. There is no quantitative information on the system usage in the use cases, so this information has to be collected elsewhere. Typical information sources are measurements on earlier releases of the system and interviews with intended users of the system.

Two usage profiles are defined, for the NCC behaviour model, one for each user sub-type. Fictitious data is presented in the tables below.

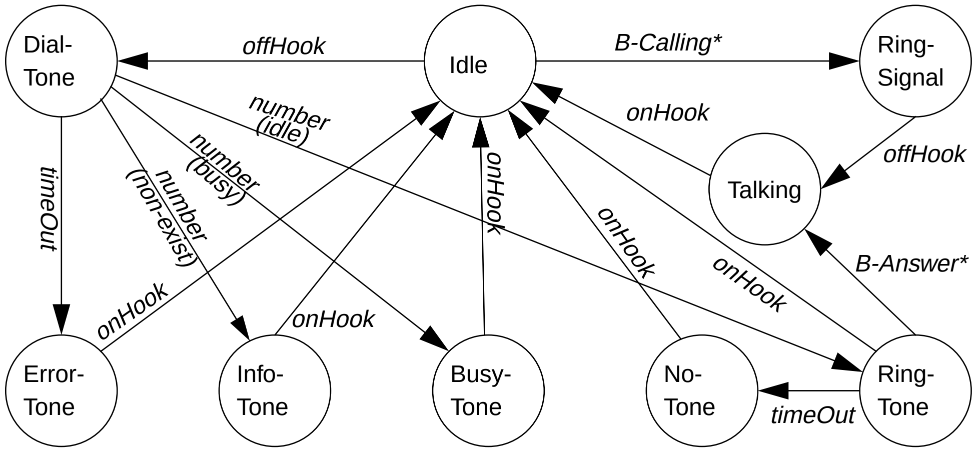


Figure 15. *Behaviour model for service NCC, all scenarios integrated.*

Table 6. Behaviour profile "Subscriber"

State	Transition	$\mathbf{Subscr_{lo}}$	$\mathbf{Subscr_{hi}}$
Idle	offHook	1.0	1.0
DialTone	number(idle)	0.70	0.60
	number(busy)	0.25	0.35
	number(non-exist)	0.03	0.03
	timeOut	0.02	0.02
RingSignal	offHook	1.0	1.0
RingTone	onHook	0.98	0.98
	timeOut	0.02	0.02
Talking	onHook	1.0	1.0
BusyTone	onHook	1.0	1.0
NoTone	onHook	1.0	1.0
ErrorTone	onHook	1.0	1.0
InfoTone	onHook	1.0	1.0

The state weights represent the frequency of use when being in the respective states of the behaviour model.

The state weights for the idle state show that a $\text{subscr}_{\text{hi}}$ has twice as high frequency for starting a talk; the state weights for the talking state show that a $\text{subscr}_{\text{hi}}$ talks 50% longer than a subsc_{lo}.

Table 7. State weights.

State weight	**Subscr$_{lo}$**	**Subscr$_{hi}$**
W $_{Idle}$	1	2
W $_{DialTone}$	100	100
W $_{RingTone}$	100	100
W $_{RingSignal}$	50	50
W $_{Talking}$	15	10
W $_{BusyTone}$	100	100
W $_{NoTone}$	100	100
W $_{ErrorTone}$	100	100
W $_{InfoTone}$	100	100

Define the hierarchy usage profile. Finally the hierarchy profile is calculated, based on the state weights for each service.

$$P_i = \frac{W_i}{\sum_{k=1}^{16} W_k}$$

The concept translations discussed are, as shown in the example, not sufficient for automatic transformation. There is still a need for skill and intellectual work in the creation of the usage model.

It can be concluded that the use case model can be transformed into a usage model. The environment and structure levels contribute to the hierarchy model with a few additional modelling decisions. The behaviour model derivation is supported by the structure and event levels, while the profile information has to be collected from other sources.

5.2 Approach 2: Integration by Model Extension

Instead of creating a completely new model by transforming the use case model into a state hierarchy model, we can adopt the principal ideas behind statistical usage profiles and create an extended use case model, complemented with event statistics. If we can create well defined semantics for how test cases can be generated directly out of the use case model, we will save the effort of making two different models. The model extension approach is illustrated in Figure 16.

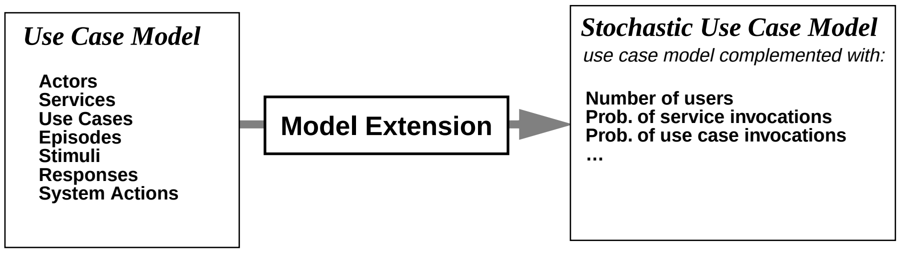

Use Case Model
Actors
Services
Use Cases
Episodes
Stimuli
Responses
System Actions
Model Extension
Stochastic Use Case Model
use case model complemented with:
Number of users
Prob. of service invocations
Prob. of use case invocations
...

Figure 16. *Extending a use case model with additional information needed for test case generation.*

The basic idea behind the model extension is to complement every part of the use case model where there are non-deterministic choices with the probabilities of the different choices.

Environment level. On the environment level, we have to extend the use case model with information on the number of instances of each actor and the probabilities for each actor to generate stimuli to the system. Furthermore, it has to be analysed if there are variants of actors with respect to their usage profile.

In our example, there are 7 instances of the subscriber actor and 1 instance of the operator actor. There are two variants of the subscribers, with respect to their usage frequency, subscr$_{hi}$ and subscr$_{lo}$. This information and the fictitous usage profile is summarized in Table 8.

Table 8. Added information to the use case model on the environment level.

	Actor		**Variants**		**Use Cases**
<0.95>	Subscriber (7)	<0.55>	Subscr$_{lo}$ (5)	<0.7>	Normal Call
				<0.05>	Activate CFU
				<0.05>	Deactivate CFU
				<0.2>	CFU Call
		<0.45>	Subscr$_{hi}$ (2)	<0.6>	Normal Call
				<0.05>	Activate CFU
				<0.05>	Deactivate CFU
				<0.3>	CFU Call
<0.05>	Operator (1)		none	<0.7>	Read Markings
				<0.3>	Reset Markings

Structure level. On the structure level we have to add profile information to the scenarios. To each branch in the HMSC flow (see Figure 4), a probability is attached. The resulting use case with profile information for the scenarios in the use case Normal Call with Charging is presented in Figure 17.

Event level. On the event level, probabilities are attached to each choice in the model. For example, the alternative operator introduces a non-deterministic choice between two or more alternatives. If we decorate the alternative operator, as shown in Figure 18, with probabilities, we can

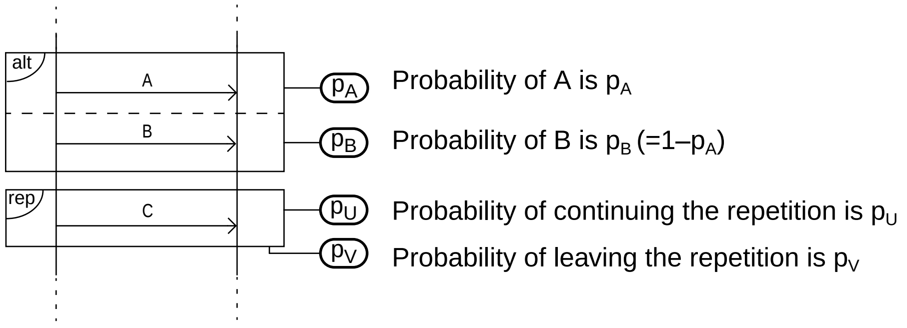

alt
A
B
rep
C
p_A
p_B
p_U
p_V
Probability of A is p_A
Probability of B is p_B (=1–p_A)
Probability of continuing the repetition is p_U
Probability of leaving the repetition is p_V

Figure 18. *Operators extended with probabilities.*

draw random numbers to decide which alternative is chosen during test case generation. This way we can construct stochastic semantics for each operator that determines how to generate scenarios. The scenarios are then used as test cases.

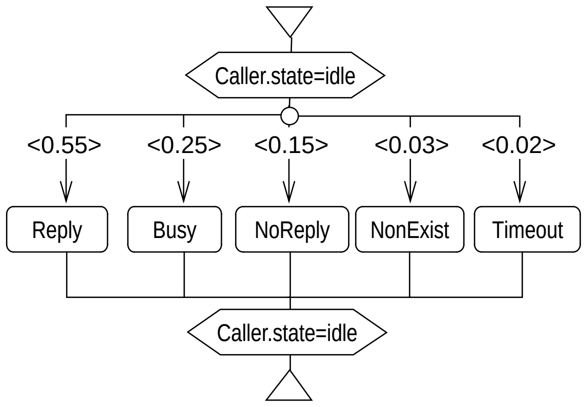


Figure 17. *MSC for the use case NCC extended with profile information.*

For each actor-service combination an *invocation probability* can be given to reflect how likely it is for this service to be selected for a given actor. As many users may interact simultaneously with the system, we need to specify the likelihood of the next event belonging to the same service invocation. To model this, we introduce for each possible actor-service combination a *continuation probability* that states the probability that the next event is within the same service invocation.

5.3 Discussion

Both the presented approaches to integration of use case modelling and statistical usage testing have shown to be feasible in a pilot study conducted on a PBX system, but there is a need for further investigation. This paper presents some examples from the pilot study, together with some preliminary observations and findings, but extensive case studies are needed to evaluate the two approaches, before deciding on which approach is preferable in which situation.

The *transformation* approach has the advantage of being based on two relatively mature disciplines which ends up in two models, each specifically defined for its purpose. The transformation rules support the modelling activities. The major disadvantage of this approach is the necessity of dealing with two different conceptual frameworks and, and having to perform the transformation between the models.

The *extension* approach has the advantage of not needing a second model, as it, instead, extends the modelling power of use cases with stochastic semantics. We can stick to the same conceptual framework for our requirements level usage model and decorate the model with probabilities of usage to enable reliability certification. Thus, the event based semantics does not need to be transformed into state based semantics.

6. Conclusions and Further Research

There remains many challenges in both requirements engineering and requirements-based system validation, and we believe that usage modelling will play an important role in both disciplines. Reliability certification is still in the cradle, but quantification of software quality will be a competition factor in the future, hence usage-based testing and a user perspective on the software are important. Use cases provide the means for communication between users and developers in the requirements phase, and usage-based testing allows for user evaluation prior to releasing the software.

The presented work addresses conceptual issues related to usage modelling and its application to both requirements engineering and testing. The objective is to integrate use case modelling and usage-based testing to form a comprehensive user-centred framework that enables both functional requirements specification and reliability certification. The presented results include a conceptual study of use case modelling and statistical usage testing based on the state hierarchy model. Both modelling techniques rely on similar concepts, which suggests that an integration is feasible. Two integration approaches are identified. The first approach aims at establishing transformations rules that allow use case models to be transformed into state hierarchy models. The second approach aims at extending use case models with stochastic semantics to allow test case generation directly from use case models. We believe that both approaches are feasible, but further research is needed to fully assess the virtues of each approach. Some of the areas where further research is needed are:

- Validation of rules for transformation of event-based use case models to state-based test models.
- Stochastic semantics of use case models for test case generation.
- Introduction of time in stochastic use case models.
- Empirical studies of an integrated usage modelling approach.

Acknowledgements

The authors would like to thank our colleague, Anders Wesslén for giving valuable comments on the paper. This work was financially supported by the National Board for Industrial and Technical Development, (NUTEK), Sweden, Reference P10505-1.

References

Booch, G., *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publ., 1994.

Bubenko, J. A., “Challenges in Requirements Engineering”, *Proceedings of Second International Symposium on Requirements Engineering*, pp. 160-164, York, UK, March 1995.

Buhr, R. J. A., Casselman, R. S., *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., *Object-Oriented Development - The Fusion Method*, Prentice Hall, 1994.

Dardenne, A., van Lamsweerde, A., Fickas, S., “Goal-directed Requirements Acquisition”, *Science of Computer Programming*, Vol. 20, pp. 3-50, 1993.

Fowler, M., Scott, K., *UML Distilled: Applying the Standard Object Modelling Language*, Addison Wesley, 1997.

Graham, D. R., “Testing”, *Software Engineering Encyclopedia*, (J. J. Marciniak, ed.), Vol. 2, pp. 1330–1353, John Wiley & Sons, 1994.

ITU-T *Recommendation Z.120*, Message Sequence Chart (MSC), International Telecommunication Union, 1996.

Jacobson, I., Christerson, M., Jonsson, P., Övergarrd, G., *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992.

Jacobson, I., “A Growing Consensus on Use Cases”, *Journal of Object-Oriented Programming*, pp. 15-19, March-April 1995.

Jarke, M., Pohl, K., Haumer, P., Weidenhaupt, K., Dubois, E., Heymans, P., Rolland, C., Ben Achour, C., Cauvet, C., Ralyté, J., Sutcliffe, A., Maiden, N. A. M., Minocha, S., “Scenario Use in European Software Organisations - Results from Site Visits and Questionnaires”, Report of ESPRIT Project CREWS, no. 97-10, 1997.
Available via e-mail: crewsrep@informatik.rwt-aachen.de.

Karlsson, J., Wohlin, C., Regnell, B., “An Evaluation of Methods for Prioritizing Software Requirements”, *Information And Software Technology*, (39) 14-15, pp. 939-947, 1998.

Kimbler, K., Wohlin, C., “A Statistical Approach to Feature Interaction”, *In Proceedings of TINA’95*, pp. 219-230, Melbourne, Australia, March 1995.

Loucopoulos, P., Karakostas, V., *System Requirements Engineering*, McGraw-Hill, UK, 1995.

Mills, H. D., Dyer, M., Linger, R. C., “Cleanroom Software Engineering”, *IEEE Software*, pp. 19-24, September 1987.

Musa, J. D., “Operational Profiles in Software Reliability Engineering”, *IEEE Software*, pp. 14-32, March 1993.

Pohl, K., “The Three Dimensions of Requirements Engineering”, *Proceedings of 5th International Conference on Advanced Information Systems Engineering*, pp. 275-292, Springer-Verlag, 1993.

Potts, C., Takahashi, K., Anton, A., “Inquiry-Based Requirements Analysis”, *IEEE Software*, pp. 21-32, March 1994.

Regnell, B., Kimbler, K., Wesslén, A, “Improving the Use Case Driven Approach to Requirements Engineering”, *Proceedings of Second International Symposium on Requirements Engineering*, pp. 40-47, IEEE Computer Society Press, March, 1995.

Regnell, B., *Hierarchical Use Case Modelling for Requirements Engineering*, Technical Report 120, Dept. of Communication Systems, Lund University, Tech. Lic. dissertation, 1996.

Regnell, B., Andersson, M., Bergstrand, J., “A Hierarchical Use Case Model with Graphical Representation”, *Proceedings of International Symposium and Workshop on Engineering Computer-Based Systems*, pp. 270-277, IEEE Computer Society Press, March, 1996.

Rolland, C., Ben Achour, C., Cauvet, C., Ralyté, J., Sutcliffe, A., Maiden, N., Jarke, M., Haumer, P., Pohl, K., Dubois, E., Heymans, P., “A Proposal for a Scenario Classification Framework”, *Requirements Engineering Journal*, 3:1, 1998.

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

Rumbaugh, J.,“Getting Started - Using Use Cases to Capture Requirements”, *Journal of Object-Oriented Programming*, pp. 12-23, June 1994.

Runeson, P., Wohlin, C., “Usage Modelling: The Basis for Statistical Quality Control”, *Proceedings 10th Annual Software Reliability Symposium*, Denver, Colorado, pp. 77–84, 1992.

Runeson, P., Wohlin, C., “Statistical Usage Testing for Software Reliability Control”, *Informatica*, Vol. 19, No. 2, pp. 195-207, 1995.

Runeson, P., Wesslén, A., Brantestam, J., Sjöstedt, S., “Statistical Usage Testing using SDL”, *SDL´95 with MSC in CASE*, pp. 323-336, edited by R. Braek and A. Sarma, Elsevier Science B. V., 1995.

Runeson, P., Wohlin, C., “A Dynamic Usage Modelling Approach to Software Reliability Engineering”, In *Models for Estimation of Software Faults and Failures in Inspection and Test*, pp. 119–146, PhD thesis, Department of Communication Systems, Lund University, Lund, Sweden, 1998.

Selic, B., Gullekson, G., Ward, P. T., *Real-Time Object-Oriented Modeling*. Wiley & Sons, 1994.

Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P., “Scenario Usage in System Development: A Report on Current Practice”, *IEEE Software*, March, 1998.

Wesslén, A., Wohlin, C., “Modelling and Generation of Software Usage”, *Proceedings Fifth International Conference on Software Quality*, pp. 147-159, Austin, Texas, USA, 1995.

Whittaker, J. A., Thomason, M. G., “A Markov Chain Model for Statistical Software Testing”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 10, pp. 812–824, 1994.

Wohlin, C., Runeson, P., “Certification of Software Components”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 494-499, 1994.

Wohlin, C., Regnell, B., Wesslén, A., Cosmo, H., “User-Centred Software Engineering - A Comprehensive View of Software Development”, *Proceedings of Nordic Seminar on Dependable Computing Systems*, Denmark, August 1994.

Paper IV:

Derivation of an Integrated Operational Profile and Use Case Model

Per Runeson and Björn Regnell

Published in Proceedings of ISSRE'98, *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, November 1998.

Abstract

Requirements engineering and software reliability engineering both involve model building related to the usage of the intended system; requirements models and test case models respectively are built. Use case modelling for requirements engineering and operational profile testing for software reliability engineering are techniques which are evolving into software engineering practice. In this paper, approaches towards integration of the use case model and the operational profile model are proposed. By integrating the derivation of the models, effort may be saved in both development and maintenance of software artifacts. Two integration approaches are presented, transformation and extension. It is concluded that the use case model structure can be transformed into an operational profile model adding the profile information. As a next step, the use case model can be extended to include the information necessary for the operational profile. Through both approaches, modelling and maintenance effort as well as risks for inconsistencies can be reduced. A positive spin-off effect is that quantitative information on usage frequencies is available in the requirements, enabling planning and prioritizing based on that information.

1. Introduction

Bringing the software development under control is a common goal for requirements engineering (RE) and software reliability engineering (SRE). Requirements engineers strive for capturing as much as possible of the requirements on the software, before it is being built. Thereby late and expensive changes are reduced. Software reliability engineers strive for quantifying and improving the quality of the software, in particular the reliability. Thereby disappointing experiences on insufficient operational reliability, with belonging costs are reduced.

However, the strive for reduced costs by taking problems upfront is not the only common denominator for RE and SRE. In both areas a large share of the job is collecting information and building models, both of which take the user's viewpoint. There are opportunities for coordinating tasks for RE and SRE, which leads to reduction of total modelling effort.

Use cases are means for requirements engineering to capture the requirements on a system [1, 2]. Use cases define usage scenarios for different users of the system, thereby defining the external requirements on system capabilities.

The operational profile is an external user-oriented test model which specifies the intended usage of the system in terms of events and their invocation probabilities [3, 4]. A similar approach is presented as statistical usage testing in the Cleanroom methodology [5, 6, 7]. Test cases are generated from the operational profile, thus enabling estimation of the operational software reliability already during system test.

Positive results are reported on the application of both methods. Use cases and scenarios have gained acceptance, both in research and industry, for their ability to support analysis, documentation and validation of requirements [8, 9, 10, 11]. Operational profile testing is reported to save effort during system test and to reduce the operational failures [4]. In this paper we present an integrated approach which provides both use cases for the requirements specification and the operational profile for testing from the same information collection and modelling. A use case based requirements specification has a structure very similar to an operational profile model. Many concepts can be mapped upon each others. We propose a mapping scheme for transformation of a requirements model into a test model. This approach can be further elaborated towards making a single integrated model that fulfils both purposes: requirements and test model. In both cases, effort can be saved in the derivation of the model and in the second approach, also in the maintenance of the model during development and in future maintenance of the system.

The paper is structured as follows. Chapter 2 gives a brief overview of the use case based requirements and Chapter 3 provides and introduction to operational profile testing. In Chapter 4 two approaches to integration are presented and evaluated, the *transformation* and *extension* approaches. A summary is given in Chapter 5.

A subset of a telephone system, a Private Branch eXchange (PBX), which is used in an educational environment, is used throughout the paper as an example [12]. The PBX is a commercial switch with its control processor “short-circuited” and connected to a UNIX workstation running the control software. The PBX provides basic telephony to the connected subscribers, see Figure 1. In its basic version, its only feature is to make a call from one caller to a callee. During a project course, the students implement extended services to the control software, such as charging, and call forwarding.

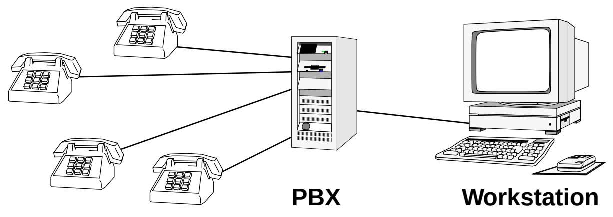


Figure 1. *Example PBX system*

2. Use Case Based Requirements

2.1 Background

The elicitation, analysis and documentation of requirements on software systems is a crucial and non-trivial task. The concepts of use cases and scenarios has gained wide-spread acceptance in object-oriented methods [1, 13, 14] and in the field of requirements engineering [15, 16, 17]. A strong motivation for applying use cases is their support for the modelling of functional requirements in a way that is understandable to users and customers. This ability is embodied in the main idea behind use case modelling, which is to elicit and document requirements by defining and discussing specific contexts of system usage as they are anticipated by the users.

2.2 Method

In [18, 19] a method for use case modelling is presented. This method is an extension of the use case modelling part of the OOSE approach [1]. The major activities are summarized below:

- Elicit actors and their goals.
- Define use cases based on the actors and their goals.
- Elicit scenarios for each use case.
- Describe the events in the scenarios.

The main concepts in the these activities are briefly described in the following. Users can be of different types, called **actors**. A user is thus an instance of an actor. An actor represents a set of users that have some common characteristics with respect to why and how they use the target system. In the PBX system, two actors can be identified, *subscriber* and *operator.* Each actor has a set of **goals**, reflecting such common characteristics. Goals are objectives that users have when using the services of a target system. Thus, goals are used to categorize users into actors. Figure 2a shows the goals of the two actors in the PBX example.

A **service** is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. Figure 2b includes the services of our PBX example system.

A **use case** represents a usage situation where one or more services of the target system are used by one or more actors with the aim to accomplish one or more goals. Figure 2c shows use cases of the PBX example, and their relation to actors, goals, and services.

A **scenario**[1] is a specific realisation of a use case described as a finite sequence of events. A scenario may either model a successful or an unsuccessful accomplishment of one or more goals. A use case may cover an unlimited number of scenarios as it may include alternatives and repeti-

1. Some authors use the terms scenarios and use cases as synonyms, but here we distinguish between them, to differentiate between type level and instance level.

a) Actors and goals.

Actors	Goals
Subscriber	GS1 To achieve communication with another subscriber
	GS2 To cease communication with another subscriber
	GS3 To achieve reachability at another destination
	GS4 To cease reachability at another destination
Operator	GO1 To maintain markings information representing call duration
	GO2 To achieve a printout of the number of markings for each subscriber
	GO3 To achieve a resetting of the number of markings for each subscriber

b) Services.

Service	Description
NCC	Normal Call with Charging
CFU	Call Forward Unconditional
RMR	Read Markings and Reset

c) Uses cases.

Use Cases	Actors	Goals	Services
Normal Call	Subscriber	GS1, GS2, GO1	NCC
Activate CFU	Subscriber	GS3	CFU
Deactivate CFU	Subscriber	GS4	CFU
Invoke CFU	Subscriber	GS1, GS2, GS3	CFU
Read Markings	Operator	GO2	RMR
Reset Markings	Operator	GO3	RMR

Figure 2. *Actors, goals, services and use cases in the PBX example.*

tions. A scenario, however, is a specific and bound realisation of a use case, with all choices determined to one specific path. Figure 3a shows a number of scenarios identified for the use case *normal call*.

When describing the events of each scenario it is useful to define a **data model**, **messages** of the system and **system actions**. The latter mean system intrinsic events which are atomic in the sense that there is no com-

a) Scenarios for use case “Normal Call”.

Scenario	Description
Reply	Call to idle subscriber that replies
Busy Subscriber	Call to busy subscriber
No Reply	Call to idle subscriber that does not reply
Non-Existent	Call to non-existent subscriber
Timeout	Subscriber waits too long after offhook

b) Data model.

```
toneType =
 (dialTone, ringSignal, ringTone,
  busyTone, errorTone, infoTone);
maxNumberOfSubscribers: Natural;
SubscriberType: record (
    state: (idle, busy, off);
    telNumber: TelNbrType;
    CFU_active: Boolean;
    CFU_number: TelNbrType;
    markings: Natural;
    talkingTo: SubscriberType);
SubscriberSet:
    Set (1..maxNumberOfSubscribers)
    Of
       SubscriberType;
```

c) Messages.

Message	Description
offHook	From Subscriber when lifting receiver
onHook	From Subscriber when hanging up the receiver
number(telNbrType)	From Subscriber when dialling a Number
activateCFU(telNbrType)	From Subscriber when activating CFU
deactivateCFU	From Subscriber when deactivating CFU
startTone(toneType)	To Subscriber when a tone is given
stopTone(toneType)	To Subscriber when a tone is stopped
readMarkings	From Operator when issuing a reading of markings
resetMarkings	From Operator when issuing a reset of markings
markings(markingListType)	To Operator when reporting the markings

Figure 3. *Data, messages and a subset of the scenarios for the PBX example.*

munication between the target system and the users that participate in the use case. The data and messages for the PBX example are shown in Figure 3b and 3c respectively, and the system actions can be seen in the example scenario described in Table 1.

Table 1. Use Case “Normal Call” Scenario “Reply”

Actor(s)	Caller, Callee: Subscriber	
Pre-condition(s): Caller.state = idle		
	Events	Constraints
1.	Caller to System: offHook	Caller.state = busy
2.	System to Caller: startTone(dialTone)	
3.	Caller to System: number(X)	
4.	System to Caller: stopTone(dialTone)	

5.	System action: number analysis	Callee.telNumber = X;
		Callee.state = idle

6.	System to Callee: startTone(ringSignal) System to Caller: startTone(ringTone)	Callee.state = busy
7.	Callee to System: offhook	
8.	System to Callee: stopTone(ringSignal)	
	System to Caller: stopTone(dialTone)	

9.	System action: connect Caller and Callee	Caller.talkingTo = Callee;
		Callee.talkingTo = Caller;

10.	Callee to System: onHook	
	Caller to System: onHook	

11.	System action: disconnects Caller and Callee	Caller.talkingTo = nil;
		Calle.talkingTo = nil;

12.	System action: updates marking info	Caller.markings is incremented based on the time between 9–11;
Post-condition(s): Caller.state = idle; Callee.state = idle;		

2.3 Results

Use cases and scenarios are gaining increased attention in requirements engineering research and industrial application. In [8], an industrial case study reports that use cases facilitates all stakeholders to participate in the requirements process with good results in revealing defects. In [11], a survey of industrial application of use cases and scenarios identifies a number of perceived benefits.

Potential problems applying use cases and scenarios are to choose the appropriate level of detail and degree of completeness for the use cases.

In [10], it is concluded that there is an industrial need to base system tests on use cases and scenarios. The studied projects, however, rarely satisfied this demand, as most projects lacked a systematic approach for defining test cases based on use cases. In the this paper we investigate if such an approach can be based on operational profile testing.

3. Operational Profiles

3.1 Background

Software reliability is defined as “the probability for failure-free operation of a program for a specified time under a specified set of operating conditions” [20]. The reliability is hence not only depending on the number of faults in the software, but on how the software is used, hence exposing the faults as failures. In order to predict the operational reliability during test, the test cases executed has to resemble the operational usage, thus constituting a sample from the future operation. For this purpose a model of the future operation is built: the operational profile.

The operational profile consists of the structure of the usage and the probabilities for different uses. Examples of structural elements of the operational profile are different types of customers, different types of users, modes in which the system can operate, functions and operations which can be invoked by the user. The probabilities for activation of users, services etc. are connected to each structural element, constituting the operational profile.

From the operational profile, test cases are selected and executed. The test cases constitute a sample from the future operation, and hence the failure data collected during the test can be used for predicting the operational software reliability. In addition, the tests based on the operational profile has appeared to be efficient with respect to improved software reliability during testing [3, 4].

3.2 Method

In [4] a method for deriving an operational profile is presented. The method is summarized below in five steps:

- Develop a customer type list.
- Develop a user type list.
- List system modes.
- Develop a functional profile.
- Convert the functional profile to an operational profile.

For a more thorough description of the steps, refer to [4]. Here we illustrate the steps with the PBX example.

The **customer type list** collects the different types of customers that *acquire* the system. A customer type represents a set of customers which utilize the system in a similar manner. The example PBX can be sold to small companies with 4-8 employees and medium-sized companies with 9–50 employees. Hence there are two customer types, *small* and *medium*.

The **user type list** collects the different types of users that *use* the system. This list is not necessarily the same as the customer type list. For larger systems, it is generally not the same. In our example case, there are two user types, *subscribers* that make calls via the PBX and *operators* which maintains the charging information.

System modes represent a set of functions or operations that are grouped together in a way that is suitable for the application. The system modes need not to be orthogonal to each others; a function or operation can be member of different system modes. Criteria for defining system modes can be according to [4]: Relatedness of functions/operations to larger task, significant environmental conditions, operational architectural structure, criticality, customer or user, and user experience.

In our example, we define three system modes: *low-traffic subscriber use*, *high-traffic subscriber use* and *operator use*. The low-traffic and high-traffic system modes represent the same functions and operations but with different frequencies of use.

The first step in defining the **functional profile** is to create a function list. Functions are defined from the user's perspective and do not involve architectural or design factors. In the sample system there are four functions for the *low-traffic* and *high-traffic subscriber* system modes: *normal* *call*, and three functions for *call forward unconditional* (CFU), *activate*, *deactivate* and *invoke*. For the *operator* system mode there are two functions: *Read markings* and *reset markings*. The function list can be modified by taking environmental variables into account, for example different traffic levels. However this is already taken into account in the system modes in this example.

Now the profile is attached to the functions. We choose an implicit form with **key input variables** for each function. The variables decide the variants of the functions. The different values of the variables are called **levels**. For the normal call function, the key input variable is the input, or lack of input from the callee. Different variants of the function are invoked depending on which input, or lack of input the called party gives. Five levels of the variable are identified: *reply, busy subscriber, no reply, non-existent* and *timeout*. Similarly the variables and levels can be identified for the other functions.

The functions and key input variables can be presented as an event tree for the user subscriber under the system mode low traffic, see Figure 4. Similarly is defined for the operator functions and the other system modes.

The **operational profile** segments for the customer types, user types and system modes are presented in Figure 5a. Note that all combinations are not possible between user types and system modes.

Finally the functions are mapped onto **operations**. For example, the *normal call* function can be built up by four operations, *number analysis*, *connect subscribers*, *disconnect subscribers* and *update markings*. It can be noted that operations may be involved in performing different functions. For example, the operations that are involved in the *normal call* function are also involved in the *invoke CFU*. The difference between the two functions is that in the *normal call*, the *number analysis* operation returns the identity of the callee, while in the *invoke CFU*, the *number analysis* returns the identity of the subscriber to which the CFU is directed. The resulting operations are listed in Figure 5b. The four subscriber functions result in six unique operations, of which four are shared by multiple functions.

There are key input variables for the operations as well, but this is not elaborated here. Nor is the functional profile mapped onto the operations, since the focus in this paper is on the structural parts, not on the profile parts.

3.3 Results

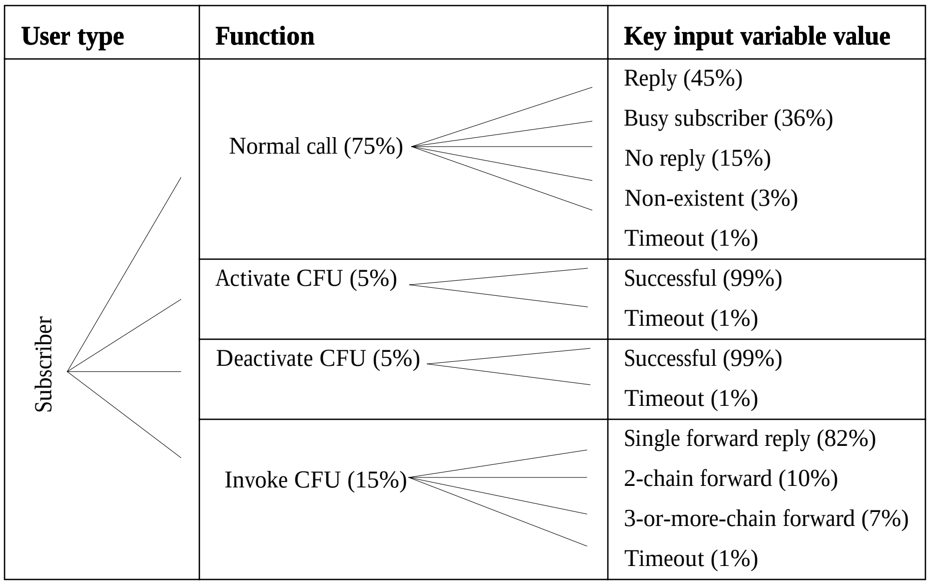
Application of operational profile testing is reported in [3, 4]. Firstly it provides measures of the software reliability which can be used for project and market planning. This is a substantial step forwards in the strive for having the software engineering process under control.

Secondly, it is reported to save effort during test. In [3] it is reported that test costs are saved with up to 56%. These figures has of course to be taken as is: a case study. Even if there was no saving at all in the general case, the operational profile testing moves effort upfront to earlier phases of the project which supports the approach of solving problems earlier, and thus cheaper.

Thirdly, the user perspective taken in the operational profile definition is beneficial since it affects the system design effort as well [21]. By looking at the system from outside-in, the customer and user viewpoints are taken. This helps prioritizing what are the requirements on the system, that fulfil the customer and user needs.

Problems encountered in operational profile testing are related to the usage information. For newly developed systems, the usage information may not be known in detail. However for evolving systems, usage information is available from earlier releases in operation.

Figure 4. *Event tree for user "subscriber" under system mode "low traffic".*



User type	Function	Key input variable value
Subscriber	Normal call (75%)	Reply (45%)
		Busy subscriber (36%)
		No reply (15%)
		Non-existent (3%)
		Timeout (1%)
	Activate CFU (5%)	Successful (99%)
		Timeout (1%)
	Deactivate CFU (5%)	Successful (99%)
		Timeout (1%)
	Invoke CFU (15%)	Single forward reply (82%)
		2-chain forward (10%)
		3-or-more-chain forward (7%)
		Timeout (1%)

a) Profile segments.

Customer type	User type	System mode
Small (75%)	Subscriber (95%)	Low-traffic subscriber (55%)
Medium (25%)	Operator (5%)	High-traffic subscriber (45%)
		Operator use (100%)

b) Functions and their mapping onto operations.

Function	Operations
Normal call	Number Analysis Connect Subscribers Disconnect Subscribers Update Markings
Activate CFU	CFU Activation
Deactivate CFU	CFU Deactivation
Invoke CFU	Number Analysis Connect Subscribers Disconnect Subscribers Update Markings

Figure 5. *Operational profile parts.*

These results from applying operational profile testing can be combined with use case modelling, giving coordination benefits, which is further elaborated in next chapter.

4. Derivation of Operational Profile from Use Cases

A use case model and operational profile model has very much information in common. It can be observed in the examples in Chapter 2 and Chapter 3, that the structures of the models are very much the same. The operational profile model primarily adds the profile information i.e. the probabilities for use of different system capabilities. However the models originate from different disciplines and different terminology is used.

In the search for possibilities for integrating models from the two domains, we elaborate two different approaches, the *transformation* approach and the *extension* approach, see Figure 6.

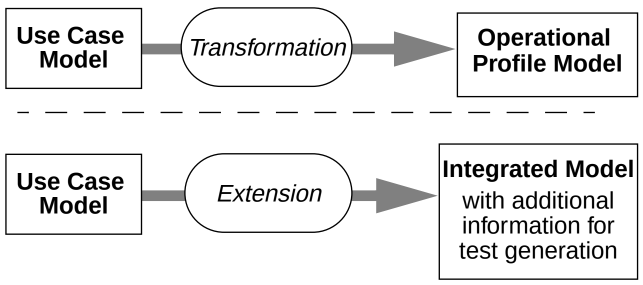

Use Case Model
Transformation
Operational Profile Model
Use Case Model
Extension
Integrated Model
with additional information for test generation

Figure 6. *Two ways of integrating use case models with operational profile models.*

4.1 Transformation approach

The transformation approach takes the information in the use case model and builds the operational profile model based on that information and additional information from other sources. In order to find the common parts in the two models, the concepts in the two domains are elaborated below.

The requirements model does not include quantitative aspects on usage frequencies, while the operational profile model does. This information has hence to be derived in addition to the use case information.

The **customer types** in the operational profile model address the customers, which acquire the system, and the probability information for different types of customers. If high-level goals are included in the use case model, customers might be found among the **actors**, otherwise others sources have to be consulted.

The **user types** in the operational profile generally correspond to the **actors** in the use case model. Both concepts refer to categories of objects, either human users or other systems, that interact with the system.

The **system mode** is a set of functions and operations that relate to each other. This corresponds in part to the **service** concept of the use case model. However the system mode may also cover profiling information, and that part cannot be found in the use case model.

The usage of **functions** in the operational profile model is described by **use cases** in the requirements model. The functions as well as the use cases are defined from the user's perspective, and do not consider architectural and design factors. Variants of functions in the operational profile are distinguished by **key input variables.** Depending on the value of the variable (the level), different alternatives of the function are chosen. The levels of the key input variables have their counterparts in the **scenarios** of the use case model, which are variants of a use case. A specific combination of key input variables gives a specific scenario, i.e. a realisation of a use case.

In the operational profile model, **operations** are tasks accomplished by the system in order to deliver a function to the user. A similar concept in the use case model are the **system actions**.

The mapping between concepts in an operational profile model and a use case model are summarized in Table 2.

Table 2. Mapping of terminology [a]

Operational profile model		Use case model
Customer type	≈	Actor
User type	=	Actor
System mode	≈	Service
Function	≈	Use case
Key input variable	≈	Scenario
Operation	=	System action

a. "≈" means that the concepts partially model the same information,
"=" means that they actually model the same information.

4.2 Example on the transformation approach

In this section we present an example on the transformation approach. Given the mapping scheme from Section 4.1, an operational profile model is derived from the use case model presented in Chapter 2. As in Chapter 3, we follow the steps proposed in [4], but now the information is collected from the use case model.

First the **customer type** list is derived. Since there are no high-level actors in the use case model, there is no information on customer types, so this information has to be collected elsewhere. Generally the different customer types are defined as a requirement written in plain text. In our example we have *small* (4–8 employees) and *medium-sized* (9–50 employees) enterprises as two different customer types.

The **user type** list can be derived directly from the list of **actors** in Figure 2a. The actors *subscriber* and *operator* are selected as user types.

Next, the **systems modes** are elaborated. The information can be taken in parts form the **services** in Figure 2b: *normal call with charging (NCC), call forward unconditional (CFU)* and *read markings and reset (RMR)*. In addition to this, we add the profile information. Among the subscribers, there are two different groups with respect to their usage frequencies Thus we add, *low-traffic* and *high-traffic* to the system modes for the subscriber services, defining in total four: *low-traffic CFU, low-traffic NCC, high-traffic CFU* and *high-traffic NCC*. The *operator use* is defined as a fifth system mode.

The list of **functions** is derived directly from the list of **use cases**, see Figure 2c. There are in total six functions, *normal call, activate CFU, deactivate CFU, invoke CFU, read markings* and *reset markings.*

The **key input variables** which define variants of the functions can be derived from the **scenarios** of the use case, see Figure 3a. For the normal call function, the key input variable is the input or lack of input given as response to the call. The levels of the variable are: *reply, busy subscriber, no reply, non-existent* and *timeout*.

The usage frequency information in the **functional profile** cannot be found in the use case model, but has to be derived from other sources, for example interviews or measurements on existing systems.

The functions are mapped onto **operations** which together constitute the function to the user. The **system actions** in the use case model constitute candidates for operations, see Figure 1. In the normal call use case, scenario reply, there are four system actions involved, *number analysis,* *connect subscribers, disconnect subscribers* and *update markings*. We define operations with the same names for the operational profile.

Finally, the functional profile is mapped onto the operations, defining the **operational profile**. A summary of the transformation is given in Table 3:

Table 3. Summary of information origin in the transformation approach.

Operational profile part	Information in use case model
Customer type	Partly
User type	Yes
System mode	Partly
Function	Yes
Key input variable	Yes
Functional profile	No
Operation	Yes
Operational profile	No

It can be concluded that the main part of the structural information for the transformation can be taken from the use case model. In some cases, the interpretations of the concepts differ, but the core of the information is valid. The profile information, i.e. probabilities and frequencies have to be taken from elsewhere. It can also be noticed that the operational model transformed from the use case model can be slightly different from the one that was derived directly. In our example, the system modes are not exactly the same.

4.3 Extension approach

An alternative to transforming the use case model into an operational profile model is to extend the use case model with profile information. The resulting integrated model should then fit both the purpose of requirements specification and test specification. Given the two sets of models with different concepts and terminology presented in Chapter 2 and Chapter 3, an integrated model should be developed.

Basically the extension means that user frequencies and probabilities are added to the use case model. Additionally, the concepts of the operational profile model, which have no correspondence in the use case model or has a different interpretation must be added or tailored. The extension approach also means that concepts which are close to each other in the two models, have to be defined to be exactly the same. Below it is identified which concepts can be used exactly as is, and which have to be tailored or added to the integrated model. It shall not be seen as a complete proposal for an integrated model, but a basis for further definition work.

The **customer type** is not generally a part of the operational profile model, although sometimes modelled as **actors**, e.g. when defining high-level goals for the system. The information is beneficial for the understanding of the customer needs and is often provided in text in the requirements specification, hence it is added to the integrated model.

User types and **actors** are substantially the same in the two models, and can be integrated.

The **system modes** in the operational profile and the **service** concept in the use case model are integrated.

The usage of the **functions** in the operational profile are described by **use cases** and can be integrated. The functional profile information is added to the model.

Key input variables define alternatives in functions, while **scenarios** define alternative realizations of use cases. Given that usage of functions and use cases are integrated, these two can be integrated as well.

The **operations** of the operational profile model are the tasks that build up a function which correspond to **system actions** in the use case model. The operational profile is added to the model.

Table 4. Integrated use case and operational profile model

Operational profile model		Use case model	Integrated model
Customer type	≈	Actor	Customer actor
User type	=	Actor	Actor
System mode	≈	Service	Service
Function	≈	Use case	Use case
Key input variable	≈	Scenario	Scenario
Functional profile		–	Use case profile
Operation	=	System action	Operation
Operational profile		–	Operational profile

4.4 Analysis of approaches

The use case model and the operational profile model have similar structures and much information in common. It can hence be concluded that any form of coordination is beneficial with respect to effort spent on modelling, focus on test and testability already in the requirements and quantification of requirements for priority or other purposes.

The models have different terminology and notations used. However, also the purposes of the models differ. The purpose of the use case model is to provide understanding of customer needs while the purpose of the test model is to generate test cases to verify that requirements are fulfilled. The notations and terminology used can be mapped between the two domains, as elaborated above. The purpose conflict is more serious in the *transformation* approach than in the *extension* approach.

The two approaches to integration, *transformation* of the use case model into an operational profile model and *extension* of the use case model into an integrated model, have their advantages and disadvantages. The approaches are compared below, but in order to make a better evaluation, empirical studies have to be conducted.

The *extension* approach has advantages over the *transformation* approach with respect to effort spent to derive the model. A single model is derived instead of two, which saves effort. Furthermore from a maintenance perspective, the extension approach is to prefer, since it reduces risks for inconsistencies between models. The consistency within a single model is easier to maintain. Furthermore the incentives for maintaining the integrated model are better since it is used both in requirements and test, not only for a single purpose.

The *transformation* approach has advantages over the *extension* approach when studying the fitness for purposes and level of detail. The extension approach always include compromises between needs for the different purposes. The purpose of the use case model is to bring best possible understanding of user and customer needs. The purpose of the test model is to generate test cases representative for the future operation. The main conflict concerns functions with low usage frequency, but high importance for the customer. These functions are given more attention in the requirements model and less attention in the test model.

The requirements model may be less detailed than the test model. On the other hand, if the extended model evolves during the development, the level of detail can be set for the requirements purposes in the beginning of the development, and evolve into more details, according to the test model needs.

The advantages and disadvantages are summarized in Table 5.

Table 5. Comparison between integration approaches.

	Transformation	**Extension**
Information	+ same information for both purposes	+ same information for both purposes
Structure	+ same structure for both purposes	+ same structure for both purposes
Derivation	– two models to derive	+ single model to derive
Maintenance	– two models to maintain – risk for inconsistencies	+ single model to maintain + consistency within model
Fit for purpose	+ each model tailored for its purpose	– the model is a compromise between purposes
Level of detail	+ each model at appropriate level of detail	– the model is a compromise between levels of detail

5. Summary

This paper presents work which aims at integrating requirements engineering (RE) and software reliability engineering (SRE). Two different approaches are presented, the *transformation* and the *extension* approaches.

There are many advantages of an integration between RE and SRE. The SRE aspects are addressed earlier in the development cycle, enabling proactive rather than reactive actions. The quantitative aspects of the SRE usage modelling may help in the prioritizing of requirements. The connection between the requirements defined and their verification and validation is made closer. Finally, by utilizing the same sources for information, effort may be saved in the modelling tasks.

The transformation of a use case model to an operational profile model is one step towards integration of the two domains, which is evaluated in this paper. The model extension is another step which is supposed to save more effort, in particular in maintenance of software artifacts, but on the other hand may involve compromises between RE and SRE purposes for usage modelling. The proposed approaches to integration have to be evaluated empirically in order to get more understanding on which approach is most preferable in different situations.

Acknowledgement

The authors would like to thank our colleague, Anders Wesslén for giving valuable comments on the paper. This work was financially supported by the National Board for Industrial and Technical Development, (NUTEK), Sweden, grant 1K1P-97-09690.

References

1 Jacobson, I. et al. *Object-Oriented Software Engineering – A Use Case Driven Approach*, Addison-Wesley Publishing Company and ACM Press, 1992.

2 Regnell, B., *Hierarchical Use Case Modelling for Requirements Engineering*, Technical Report 120, Dept. of Communication Systems, Lund University, Tech. Lic. dissertation, 1996.

3 Musa, J. D., "Operational Profiles in Software Reliability Engineering", *IEEE Software*, pp. 14–32, March 1993.

4 Lyu, M. R., (ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, 1995.

5 Linger, R., "Cleanroom Process Model", *IEEE Software*, pp. 50–58, March 1994.

6 Whittaker, J. A. and Thomason, M. G., "A Markov Chain Model for Statistical Software Testing", *IEEE Transactions on Software Engineering*, Vol. 20, No. 10, pp. 812–824, 1994.

7 Runeson, P. and Wohlin, C., "Statistical Usage Testing for Software Reliability Control" *Informatica*, Vol. 19, No. 2, pp. 195–207, 1995.

8 Gough, P., Fodemski, F., Higgins, S. and Ray, S., "Scenarios – an Industrial Case Study and Hypermedia Enhancements", *IEEE Second International Symposium on Requirements Engineering*, York, UK, pp. 10–17, March 1995.

9 Regnell, B. and Davidsson, Å., "From Requirements to Design with Use Cases – Experiences from Industrial Pilot Projects", *Proceedings of the Third International Workshop on Requirements Engineering: Foundations of Software Quality* (REFSQ'97), pp. 205–222, Presses Universitaires de Namur, 1997.

10 Jarke, M., et. al., "Scenario Use in European Software Organisations - Results from Site Visits and Questionnaires", Report of ESPRIT Project

CREWS, no. 97-10, 1997.
Available via e-mail: crewsrep@informatik.rwt-aachen.de.

11 Weidenhaupt, K., Pohl, K., Jarke, M. and Haumer, P., “Scenario Usage in System Development: A Report on Current Practice”, Report of ESPRIT Project CREWS, no. 97-16, 1997.
Available via e-mail: crewsrep@informatik.rwt-aachen.de.

12 Wohlin, C., “Meeting the Challenge of Large-Scale Software Development in an Educational Environment”, *Proceedings 10th Conference on Software Engineering Education and Training,* Virginia Beach, Virginia, USA, pp. 40–52, 1997.

13 Rumbaugh, J., et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

14 Booch, G., *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publ., 1994.

15 Potts, C., Takahashi, K. and Anton, A., “Inquiry-Based Requirements Analysis”, *IEEE Software*, pp. 21–32, March 1994.

16 Hsia, P., Samuel, J., Gao J. and Kung, D., “Formal Approach to Scenario Analysis”, *IEEE Software*, pp. 33–41, March 1994.

17 Rolland, C., et. al. “A Proposal for a Scenario Classification Framework”, Report of ESPRIT Project CREWS, no. 96-01, 1996.
Available via e-mail: crewsrep@informatik.rwt-aachen.de.

18 Regnell, B., Kimbler, K. and Wesslén, A., “Improving the Use Case Driven Approach to Requirements Engineering”, *IEEE Second International Symposium on Requirements Engineering*, York, UK, pp. 40–47, March 1995.

19 Regnell, B., Andersson, M. and Bergstrand, J., “A Hierarchical Use Case Model with Graphical Representation”, *IEEE International Symposium and Workshop on Engineering of Computer-Based Systems*, Germany, pp. 270–277, March 1996.

20 “IEEE Standard Glossary of Software Engineering Terminology” IEEE 610.12, 1990.

21 Chruscielski, K. and Tian J., “An Operational Profile for the Cartridge Support Software”, *Proceedings 8th International Symposium on Software Reliability Engineering*, Albuquerque, New Mexico, USA, pp. 203–212, 1997.

Paper V:

Are the Perspectives Really Different?

- Further Experimentation on Scenario-Based Reading of Requirements

Björn Regnell, Per Runeson, Thomas Thelin

Technical Report CODEN: LUTEDX(TETS-7172) / 1-40 / 1999 & local 4,
Dept. of Communication Systems, Lund University, 1999.

Abstract

Perspective-Based Reading (PBR) is a scenario-based inspection technique where several inspectors read a document from different perspectives (e.g. user, designer, tester). The reading is made according to a special scenario, specific for each perspective. The basic assumption behind PBR is that the perspectives find different defects and a combination of several perspectives detects more defects compared to the same amount of reading with a single perspective. This paper presents a study which analyses the differences in the perspectives. The study is a partial replication of previous studies. It is conducted in an academic environment using MSc and PhD students as subjects. Each perspective applies a specific modelling technique: use case modelling for the user perspective, equivalence partitioning for the tester perspective and structured analysis for the design perspective. A total of 30 MSc students were divided into 3 groups, giving 10 subjects per perspective. A control group of 9 PhD students used a checklist reading technique. The analysis results show that (1) there is no significant difference between the three perspectives in terms of defect detection rate and number of defects found per hour, (2) there is no significant difference in the defect coverage of the three perspectives, and (3) PhD students with a checklist approach find significantly more defects per hour and have a significantly higher detection rate than MSc students

with a PBR approach. The results suggest that a combination of multiple perspectives may not give higher coverage of the defects compared to single-perspective reading. It is also indicated that individual abilities and motivation are more important than the reading technique used.

1. Introduction

The validation of requirements documents is often done manually, as requirements documents normally include informal representations of what is required of an intended software system. A commonly used technique for manual validation of software documents is inspections, proposed by Fagan (1971). Inspections can be carried out in different ways and used throughout the software development process for (1) understanding, (2) finding defects, and (3) as a basis for making decisions. Inspections are used to find defects early in the development process, and have shown to be cost effective (e.g. Doolan, 1992).

A central part of the inspection process is the *defect detection* carried out by an individual reviewer reading the document and recording defects (a part of preparation, see Humphrey, 1989). Three common techniques for defect detection are Ad Hoc, Checklist and Scenario-based reading (Porter, 1995). Ad Hoc detection denotes an unstructured technique, providing no guidance and the reviewers detect defects based on their personal knowledge and experience. The checklist detection technique provides a list of issues and questions, capturing the knowledge of previous inspections, helping the reviewers to focus their reading.

In the scenario-based approach, different reviewers have different responsibilities and are guided in their reading by specific scenarios which aim at constructing a model, instead of just passive reading. A scenario[1] here denotes a script or procedure that the reviewer should follow. Two variants of scenario-based reading have been proposed: Defect-Based Reading (Porter, 1995) and Perspective-Based Reading (Basili, 1996). The former (subsequently denoted DBR) concentrates on specific defect

1. There is considerable risk for terminology confusion here, as the term *scenario* also is used within requirements engineering to denote a sequence of events involved in an envisaged usage situation of the system under development. A *use case* is often said to cover a set of related (system usage) scenarios. In scenario-based reading, however, the term scenario is a meta-level concept, denoting a procedure that a reader of a document should follow during inspection.

classes, while the latter (subsequently denoted PBR) focuses on the points of view of the users of a document.

Another part of the inspection process is the *compilation of defects* into a consolidated defect list where all individual reviewers' defect lists are combined. This step may include the removal of false positives (reported defects that were not considered to be actual defects) as well as the detection of new defects. This step is often done in a structured *inspection meeting* where the *team* of reviewers participate. The effectiveness of a team meeting has been questioned and studied empirically by Votta (1993) and Johnson (1998).

This paper describes research on scenario-based reading with a PBR approach. The research method is empirical and includes a formal factorial experiment in an academic environment. The presented experiment is a partial replication of previous experiments in the area and focuses on refined hypotheses regarding the differences between the perspectives in PBR. The paper concentrates on defect detection by *individual reviewers*, while the team meeting aspects are left out for future analysis.

The structure of the paper is as follows. Section 2 gives an overview of related work by summarising results from previously conducted experiments in requirements inspections with a scenario-based approach. Section 3 includes the problem statement motivating the presented work. In Section 4, the experiment plan is described including a discussion on threats to the validity of the study, and Section 5 reports on the operation of the experiment. The results of the analysis is given in Section 6, and Section 7 includes an interpretation of the results. Section 8 provides conclusions and an outline of future work.

2. Related Work

The existing literature on empirical software engineering includes a number of studies related to inspections, where formal experimentation has shown to be a relevant research strategy. The experiment presented in this paper relates to previous experiments on inspections with a scenario-based approach. The findings of a number of experiments on scenario-based inspection of requirements documents are summarized below.

- The *Maryland-95* study (Porter, 1995) compared DBR with Ad Hoc and Checklist in an academic environment. The experiment was run twice with 24 subjects in each run. The requirements docu-

ments used were a water level monitoring system (WLMS, 24 pages) and an automobile cruise control system (CRUISE, 31 pages).
Result 1: DBR reviewers have significantly higher defect detection rates than either Ad Hoc or Checklist reviewers.
Result 2: DBR reviewers have significantly higher detection rates for those defects that the scenarios were designed to uncover, while all three methods have similar detection rates for other defects.
Result 3: Checklist reviewers do *not* have significantly higher detection rates than Ad Hoc reviewers.
Result 4: Collection meetings produce *no* net improvement in the detection rate – meeting gains are offset by meeting losses.

- The *NASA* study (Basili, 1996) compared PBR with Ad Hoc[1] in an industrial environment. The experiment consisted of a pilot study with 12 subjects and a second main run with 13 subjects. There were two groups of requirements documents used; general requirements documents: an automatic teller machine (ATM, 17 pages), a parking garage control system (PG, 16 pages); and two flight dynamics requirements documents (27 pages each).
Result 1: Individuals applying PBR to general documents have significantly higher detection rates compared to Ad Hoc.
Result 2: Individuals applying PBR to NASA-specific documents do *not* have significantly higher detection rates compared to Ad Hoc.
Result 3: Simulated teams applying PBR to general documents have significantly higher detection rates compared to Ad Hoc.
Result 4: Simulated teams applying PBR to NASA-specific documents have significantly higher detection rates compared to Ad Hoc.
Result 5: Reviewers with more experience do *not* have higher detection rates.

- The *Kaiserslautern* study (Ciolkowski, 1997) compared PBR with Ad Hoc in an academic environment using the ATM and PG documents from the NASA study. The experiment consisted of two runs

1. Basili (1996) refers to this reading technique as the "usual technique" at NASA SEL described in *Recommended Approach to Software Development*, Revision 3, available at (1 Feb. 1999) http://sel.gsfc.nasa.gov/doc-st/docs/81-305.pdf
Requirements inspections according to this document do not prescribe any specific reading technique.

with 25 and 26 subjects respectively.
Result 1: Individuals applying PBR to general documents have significantly higher detection rates compared to Ad Hoc.
Result 2: Simulated teams applying PBR to general documents have significantly higher detection rates compared to Ad Hoc.
Result 3: The detection rates of five different defect classes are *not* significantly different among the perspectives.

- The *Bari* study (Fusaro, 1997) compared DBR with Ad Hoc and Checklist in an academic environment using the WLMS and CRUISE documents from the Maryland-95 study. The experiment had one run with 30 subjects.
  Result 1: DBR did *not* have significantly higher defect detection rates than either Ad Hoc or Checklist.
  Result 2: DBR reviewers did *not* have significantly higher detection rates for those defects that the scenarios were designed to uncover, while all three methods had similar detection rates for other defects.
  Result 3: Checklist reviewers did *not* have significantly higher detection rates than Ad Hoc reviewers.
  Result 4: Collection meetings produced *no* net improvement in the detection rate – meeting gains where offset by meeting losses.

- The *Trondheim* study (Sørumgård, 1997) compared the NASA study version of PBR with a modified version of PBR (below denoted PBR2) where reviewers were given more instructions on how to apply perspective-based reading. The study was conducted in an academical environment using the ATM and PG documents from the NASA study. The experiment consisted of one run with 48 subjects.
  Result 1: PBR2 reviewers did *not* have significantly higher defect detection rates than PBR.
  Result 2: Individuals applying PBR2 reviewed significantly longer time compared to those who applied PBR.
  Result 3: Individuals applying PBR2 suggested significantly fewer potential defects compared to those who applied PBR.
  Result 4: Individuals applying PBR2 had significantly lower productivity and efficiency than those who applied PBR.

- The *Strathclyde* study (Miller, 1998) compared DBR with Checklist in an academic environment using the WLMS and CRUISE documents from the Maryland study. The experiment consisted of one

run with 50 subjects.
Result 1: In the WLMS document, DBR did *not* have significantly higher defect detection rates than Checklist.
Result 2: In the CRUISE document, DBR had significantly higher defect detection rates than Checklist.
Result 3: Collection meetings produced *no* net improvement in the detection rate – meeting gains were offset by meeting losses.

- The *Linköping* study (Sandahl, 1998) compared DBR with Checklist in an academic environment using the WLMS and CRUISE documents from the Maryland study. More defects were added to the list of total defects. The experiment consisted of one run with 24 subjects.
  Result 1: DBR reviewers did *not* have significantly higher defect detection rates than Checklist reviewers.
  Result 2: DBR reviewers did *not* have significantly higher detection rates than Checklist reviewers.

- The *Maryland-98* study (Shull, 1998) compared PBR with Ad Hoc in an academical environment using the ATM and PG documents from the Maryland study. The experiment consisted of one run with 66 subjects.
  Result 1: PBR reviewers had significantly higher defect detection rates than Ad Hoc reviewers.
  Result 2: Individuals with high experience applying PBR did *not* have significantly[1] higher defect detection rates compared to Ad Hoc.
  Result 3: Individuals with medium experience applying PBR had significantly higher defect detection rates compared to Ad Hoc.
  Result 4: Individuals with low experience applying PBR had significantly higher defect detection rates compared to Ad Hoc.
  Result 5: Individuals applying PBR had significantly lower productivity compared to those who applied Ad Hoc.

- The *Lucent* study (Porter, 1998) replicated the Maryland-95 study in an industrial environment using 18 professional developers at Lucent Technologies. The replication was successful and completely corroborated the results from the Maryland-95 study.

1. Results 2-4 of the Maryland-98 study apply a significance level of 0.10, while 0.05 is the chosen significance level in all other results.

The Maryland-95, NASA, Kaiserslautern, Maryland-95, and Lucent studies indicate that a scenario-based approach give higher detection rate. The Bari, Strathclyde, and Linköping studies could, however, not corroborate this with statistically significant results, which motivates further studies to increase the understanding of scenario-based reading.

Many of the studies concluded that real team meetings were ineffective in terms of defect detection. (There may of course be other good reasons for conducting team meetings apart from defect detection, such as consensus building, competence sharing, and decision making.)

The efficiency (detected defects per time unit) of scenario-based reading has, however, not been studied, as the major dependent variable is detection rate which does not take into account the time spent on reading.

The series of studies summarised above is a salient example of how independently conducted replications of experiments in different environments are used to strengthen or question theories and hypotheses. In Software Engineering, replicated experiments are relatively uncommon, compared to other scientific disciplines. This paper follows the empirical tradition, by building on the understanding gained by previous studies and testing refined hypotheses in a continued replication effort.

The study presented here is subsequently denoted the *Lund* study. The Lund study is a partial replication of the NASA study, and is based on a lab package (Basili, 1998) provided by the University of Maryland in order to support empirical investigations of scenario-based reading. The problem statement motivating the Lund study is given in the subsequent section.

3. Background and Motivation

The previous studies, summarised in Section 2, have mainly concentrated on comparing scenario-based reading with checklist and Ad Hoc techniques in terms of defect detection rates. The objective of the Lund study is, however, to investigate the basic assumption behind scenario-based reading, that the different perspectives find different defects. Another interest is the efficiency of the different perspectives in terms of defects detected per hour. This section describes PBR and states the research questions motivating the presented work.

3.1 Perspective-Based Reading Theory

The basic assumptions behind PBR are (1) that a reviewer with a specific focus performs better than a reviewer with the responsibility to detect all types of defects, and (2) that different foci can be designed so that their union yields full coverage of the inspected document. Another important assumption is that it is easier to detect defects if a reviewer works structured and reads actively. Also, by utilising the document under inspection in a way that it will be used in subsequent phases, the reviewer is able to detect defects that otherwise are not detected until later on.

The idea behind reading a document from different perspectives and use of scenarios is to gain a better detection coverage of defects in a software document. Basili et al. (1996) use three different perspectives which originate from roles in the software development process. Three different phases of special importance are design, verification and operation; the roles are *designer*, *tester* and *user*[1].

Model building is a central part of PBR. The models used in the study of PBR stem from well-known techniques used in the different software phases. Designers utilise *structured analysis*, tester apply *equivalence partitioning* and users construct *use cases*. All these models are adapted to be used as inspection techniques. Procedures for all three perspectives are provided by Basili et. al. (1998).

Structured analysis aims at creating a high-level and a low-level data flow diagram from requirements, where internal software processes are modelled. After a number of iterative steps between the low-level and high-level data flows, the diagrams show the relationship between the system and its environment as well as the internal processing in the system. During the inspection a number of questions are answered to validate that the design fulfils all requirements documented and that no requirements are missing. As a result of the structured analysis based inspection, it is expected that some defects are detected in the requirements phase which otherwise not would have been detected until the design phase.

Equivalence partition testing aims at creating a set of test cases using a requirements document as input. The test cases are based on partitions of input data into classes equivalent in terms of how the system functions. Test cases close to the edge values and the middle of a range of values are

1. The user perspective is in Basili (1996) represented by a technical writer who will, based on the requirements document, prepare a user manual.

selected. During the work with finding test cases different questions are answered by the reviewer to ensure that the requirements are correct and that no requirements are missing. As a result of the equivalence partitioning based inspection, it is expected that some defects are detected in the requirements phase which otherwise not would have been detected until the testing phase.

Use case modelling aims at documenting the functionality of the product as seen by the user. This is carried out in three iterative activities. First, the person inspecting as a user tries to find the participants involved from which actors are identified. Second, the functions of the product is identified from which use cases are modelled. Third, the actors are matched to the corresponding use cases in which they participate. Q uestions are answered during the creation of use cases. As a result of the use case based inspection, it is expected that some defects are detected in the requirements phase which otherwise not would have been detected until operation.

By combining these three perspectives, the expected results are *better defect coverage* and *earlier defect detection*.

3.2 Research Questions

In this experiment the focus is to investigate whether perspective-based reading works as it is supposed to. Two questions are addressed:

1. Do the perspectives detect different defects?

2. Is one perspective superior to another?

Addressing the latter first, there are two aspects of superiority that are to be addressed, namely *effectiveness*, i.e. how high fraction of the existing defects are found (detection rate), and *efficiency*, i.e. how many defects are found per time unit.

The perspectives proposed by Basili (1996) are designer, tester and user. The users are important stakeholders in the software development process, and especially when the requirements are elicited, analysed and documented. The user role in PBR is focused on detecting defects at a high abstraction level related to system usage, while the designer is focused on internal structures and the tester is focused on verification. It may not be unlikely that the user perspective is more efficient in finding missing requirements, as the user often is a major source of the requirements. Furthermore, the user perspective is focused on the employment of the product, while the designer and the tester are focused on the development of the product and are thereby closer to the code. The user may very well be the most capable perspective for detecting requirements defects. With these suppositions in mind, it would be interesting to investigate the capabilities of the user perspective in comparison with other perspectives.

Previous studies have mainly concentrated on the effectiveness in terms of detection rate. From a software engineering viewpoint it is important also to assess the efficiency (e.g. in terms of detected defects per time unit), as this factor is important for a practitioner's decision to introduce a new reading technique. The specific project and application domain constraints then can, together with estimations of how much effort is needed, be a basis for a trade-off between quality and cost.

One main purpose of PBR is that the perspectives detect different kinds of defects in order to minimise the overlap among the reviewers. Hence, a natural question is whether reviewers do find different defects or not. If they detect the same defects, the overlap is not minimised and PBR does not work as it was meant to. If all perspectives find the same kinds of defects it may be a result of (1) that the scenario-based reading approach is inappropriate, (2) that the perspectives may be insufficiently supported by their accompanying scenarios, or (3) that other perspectives are needed to gain a greater coverage difference. The optimal solution is to use perspectives with no overlap and as high defect detection rate as possible, making PBR highly dependable and effective. The Lund study addresses the overlap by investigating whether the perspectives detect different defects.

Another structured inspection technique is checklist reading. In contrast to PBR, which concentrates on the working procedure, checklist reading is focused on detecting specific kinds of defects that are known from previous inspections to be common or crucial. In this experiment we focus upon PBR, but a group of readers applying a checklist is also included as a comparison. The checklist used in the Lund study is the same as in (Miller, 1998). Due to educational constraints, it was not possible to have the same type of subjects for both PBR and checklist reading. The PBR technique was taught within a Software Engineering course at the Master level and the education goal of the Master students would not be fulfilled if one group just applied checklist reading. Hence, in the Lund study PhD students participated voluntarily as a control group. As a consequence, the subjects in the PBR group consisting of MSc students have different training, experience, motivation and ability compared to the checklist group consisting of PhD students. This fact introduces a confounding factor in the experiment, but the comparison with a checklist is still interesting, as the following research question can be answered:

3. Can less experienced reviewers with PBR outperform more experienced reviewers with a less sophisticated approach, or are personal abilities more important than the technique used?

The Maryland-95 study indicates that checklist inspections are approximately as good as Ad Hoc inspections and the NASA study suggests that PBR inspections are significantly better than Ad Hoc inspections in terms of detection rate, so it is interesting to investigate if this effect can be observed even if the group using the more “primitive” approach of checklist reading is more experienced.

The three questions stated above are the main input to the more detailed planning of the experiment. This plan is described in the subsequent section.

4. Experiment Planning

In this section describes the planning of the reading experiment. The planning includes the definition of dependent and independent variables, hypotheses to be tested in the experiment, experiment design, instrumentation and an analysis of threats to the validity of the experiment. The purpose is to describe and motivate the procedures applied when conducting the experiment in order to enable the reader to validate the reported results. This section is also aimed at supporting further replication of the experiment, which is encouraged to be conducted in industrial as well as academical environments.

The reading experiment is conducted in an academical environment with close relations to industry. The subjects are fourth-year students at the Master’s programmes in Computer Science & Engineering and Electrical Engineering at Lund University and PhD students at the Department of Communication Systems and the Department of Computer Science at the same university. The departments have close cooperation with industry, both in joint research projects and in education matters.

4.1 Variables

The independent variables determine the cases for which the dependent variables are sampled. The purpose is to investigate different reading perspectives and methods, applied to two objects (requirements documents) by different types of subjects (reviewers). The independent variables are hence:

- STYPE. Among the subjects, there are two types: fourth-year Master students (MSc) and PhD students (PhD). 30 Msc students and 9 PhD students took part in the study.
- PERSP. In the PBR reading method used by MSc students, one of three perspectives is applied by each subject: User, Tester or Designer (U, T, D). The control group of PhD students uses the checklist method, and for simplicity this method is treated as yet another value of the PERSP variable: Checklist (C).
- DOC. The inspection objects are two requirements documents one for an automatic teller machine (ATM) and one of a parking garage control system (PG). The ATM document is 17 pages and contains 29 defects. The PG document is 16 pages and contains 30 defects.

The inspection objects are the same as in the University of Maryland lab package (Basili, 1998), and the design and instrumentation is also based on this lab package.

There is one variable that is controlled, in order to guide the allocation of subjects to different perspectives:

- EXPERIENCE. The experience is measured through a questionnaire which covers software engineering experience in general, experience with inspections, and experience with the modelling techniques of the three perspectives, user, tester and designer. The experience is measured for each perspective and each modelling technique on a 5 level ordinal scale (1 = none, 2 = studied in class or from book, 3 = practised in a class project, 4 = used on one project in industry, 5 = used on multiple projects in industry)

The variables in the study are summarized in Table 1 together with brief explanations.

Table 1. Variables.

	Name	**Values**	**Description**
Independent variables	STYPE	{MSc,PhD}	Type of subject
	PERSP	{U,T,D,C}	Perspective (U,T,D) or checklist (C)
	DOC	{ATM,PG}	Inspected requirements document
Controlled Variable	EXPERIENCE	Ordinal	Experience with user, tester, design and checklist techniques, measured on a five-level ordinal scale.
Dependent Variables	TIME	Integer	Time in minutes spent on finding defects
	DEF	Integer	Number of defects found after removal of false positives
	EFF	60*DEF/TIME	Number of defects found per hour
	RATE	DEF/TOT	The fraction of found defects per total number of defects
	FOUND	Integer	How many in a certain perspective PERSP in {U,T,D} have found defect DEFID for a given document DOC

The problem statements on PBR effectiveness, efficiency and distribution over perspectives imply that the following set of dependent variables are measured.

- TIME. The time spent by each inspector in individual preparation is recorded by all subjects. The time unit used is minutes.
- DEF. The number of defects found by each inspector is recorded, excluding false positives. The false positives are removed by the experimenters, in order to ensure that all defect candidates are treated equally.
- EFF. The fault finding efficiency, i.e. the number of defects found per hour, calculated as (DEF*60)/TIME.
- RATE. The fault finding effectiveness, i. e. the fraction of found defects by total number of defects (also called detection rate) is calculated as DEF divided by the total number of known defects contained in the inspected documents.
- FOUND. The number of reviewers, belonging to a certain perspective, which have found a certain defect in a specific document is recorded. This variable is used for analysing fault finding distributions for different perspectives.

4.2 Hypotheses

Perspective-Based Reading is assumed to provide more efficient inspections, as different reviewers take different perspectives making the defect overlap smaller (Basili, 1996). The objective of the study is to empirically test if these assumptions are true. In consequence, hypotheses related to performance of different perspectives and the checklist method are stated. The checklist method is treated as a fourth perspective with respect to efficiency and rate. The hypotheses are elaborated below, and the formal definitions of the hypotheses, as well as their null hypotheses counterparts are presented in Table 2. The three hypotheses address efficiency, effectiveness and distribution over perspectives.

- H_{EFF}. The perspectives are assumed to have different finding efficiency, i.e. the number of defects found per hour of inspection is different for the various perspectives.
- H_{RATE}. The perspectives are assumed to have different effectiveness or detection rates, i.e. the fraction of defects identified is different for the various perspectives.
- H_{FOUND}. The perspectives are assumed to find different defects, i.e. the distribution over defects found are not the same for different perspectives.

Table 2. Hypotheses.

Hypothesis	**Definition**	**Null**	**Definition**
H_{EFF}	The average defect detection efficiency $\overline{EFF}_{PERSP}$ is *different* for at least one perspective {U,D,T,C}	$H_{0,EFF}$	The average defect detection efficiency $\overline{EFF}_{PERSP}$ is *equal* for all perspectives {U,D,T,C}
H_{RATE}	The average defect detection rate $\overline{RATE}_{PERSP}$ is *different* for at least one perspective {U,D,T,C}	$H_{0,RATE}$	The average defect detection rate $\overline{RATE}_{PERSP}$ is *equal* for all perspectives {U,D,T,C}
H_{FOUND}	The distribution of FOUND is *different* for at least one perspective {U,D,T}	$H_{0,FOUND}$	The distribution of FOUND is *equal* for all perspectives {U,D,T}

4.3 Design

To test these hypotheses an experiment with a factorial design (Montgomery, 1997) is used with two factors (PERSP and DOC). The design is summarized in Table 3. The experiment varies the four perspectives over two documents and two types of subjects. The documents are independent of the other two factors, while there is a constraint in the allocation of subject types to perspectives. Due to educational limitations, the STYPE factor could not be used in randomisation as MSc students could only be allocated to either the User, Designer or Tester perspective, and hence, the PhD subject types are allocated to the Checklist perspective.

Table 3. Experiment design

		PERSP			
		User	**Designer**	**Tester**	**Checklist**
DOC	ATM	5 MSc	5 MSc	5 MSc	6 PhD
	PG	5 MSc	5 MSc	5 MSc	3 PhD

The assignment of an individual subject to one of the three PBR perspectives (U, D, T), was conducted based on their reported experience, similar to the NASA study (Basili, 1996). The objective of experience-based perspective assignment is to ensure that each perspective gets a fair distribution of experienced subjects, so that the outcome of the experience is affected by perspective difference rather than experience difference. The experience questionnaire required the subjects to grade their experience with each perspective on a five level ordinal scale. The subjects were then sorted three times, giving a sorted list of subjects for each perspective with the most experienced first. Within the same experience level, the subjects were placed in random order. The subjects were then assigned to perspectives by selecting a subject on top of a perspective list and removing this subject in the other lists before continuing with the next perspective in a round robin fashion starting with a randomly selected perspective, until all subject were assigned a perspective.

Each PhD student was randomly assigned to one of the two documents in a way that allowed team meetings[1] with three participants in each team, hence the unbalanced 6/3 in the number of subjects for DOC.

The instruments for the reading experiment consists of two requirements documents and reporting templates for time and defects. These instruments are taken from the University of Maryland lab package (Basili, 1998) and are reused with minimal changes.

The factorial design described above is analysed with descriptive statistics (bar plots and box plots) and analysis of variance (ANOVA) (Montgomery, 1997) for the hypotheses H_{EFF}, and H_{RATE}.

For the H_{FOUND} hypothesis a Chi-square test (Siegel, 1988) is used together with correlation analysis (Robson, 1993).

4.4 Threats to Validity

The validity of the results achieved in experiments depends on factors in the experiment settings. Different types of validity can be prioritized depending on the goal of the experiment. In this case, threats to four types of validity are analysed (Cook, 1979): Conclusion validity, internal validity, construct validity and external validity.

Conclusion validity concerns the statistical analysis of results and the composition of subjects. In this experiment, well known statistical techniques are applied which are robust to violations of their assumptions. One general threat to conclusion validity is, however, the low number of samples, which may reduce the ability to reveal patterns in the data, and in particular there are few samples for the chi-square test, which is further elaborated in Section 6.2. The subjects consists of two groups, MSc and PhD students, which are homogeneous within the groups but heterogeneous between the groups. This reduces the ability to draw conclusions over the different groups.

Internal validity concerns matters that may affect the independent variable with respect to causality, without the researchers knowledge. There are two threats to the internal validity in this experiment, selection and instrumentation. The experiment was a mandatory part of a software engineering course, thus the selection of subjects is not random, which involves a threat to the validity of the experiment. The requirements doc-

1. Team meetings were held for educational reasons, but the analysis of team meeting results are not included in this investigation.

uments used may also affect the results. The documents are rather fault-prone and additional issues in the documents could be considered as defects. On the other hand, it is preferable to have the same definition of defects as in the previous studies for comparison reasons. Other threats to internal validity are considered small. Each subject was only allocated to a single object and a single treatment, hence there is no threat of maturation in the experiment. The subjects applied different perspectives during inspection, but the difference between perspectives are not large enough to suspect compensatory equalisation of treatments or compensatory rivalry. The subjects were also told that their grading in the course was not depending on their performance in the experiment, only on their serious attendance. There is of course a risk that the subjects lack motivation; they may, for example, consider their participation a waste of time or they may not be motivated to learn the techniques. The teacher in the course in which the experiment is performed, has, however, made a strong effort in motivating the students. It was clearly stated that a serious participation was mandatory for passing the course. It is the teacher's opinion that the students made a very serious attempt in their inspection.

Construct validity concerns generalisation of the experiment result to concept or theory behind the experiment. A major threat to the construct validity is the limitation of allocation of subjects to the different perspectives. Since the MSc students had to be allocated to one of the U, D or T perspective and the PhD students were allocated to the C perspective, the C perspective cannot be compared in general with the U, D and T perspectives. Further, the chosen perspectives or the reading techniques for the perspectives (Basili, 1996) might not be representative or good for scenario-based reading. This limits the scope for the conclusions made to these particular perspectives and techniques. Other threats to the construct validity are considered small. The subjects did not know which hypotheses were stated, and were not involved in any discussion on advantages and disadvantages of PBR, thus they were not able to guess what the expected results were.

External validity concerns generalisation of the experiment result to other environments than the one in which the study is conducted. The largest threat to the external validity is the use of MSc students as subjects. However this threat is reduced by using fourth-year students which are close to finalise their education and start working in industry. The use of PhD students was intended as a control group with respect to external validity, but due to limitations in allocation of subjects to treatments, general conclusions regarding checklist performance in comparison to PBR performance cannot be drawn from this experiment. The setting is intended to resemble a real inspection situation, but the process that the subjects participate in is not part of a real software development project. The assignments are also intended to be realistic, but the documents are rather short, and real software requirements documents may include many more pages. The threats to external validity regarding the settings and assignments are, however, considered limited, as both the inspection process and the documents resemble real cases to a reasonable extent.

It can be concluded that there are threats to the construct, internal and external validity. However, these are almost the same as in the original studies. Hence, as long as the conclusions from the experiment are not drawn outside the limitations of these threats, the results are valid.

5. Experiment Operation

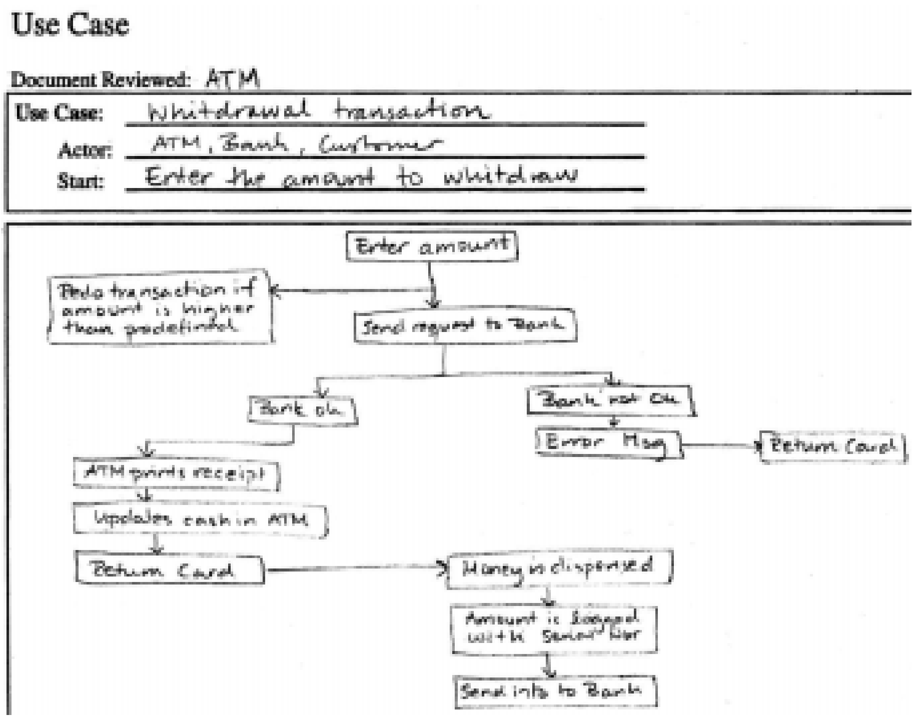
The experiment was run during spring 1998. The MSc students were all given a two hour introductory lecture where an overview of the study was given together with a description of the defect classification. A questionnaire on experience was given and each subject was assigned to a perspective, as described in Section 4.4. The students were informed that the experiment was a compulsory part of the course, but the grading was only based on serious participation in the study and not on the individual performance of the students. The anonymity of the students was guaranteed.

A two hour exercise was held, where the three PBR perspectives were described and illustrated using a requirements document for a video rental system (VRS). During the second hour of the exercise, the subjects were practising their own perspective reading technique for the VRS document, and had the opportunity to ask questions. The data collection forms were also explained and used during the exercise. The perspective-based reading of the VRS document was completed by the students on their own after the classroom hours.

The hand-outs for the experiment, which were handed out during the exercise, included the following instrumentation tools:

1. Defect Classification which describes defect classes to be used in the defect list.

2. Time Recording Log for recording the time spent on reading.


Use Case

Document Reviewed: ATM

Use Case: Whitdrawal transaction

Actor: ATM, Bank, Customer

Start: Enter the amount to whitdraw

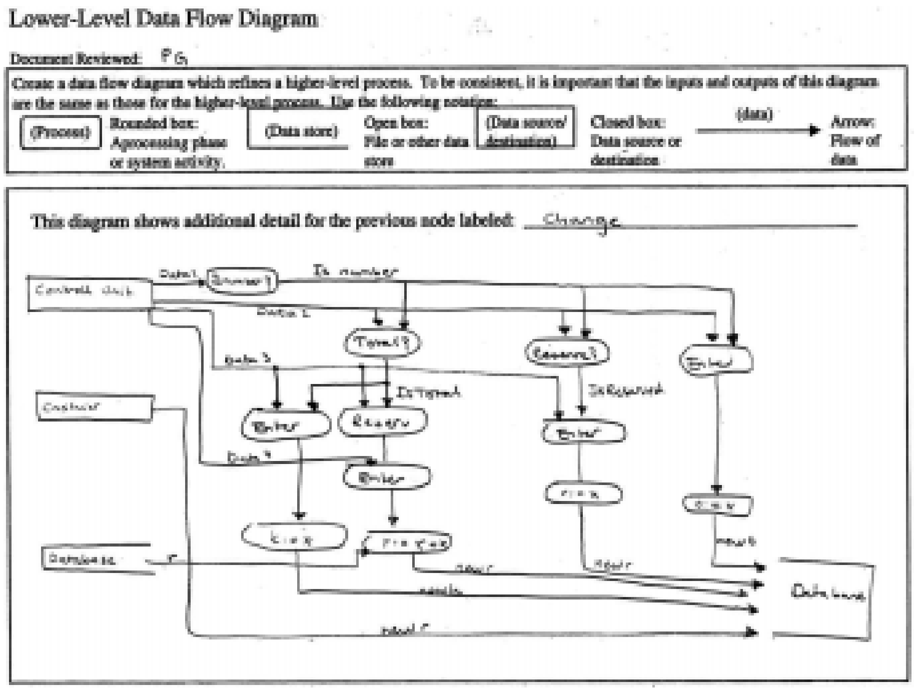
Enter amount
Redo transaction if amount is higher than predefined
Send request to Bank
Bank ok
Bank not ok
Error Msg
Return Card
ATM prints receipt
Updates cash in ATM
Return Card
Money is dispensed
Amount is logged with Serial Nbr
Send info to Bank

Figure 1. *An example of how subjects filled in the form for the user perspective.*

3. Defect List for recording the found defects.
4. Reading Instruction, specific for the user, designer, and tester perspective respectively.
5. Modelling Forms, specific for the user, designer, and tester perspective respectively. (See Figure 1 and 2 for examples of these forms.)
6. The requirements document (either ATM or PG).

The students were instructed not to discuss the ATM or PG documents and the defects that they may find. They were allowed to discuss the PBR perspectives in relation to the VRS document before they started with the actual data collection.

For the control group using the checklist approach, subjects were recruited by inviting PhD students to participate. All PhD students that volunteered were allowed to participate. A one-hour introduction was



Lower-Level Data Flow Diagram

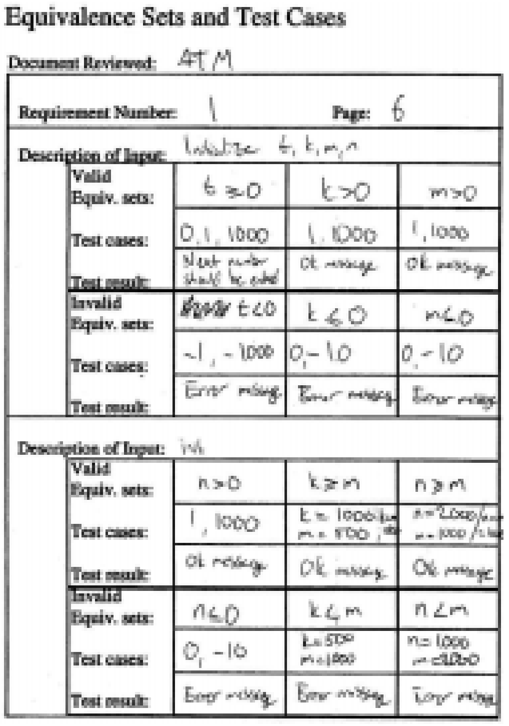
Document Reviewed: PG

Create a data flow diagram which refines a higher-level process. To be consistent, it is important that the inputs and outputs of this diagram are the same as those for the higher-level process. Use the following notation:

(Process) Rounded box: A processing phase or system activity. (Data store) Open box: File or other data store (Data source/ destination) Closed box: Data source or destination (data) Arrow: Flow of data

This diagram shows additional detail for the previous node labeled: Change

Data 1
ID number
Data 2
Data 3
Data 4
Enter
Database



Equivalence Sets and Test Cases

Document Reviewed: ATM

Requirement Number: 1		**Page:** 6	
Description of Input: Initialize t, k, m, n			
Valid Equiv. sets:	t ≥ 0	k > 0	m > 0
Test cases:	0, 1, 1000	1, 1000	1, 1000
Test result:	Next number should be asked	Ok message	Ok message
Invalid Equiv. sets:	~~[illegible]~~ t < 0	k ≤ 0	m ≤ 0
Test cases:	-1, -1000	0, -10	0, -10
Test result:	Error msg.	Error message	Error message
Description of Input: n			
Valid Equiv. sets:	n > 0	k ≥ m	n ≥ m
Test cases:	1, 1000	k = 1000 [illegible] m = 500 [illegible]	n = 2000 [illegible] m = 1000 [illegible]
Test result:	Ok message	Ok message	Ok message
Invalid Equiv. sets:	n ≤ 0	k ≤ m	n < m
Test cases:	0, -10	k = 500 m = 1000	n = 1000 m = 2000
Test result:	Error message	Error message	Error message

Figure 2. *Examples of how subjects filled in the forms for the designer and tester perspectives.*

given for the PhD students where the experiment was described, the defect classes explained and the checklist method was outlined. The checklist published in Miller (1998) was handed out together with defect lists and time recording logs.

6. Data Analysis

This section presents the statistical analysis of the gathered data. The data were collected from the hand-ins from subjects. Each defect in each subject's defect log was compared with the original "correct" defect list provided by the University of Maryland lab package. In a meeting, we discussed each defect and decided if it corresponded to a "correct" defect. If no corresponding "correct" defect was found, the reported defect was considered a false positive. The reported time spent was also collected and the EFF, RATE, and FOUND measures were calculated. The total data set is given in Appendices A and B.

6.1 Individual Performance for Different Perspectives

The means of individual performance in terms of number of defects found per hour (EFF), the fraction of found defects against the total number of defects (RATE), and time spent on reading (TIME), are shown in Figure 3 and Figure 4, grouped by document and perspective. For simplicity in comparisons, the Checklist reading technique is regarded as a fourth value of the PERSP variable.

The means of EFF, and RATE for the Checklist reading technique are higher than the Designer, Tester, and User perspectives for both documents. For RATE the Designer means are slightly higher compared to the User and Tester perspectives for both documents. For the EFF mean the Tester perspective on the PG document is higher than the User and Designer perspectives, while for the ATM document, the Designer perspective has a higher mean. The variation in the data is indicated by the standard deviations, which show that Checklist reading on the ATM document has a larger variation than other groups.

The means of the TIME measures show that, on average per document, the subjects reading with the Checklist technique have spent the shortest time. For the ATM document the TIME measures show that on average, the Designer perspective spent shorter time than the Tester per-

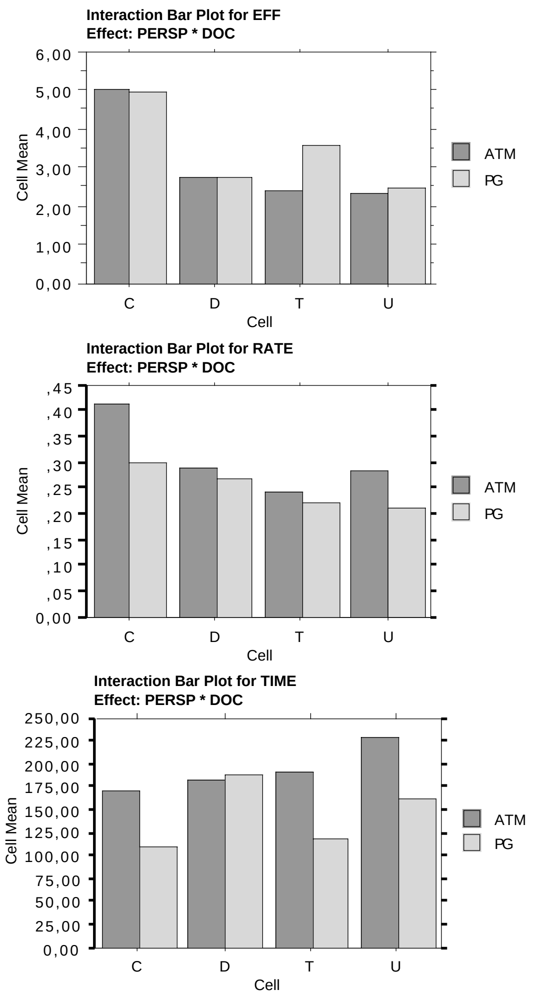


Figure 3. *Bar plots for EFF, RATE and TIME grouped by PERSP and DOC.*

spective which in turn spent shorter time than the User perspective, while for the PG document the Tester perspective spent shorter time followed by the User and Designer perspectives.

Figure 5 shows box-plots[1] of the data split by document and perspective. In general, the ATM document has a larger variation, especially for Checklist reading. There are, however, too few data points per group for

1. The box-plots are drawn with the box width corresponding to the 25th and 75th percentile, with the 50th percentile (the median) marked in the box. The whiskers correspond to the 10th and 90th percentile.

Means Table for EFF
Effect: PERSP * DOC

	Count	Mean	Std. Dev.
C, ATM	6	4,985	3,071
C, PG	3	4,921	,672
D, ATM	5	2,755	,392
D, PG	5	2,760	1,155
T, ATM	5	2,360	,790
T, PG	5	3,593	1,587
U, ATM	5	2,308	1,280
U, PG	5	2,473	,930

Means Table for RATE
Effect: PERSP * DOC

	Count	Mean	Std. Dev.
C, ATM	6	,414	,135
C, PG	3	,300	,067
D, ATM	5	,290	,062
D, PG	5	,267	,058
T, ATM	5	,241	,055
T, PG	5	,220	,060
U, ATM	5	,283	,145
U, PG	5	,213	,090

Means Table for TIME
Effect: PERSP * DOC

	Count	Mean	Std. Dev.
C, ATM	6	170,000	71,875
C, PG	3	110,000	22,539
D, ATM	5	182,400	25,126
D, PG	5	187,000	46,583
T, ATM	5	191,800	68,882
T, PG	5	117,200	25,074
U, ATM	5	229,400	53,379
U, PG	5	161,600	43,102

Figure 4. *Means tables for EFF, RATE and TIME grouped by PERSP and DOC.*

any further interpretation of the box-plots, with respect to outliers and skewdness.

When several dependent variables are measured, the multi-variate analysis of variance (MANOVA) can be used to assess if there exists any statistically significant difference in the total set of means. In Figure 6 the results of four different MANOVA tests regarding three effects are shown: PERSP, DOC, and the combined effect. For the PERSP and DOC effects, the MANOVA tests show significance at the 5% level. For the combined effect the results are not significant, indicating absence of interaction effects.

In order to investigate if the differences in the means of EFF, RATE, and TIME are of any statistical difference, an analysis of variance (ANOVA) is presented in Figure 7 for each variable.

There are three P-values below 5%, indicating that there is a statistical significant difference in:

- EFF by PERSP
- RATE by PERSP
- TIME by DOC

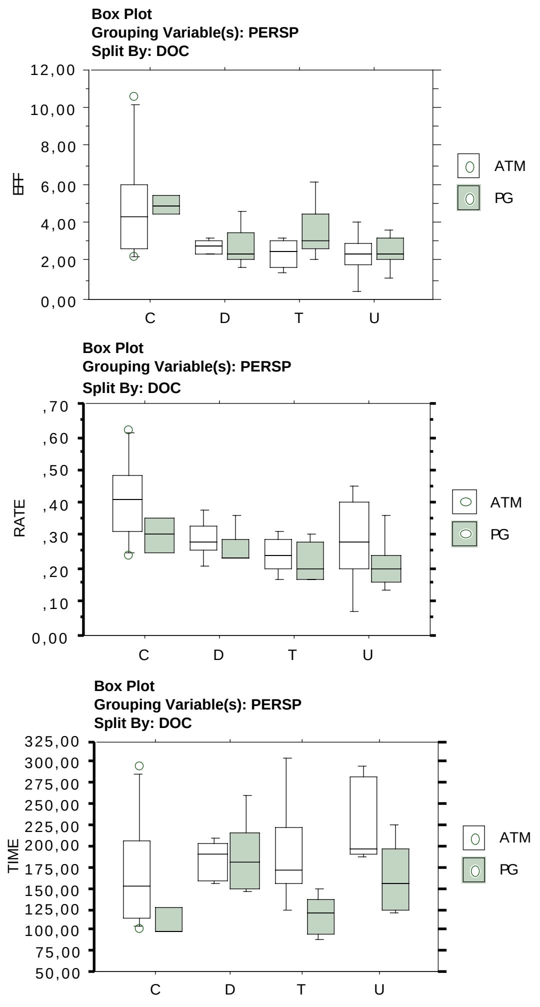


Figure 5. *Box plots for EFF, RATE and TIME split by DOC and PERSP.*

In order to find out which pairs of values of PERSP that give statistically significant difference in the means of EFF and RATE, Fisher's Protected Least Square Difference (PLSD) tests are presented in Figure 8. These post hoc tests may only be used when the ANOVA tests show significant results.

The post-hoc tests in Figure 8 show that there is a statistically significant difference in 6 pairwise combinations:

- EFF: C find more defects per hour than D, T, U
- RATE: C has a higher detection rate than D, T, U

MANOVA Table for PERSP

	Value	F-Value	Num DF	Den DF	P-Value
S	3,000				
M	-,500				
N	13,500				
Wilks' Lambda	,578	1,984	9	71	,0539
Roy's Greatest Root	,445	4,597	3	31	,0090
Hotelling-Lawley Trace	,638	1,963	9	83	,0541
Pillai Trace	,475	1,945	9	93	,0549

MANOVA Table for DOC

	Value	F-Value	Num DF	Den DF	P-Value
S	1,000				
M	,500				
N	13,500				
Wilks' Lambda	,653	5,139	3	29	,0057
Roy's Greatest Root	,532	5,139	3	29	,0057
Hotelling-Lawley Trace	,532	5,139	3	29	,0057
Pillai Trace	,347	5,139	3	29	,0057

MANOVA Table for PERSP * DOC

	Value	F-Value	Num DF	Den DF	P-Value
S	3,000				
M	-,500				
N	13,500				
Wilks' Lambda	,817	,680	9	71	,7244
Roy's Greatest Root	,160	1,650	3	31	,1980
Hotelling-Lawley Trace	,214	,659	9	83	,7435
Pillai Trace	,191	,702	9	93	,7059

Figure 6. *MANOVA tables for the PERSP independent variable and the EFF, RATE and TIME as dependent variables.*

Fisher's PLSD for EFF
Effect: PERSP
Significance Level: 5 %

	Mean Diff.	Crit. Diff	P-Value	
C, D	2,206	1,473	,0046	S
C, T	1,987	1,473	,0098	S
C, U	2,573	1,473	,0012	S
D, T	-,219	1,434	,7575	
D, U	,367	1,434	,6057	
T, U	,586	1,434	,4112	

Fisher's PLSD for RATE
Effect: PERSP
Significance Level: 5 %

	Mean Diff.	Crit. Diff	P-Value	
C, D	,098	,088	,0303	S
C, T	,145	,088	,0020	S
C, U	,128	,088	,0057	S
D, T	,047	,085	,2665	
D, U	,030	,085	,4778	
T, U	-,017	,085	,6824	

Figure 8. *Post-hoc tests for EFF, RATE over PERSP.*

The significant difference of TIME over DOC is investigated by comparing the means of the TIME spent for each document in Figure 4:

- TIME: ATM has more time spent on it than PG

From this analysis regarding individual performance we can conclude that the null hypotheses for EFF and RATE only can be rejected for the Checklist reading technique.

ANOVA Table for EFF

	DF	Sum of Squares	Mean Square	F-Value	P-Value	Lambda	Power
PERSP	3	33,577	11,192	4,530	,0096	13,591	,846
DOC	1	1,055	1,055	,427	,5182	,427	,095
PERSP * DOC	3	2,674	,891	,361	,7817	1,082	,111
Residual	31	76,585	2,470				

ANOVA Table for RATE

	DF	Sum of Squares	Mean Square	F-Value	P-Value	Lambda	Power
PERSP	3	,081	,027	3,061	,0426	9,184	,657
DOC	1	,030	,030	3,451	,0727	3,451	,422
PERSP * DOC	3	,013	,004	,487	,6936	1,462	,135
Residual	31	,272	,009				

ANOVA Table for TIME

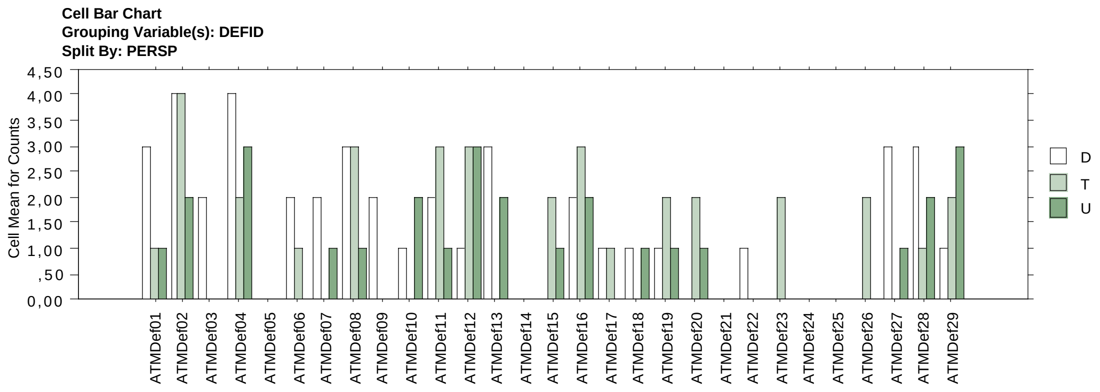
	DF	Sum of Squares	Mean Square	F-Value	P-Value	Lambda	Power
PERSP	3	18264,611	6088,204	2,408	,0860	7,224	,539
DOC	1	23014,612	23014,612	9,103	,0051	9,103	,847
PERSP * DOC	3	9946,295	3315,432	1,311	,2883	3,934	,307
Residual	31	78373,200	2528,168				

Figure 7. *ANOVA tables for EFF, RATE and TIME.*

6.2 Defects found by different perspectives

The hypothesis H_{FOUND} regarding the overlap of the found defects between the perspectives, is studied in this section. Descriptive statistics in the form of bar chart plots are shown in Figure 9. For each document the distribution of number of found defects per perspective is shown. There do not seem to be any particular patterns in the different perspective distributions; the defect findings of each perspective seem similarly spread over the defect space. If there had been large differences in the perspective distributions, the bar plot would presumably have groups of defects where one perspective would have a higher number of findings while the other would have a low number of findings.

In order to compare the distributions of found defects for each perspective and investigate if there is a significant difference between which defects the perspectives find, a contingency table is created for which a Chi Square test is made (Siegel, 1988, pp. 191-194), as shown in Figure 10. The defects that no perspective has found are excluded from



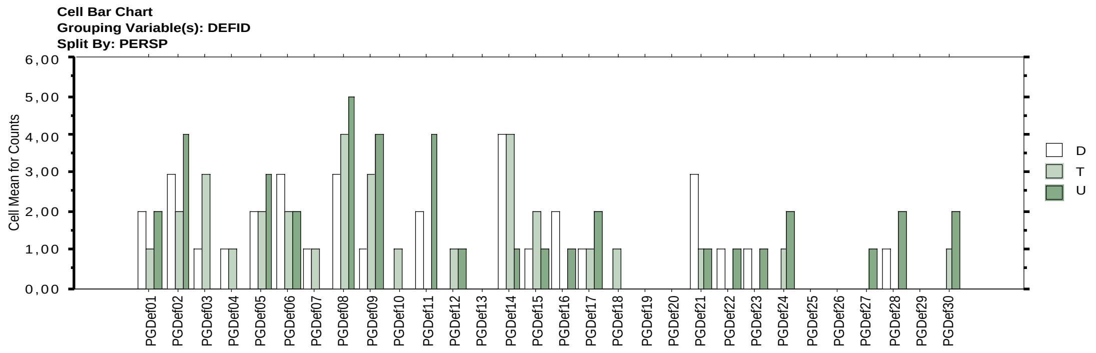


Figure 9. *Bar charts illustrating the distribution of number of reviewers that found each defect.*

the contingency tables, as these cases do not contribute to the testing of differences.

The Chi Square P-values are far from significant, indicating that it is not possible with this test and this particular data set to show a difference in the perspectives' defect finding distributions.

There are rules of thumb regarding when the Chi Square test can be used (Siegel, 1988, pp. 199-200), saying that no more than 20% of the cells should have an expected frequency of less than 5, and no cell should have an expected frequency of less than 1. These rules of thumb are not fulfilled by the data set in this case, but it may be argued that the rules are too conservative and as the expected frequencies in our case are rather evenly distributed, the Chi Square test may still be valid.

Summary Table for DEFID, PERSP
Inclusion criteria: Counts>0 from PG.data

Num. Missing	0
DF	46
Chi Square	33,951
Chi Square P-Value	,9058
G-Squared	•
G-Squared P-Value	•
Contingency Coef.	,494
Cramer's V	,402

Summary Table for DEFID, PERSP
Inclusion criteria: Counts > 0 from ATM.data

Num. Missing	0
DF	46
Chi Square	41,676
Chi Square P-Value	,6538
G-Squared	•
G-Squared P-Value	•
Contingency Coef.	,535
Cramer's V	,448

Observed Frequencies for DEFID, PERSP
Inclusion criteria: Counts>0 from PG.data

	D	T	U	Totals
PGDef01	2	1	2	5
PGDef02	3	2	4	9
PGDef03	1	3	0	4
PGDef04	1	1	0	2
PGDef05	2	2	3	7
PGDef06	3	2	2	7
PGDef07	1	1	0	2
PGDef08	3	4	5	12
PGDef09	1	3	4	8
PGDef10	0	1	0	1
PGDef11	2	0	4	6
PGDef12	0	1	1	2
PGDef14	4	4	1	9
PGDef15	1	2	1	4
PGDef16	2	0	1	3
PGDef17	1	1	2	4
PGDef18	0	1	0	1
PGDef21	3	1	1	5
PGDef22	1	0	1	2
PGDef23	1	0	1	2
PGDef24	0	1	2	3
PGDef27	0	0	1	1
PGDef28	1	0	2	3
PGDef30	0	1	2	3
Totals	33	32	40	105

Observed Frequencies for DEFID, PERSP
Inclusion criteria: Counts > 0 from ATM.data

	D	T	U	Totals
ATMDef01	3	1	1	5
ATMDef02	4	4	2	10
ATMDef03	2	0	0	2
ATMDef04	4	2	3	9
ATMDef06	2	1	0	3
ATMDef07	2	0	1	3
ATMDef08	3	3	1	7
ATMDef09	2	0	0	2
ATMDef10	1	0	2	3
ATMDef11	2	3	1	6
ATMDef12	1	3	3	7
ATMDef13	3	0	2	5
ATMDef15	0	2	1	3
ATMDef16	2	3	2	7
ATMDef17	1	1	0	2
ATMDef18	1	0	1	2
ATMDef19	1	2	1	4
ATMDef20	0	2	1	3
ATMDef22	1	0	0	1
ATMDef23	0	2	0	2
ATMDef26	0	2	0	2
ATMDef27	3	0	1	4
ATMDef28	3	1	2	6
ATMDef29	1	2	3	6
Totals	42	34	28	104

Figure 10. *Chi Square tests and contingency tables for defects found by U,T,D per DOC.*

The Chi Square test does not give a measure of the *degree* of difference. In order to analyse how different (or similar) the perspectives are, a correlation analysis is presented in Figure 11, using the Pearson correlation coefficient (Robson, 1993, pp. 338-340).

Two different correlation analyses are provided for each document, one with all “correct” defects included and one where only those defects are included that were found by at least one reviewer. The latter may be advocated, as we are interested in the differences in the set of defects that are found by each perspective; the defects that no perspective find do not contribute to differences between perspectives.

The P-value indicates if the correlation coefficient is significant, and the confidence intervals presented indicate the range wherein the correlation coefficient is likely to be.

ATM Document

Correlation Analysis

	Correlation	P-Value	95% Lower	95% Upper
User, Tester	,480	,0076	,138	,720
User, Designer	,499	,0052	,162	,732
Tester, Designer	,258	,1789	-,120	,570

29 observations were used in this computation.

Correlation Analysis
Inclusion criteria: User > 0 OR Tester > 0 OR Designer > 0 from ATM-ctable.data

	Correlation	P-Value	95% Lower	95% Upper
User, Tester	,357	,0867	-,054	,665
User, Designer	,352	,0915	-,059	,662
Tester, Designer	,043	,8449	-,367	,439

24 observations were used in this computation.

PG Document

Correlation Analysis

	Correlation	P-Value	95% Lower	95% Upper
User, Tester	,463	,0092	,123	,706
User, Designer	,543	,0016	,228	,756
Tester, Designer	,601	,0003	,307	,790

30 observations were used in this computation.

Correlation Analysis
Inclusion criteria: User > 0 OR Tester > 0 OR Designer > 0 from PG-ctable.data

	Correlation	P-Value	95% Lower	95% Upper
User, Tester	,319	,1300	-,097	,640
User, Designer	,414	,0438	,012	,700
Tester, Designer	,493	,0134	,112	,748

24 observations were used in this computation.

Figure 11. *Correlation analysis of the perspectives for each document.*

The correlation analysis indicates that there are significantly positive correlations between the perspectives, meaning that when one perspective finds a defect it is not unlikely that others also find it. The only correlation coefficient that is far from significant is the Designer-Tester correlation for the ATM document.

Another way of qualitatively analysing the overlap between the perspectives is Venn-diagrams, as used in the NASA study (Basili, 1998, p.151).

For the purpose of comparison we include such diagrams for the Lund study data, as shown in Figure 12. Each defect is categorised in one of seven classes depending on which combinations of perspective that have a FOUND measure greater than zero. The numbers in the Venn-diagrams indicate how many defects that belong to each class. For example, for the PG document, there are 10 defects which were found by all perspectives, while 5 defects were found by both the user and designer perspectives and only one defect was found solely by the user perspective.

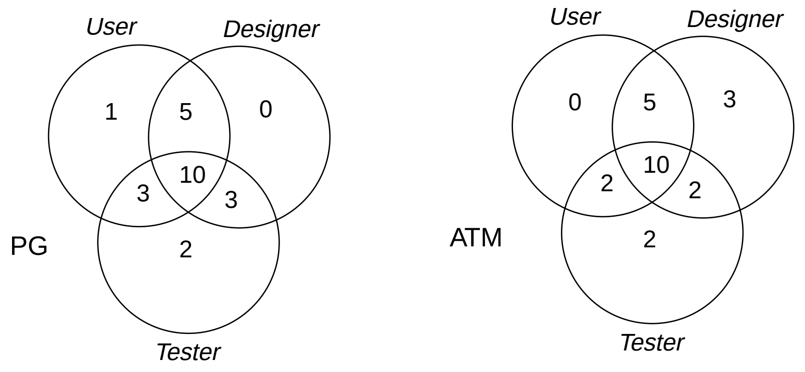


Figure 12. *Defect coverage for the PG and ATM documents.*

This type of analysis is very sensitive to the number of subjects. It is enough that only one reviewer finds a defect, for the classification to change. The probability that a defect is found increases with the number of reviewers, and if we have a large number of reviewers the defects will be more likely to be included in the class where all perspectives have found it. This means that this type of analysis is not very robust, and does not provide meaningful interpretations in the general case. In our case, we can at least say that the defect coverage analysis in Figure 12 does not contradict our previous results that we cannot reject the hypothesis that the perspectives are similar with respect the sets of defects that they find. The defects found by all perspectives is by far the largest class.

7. Interpretations of Results

In this section the data analysis is interpreted with respect to the hypotheses stated in Section 4.2. The three hypotheses are basically that there are differences among the performance of the perspectives with respect to effectiveness, efficiency and specific defects found, see Table 2. Note that the checklist method is treated as a fourth perspective in the analysis.

The first two hypotheses are tested using ANOVA and PLSD tests. The following three hypotheses can be rejected:

- $H_{0,EFF}$. The perspectives are assumed to have the same finding efficiency. This hypothesis can be rejected based on the ANOVA analysis on a p-level of 0.01. A PLSD test ($p<0.01$) shows that checklist reading has a higher efficiency measure than the other three perspectives.
- $H_{0,RATE}$. The perspectives are assumed to have the same detection rates. This hypothesis can be rejected based on the ANOVA analysis on a p-level of 0.04. A PLSD test ($p<0.03$) shows that checklist reading has a higher detection rate than the other three perspectives.

The high performance of the checklist perspective may be explained by the difference between the subjects; the PhD students may be more experienced and motivated compared to the MSc students.

The third hypothesis is tested using a chi-square test:

- $H_{0,FOUND}$. The perspectives are assumed to find the same defects. This hypothesis can *not* be rejected based on the chi-square test.

Another difference that is statistically significant, is that more time is spent on the ATM document than on the PG document. Otherwise there are no statistical differences between the documents.

In summary it can be concluded that more experienced reviewers using checklist as reading technique, perform better than less experienced reviewers, using PBR. This is true with respect to effectiveness and efficiency. This indicates that the experience seems to have larger impact on the performance than the reading technique. Unfortunately, the constraints behind the experiment design hinders the ability to make a cross-check to investigate if more experienced reviewers with PBR perform better than less experienced reviewers with checklists.

An even more interesting result from the study is that there is no significant difference between the three perspectives, user, design and test. This is true for all the three hypotheses, i.e. there is no significant difference in terms of effectiveness or efficiency. Furthermore, there is no significant difference in time spent using the different perspectives, hence, the time spent does not bias in favour of any of the techniques. The lack of difference between the three perspectives does, if the result is possible to replicate and generalize, seriously affect the cornerstones of the PBR. The advantages of PBR are assumed to be that the different perspectives focus on different types of defects, and thus detect different defect sets. This study shows no statistically significant difference between the sets of defects found by the three perspectives, and thus the advantages of PBR can be questioned.

Threats to the conclusion validity of the results are that the number of samples is low, in particular for the chi-square test. However, the bar charts over the defects found by different perspectives (see Figure 9) do not indicate any clear pattern, which supports the non-significant results. The ANOVA statistics are applied within acceptable limits, and these do not show any difference between the perspectives. Furthermore, the design itself is a limitation in the study, as mentioned before. The specific perspectives and the reading techniques for the perspectives might also be a threat to the validity of the results, when trying to apply the results to scenario-based reading in general.

The validity threat regarding the motivation of subjects can be evaluated by comparing the detection rates of the Lund study with other studies. The individual PBR detection rate for the NASA study (Basili, 1996) was on average 0,249 for the pilot study and 0,321 for the main run, while the Lund study shows an average individual PBR detection rate of 0,252. The rates are comparable, supporting the assumption that the subjects in this study was as motivated as in the NASA study.

Other threats to the validity, as elaborated in Section 4.4, are not considered differently in the light of the result.

8. Conclusions

The study reported in this paper is focused on the evaluation of Perspective Based Reading (PBR) of requirements documents. The study is a partial replication of previous experiments in an academic environment based on the University of Maryland lab package.

The objective of the presented study is twofold:

1. Investigate the differences in the performance of the perspectives in terms of effectiveness (defect detection rate) and efficiency (number of found defects per hour).

2. Investigate the differences in defect coverage of the different perspectives, and hence evaluate the basic assumptions behind PBR supposing that different perspectives find different defects.

The experiment setting includes two requirements documents and scenarios for three perspectives (*user* applying use case modelling, *designer* applying structured analysis, and *tester* applying equivalence partitioning). A total of 30 MSc students were divided into 3 groups, giving 10 subjects per perspective. A control group of 9 PhD students used a checklist reading technique. Due to limitations imposed by the educational context, the particular experiment design introduces confounding factors that makes it impossible to differentiate effects of the reading technique (PBR and checklist) from effects of the subject type (MSc and PhD).

In summary the results from the data analysis show that:

1. There is no significant difference between the user, designer and tester perspectives in terms of defect detection rate and number of defects found per hour.

2. There is no significant difference in the defect coverage of the three perspectives.

3. PhD students with a checklist approach find significantly more defects per hour and have a significantly higher detection rate than MSc students with a PBR approach.

The interpretation of these results suggests that a combination of multiple perspectives may not give higher coverage of the defects compared reading with only one perspective. The data also indicate that individual experiences and abilities are more important than the reading technique itself.

The results contradict the main assumptions behind PBR and also some of the previously conducted studies, summarized in Section 2. Some of the previous studies have shown significant advantages with PBR over Ad Hoc inspection, but no statistical analysis on the difference between perspective performance is made in any of the studies reported in Section 2. Furthermore, the previous studies have not taken the efficiency into account (number of defects found per hour), but concentrates on detection rate as the main dependent variable. From a software engineering perspective, where the cost and efficiency of a method are of central interest, it is very interesting to study not only the detection rate, but also if a method can perform well with limited effort.

There are a number of threats to the validity of the results, including:

- The setting may not be realistic.
- The perspectives may not be optimal.
- The subjects may not be motivated or trained enough.
- The number of subjects may be too small.

It can be argued that the threats to validity are under control, as (1) the inspection objects are similar to industrial requirements documents, (2) the perspectives are motivated from a software engineering process view, and (3) the subjects were 4th year students with a special interest in software engineering attending a self-chosen course. However, a single study, like this, is no sufficient basis for changing the attitudes towards PBR. Conducting the same analyses on data from existing experiments as well as new replications with the purpose of evaluating differences between perspectives will bring more clarity into the advantages and disadvantages of PBR techniques, and also give a better control over the validity threats.

In future work, the following issues are of certain interest:

New PBR scenarios. In the presented experiment, use cases are constructed *after* the requirements document is completed. This may not be optimal, as use cases is a promising technique for eliciting, analysing and documenting requirements (Weidenhaupt, 1998). Given that use cases are incorporated into the requirements document, we may design a new inspection scenario for the user perspective in PBR. Use cases have also shown to be interesting as a basis for test case construction (Regnell, 1999). Instead of equivalence partitioning, a use case based approach for the tester perspective may be used as well.

Usage-base reading. Given that a use case model is part of the requirements document rather than created afterwards, this can be a starting-point for a new type of inspection technique as an alternative to PBR (Regnell, 1998), where the focusing of inspection effort is based on use cases (i.e. system usage scenarios rather than meta-level inspection scenarios). To be able to conduct inspections with usage-based reading, we need to annotate the use cases with priority information. Techniques based on pairwise comparison may be used here (Karlsson, 1998). By comparing pairs of use cases (U_x, U_y), we may prioritise them according to criteria such as “U_x is more frequently used than U_y” and “U_x is more critical to hazard than U_y”.

With the methods in (Karlsson, 1998) it is possible to derive the relative priority p_i, ($0 \leq p_i \leq 1$, $\sum p_i = 1$), of each use case U_i. Based on this, we propose to conduct usage-based reading of design or code documents using the following scheme of *effort partitioning*: (1) Prioritise the use cases; (2) Decide on the total time T to be spent on reading of artifact A; (3) Assign the time $T_i = p_i \cdot T$ to each use case U_i; (4) For each use case U_i, inspect A for a period of T_i by “walking through” the events of U_i and decide if A is correct with respect to U_i.

By using the priority criterion of usage frequencies to focus reading by use cases, we get a static verification that corresponds to the expected operational conditions of the system. If we also record the reading time between found defects, we may use these measures to derive an estimate of reliability based on the mean time between defects.The performance of the proposed usage-based reading technique is an interesting area of further empirical research.

Appendix A. Individual performance

Table 4. Data for each subject.

ID	PERSP	DOC	STYPE	TIME	DEF	EFF	TOT	RATE
1	U	ATM	MSc	187	8	2,567	29	0,276
2	D	PG	MSc	150	8	3,200	30	0,267
3	T	ATM	MSc	165	9	3,273	29	0,310
4	U	PG	MSc	185	11	3,568	30	0,367
5	D	ATM	MSc	155	8	3,097	29	0,276
6	T	PG	MSc	121	8	3,967	30	0,267
7	U	ATM	MSc	190	7	2,211	29	0,241
8	D	PG	MSc	260	7	1,615	30	0,233
9	T	ATM	MSc	123	6	2,927	29	0,207
10	U	PG	MSc	155	6	2,323	30	0,200
11	D	ATM	MSc	210	11	3,143	29	0,379
12	T	PG	MSc	88	9	6,136	30	0,300
13	U	ATM	MSc	280	11	2,357	29	0,379
14	D	PG	MSc	145	11	4,552	30	0,367
15	T	ATM	MSc	170	5	1,765	29	0,172
16	U	PG	MSc	120	6	3,000	30	0,200
17	D	ATM	MSc	190	9	2,842	29	0,310
18	T	PG	MSc	97	5	3,093	30	0,167
19	U	ATM	MSc	295	2	0,407	29	0,069
20	D	PG	MSc	180	7	2,333	30	0,233
21	T	ATM	MSc	306	7	1,373	29	0,241
22	U	PG	MSc	223	4	1,076	30	0,133
23	D	ATM	MSc	157	6	2,293	29	0,207
24	T	PG	MSc	130	6	2,769	30	0,200
25	U	ATM	MSc	195	13	4,000	29	0,448
26	D	PG	MSc	200	7	2,100	30	0,233
27	T	ATM	MSc	195	8	2,462	29	0,276
28	U	PG	MSc	125	5	2,400	30	0,167
29	D	ATM	MSc	200	8	2,400	29	0,276
30	T	PG	MSc	150	5	2,000	30	0,167
31	C	ATM	PhD	102	18	10,588	29	0,621
32	C	PG	PhD	96	9	5,625	30	0,300
33	C	ATM	PhD	113	7	3,717	29	0,241
34	C	PG	PhD	98	7	4,286	30	0,233
35	C	ATM	PhD	140	14	6,000	29	0,483
36	C	PG	PhD	136	11	4,853	30	0,367
37	C	ATM	PhD	165	13	4,727	29	0,448
38	C	ATM	PhD	204	9	2,647	29	0,310
39	C	ATM	PhD	296	11	2,230	29	0,379

Appendix B. Defects found by perspectives

B.1 PG document

Table 5. Defects id D# found (1) or not found (0) by individuals reading the PG document.

	Individuals																	
	User Perspective						Tester Perspective						Designer Perspective					
D#	**2**	**8**	**14**	**20**	**26**	**Σ**	**4**	**10**	**16**	**22**	**28**	**Σ**	**6**	**12**	**18**	**24**	**30**	**Σ**
1	0	0	1	0	1	2	1	0	0	0	0	1	1	0	0	1	0	2
2	1	1	1	0	1	4	1	0	1	0	0	2	0	1	1	1	0	3
3	0	0	0	0	0	0	1	1	0	1	0	3	0	0	0	1	0	1
4	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	1
5	0	0	1	1	1	3	1	0	0	0	1	2	0	1	1	0	0	2
6	1	1	0	0	0	2	0	1	1	0	0	2	1	0	0	1	1	3
7	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1
8	1	1	1	1	1	5	1	1	1	1	0	4	1	1	0	1	0	3
9	1	1	1	1	0	4	1	1	1	0	0	3	0	1	0	0	0	1
10	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
11	1	1	0	1	1	4	0	0	0	0	0	0	0	1	1	0	0	2
12	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	1	1	1	1	0	1	1	4	1	1	0	1	1	4
15	0	0	1	0	0	1	0	1	0	0	1	2	1	0	0	0	0	1
16	1	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	2
17	0	0	0	1	1	2	1	0	0	0	0	1	0	1	0	0	0	1
18	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	1	0	0	1	0	0	0	0	1	1	1	0	1	0	1	3
22	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1
23	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
24	0	0	1	1	0	2	0	0	0	0	1	1	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
28	0	1	1	0	0	2	0	0	0	0	0	0	1	0	0	0	0	1
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	1	0	1	0	0	2	0	0	1	0	0	1	0	0	0	0	0	0
Σ	8	7	11	7	7	40	11	6	6	4	5	32	8	9	5	6	5	33

B.2 ATM document

Table 6. Defects number D# found (1) or not found (0) by individuals reading the ATM document.

	Individuals																	
	User Perspective						**Tester Perspective**						**Designer Perspective**					
D#	**1**	**7**	**13**	**19**	**25**	Σ	**3**	**9**	**15**	**21**	**27**	Σ	**5**	**11**	**17**	**23**	**29**	Σ
1	0	0	0	1	0	1	0	0	0	1	0	1	1	1	0	0	1	3
2	1	0	1	0	0	2	1	0	1	1	1	4	1	1	1	0	1	4
3	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	2
4	0	1	1	1	0	3	1	1	0	0	0	2	1	1	1	0	1	4
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	2
7	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	2
8	0	0	1	0	0	1	0	1	0	1	1	3	1	1	0	0	1	3
9	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	2
10	0	1	1	0	0	2	0	0	0	0	0	0	0	1	0	0	0	1
11	1	0	0	0	0	1	0	1	0	1	1	3	1	0	1	0	0	2
12	1	1	1	0	0	3	0	0	1	1	1	3	0	1	0	0	0	1
13	1	0	1	0	0	2	0	0	0	0	0	0	0	1	1	1	0	3
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	1	0	0	0	1	1	0	0	0	1	2	0	0	0	0	0	0
16	0	1	1	0	0	2	1	1	1	0	0	3	0	1	1	0	0	2
17	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	1
18	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1
19	1	0	0	0	0	1	1	1	0	0	0	2	0	0	0	1	0	1
20	0	0	1	0	0	1	0	0	1	1	0	2	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
23	0	0	0	0	0	0	0	0	1	0	1	2	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	1	0	1	0	2	0	0	0	0	0	0
27	1	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	1	3
28	1	0	1	0	0	2	1	0	0	0	0	1	1	0	1	0	1	3
29	1	1	1	0	0	3	1	0	0	0	1	2	0	0	0	1	0	1
Σ	8	7	11	2	0	28	8	6	5	7	8	34	8	11	9	6	8	42

Acknowledgements

First of all, the authors would like to thank the students who participated as subjects in the experiment. We would also like to give a special acknowledgement to Forrest Shull at University of Maryland who provided support on the UMD lab-pack and gave many good comments on a draft version of this paper. Thanks also to Claes Wohlin, Martin Höst and Håkan Petersson at Dept. of Communication Systems, Lund University, who have carefully reviewed this paper. Special thanks to Anders Holtsberg at Centre for Mathematical Sciences, Lund University, for his expert help on statistical analysis. This work is partly funded by the National Board of Industrial and Technical Development (NUTEK), Sweden, grant 1K1P-97-09690.

References

Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., and Zelkowitz, M. 1996. "The Empirical Investigation of Perspective-Based Reading" *Empirical Software Engineering*, 1(2): 133-164.

Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., and Zelkowitz, M. 1998.
Lab Package for the Empirical Investigation of Perspective-Based Reading
Available at: http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html

Ciolkowski, M., Differding, C., Laitenberger, O., and Münch, J. 1997. "Empirical Investigation of Perspective-based Reading: A Replicated Experiment" *ISERN Report* no. 97-13.
Available at: http://www.iese.fhg.de/ISERN/pub/isern_biblio_tech.html

Cook T. D., and Campbell D. T. 1979 *Quasi-Experimentation – Design and Analysis Issues for Field Settings*, Houghton Mifflin Company.

Doolan, E. P. 1992. "Experiences with Fagan's inspection method" *Software Practice and Experience*, 22(2):173-182.

Fagan, M. E. 1971. "Design and Code Inspections to Reduce Errors in Program Development" *IBM System Journal*, 15(3):182-211.

Fusaro, P., Lanubile, F., and Visaggio, G. 1997. "A Replicated Experiment to Assess Requirements Inspection Techniques" *Empirical Software Engineering*, 2(1): 39-57.

Humphrey, W. S. 1989. *Managing the Software Process*. Addison-Wesley.

Johnson, P. M., and Tjahjono, D. 1998. "Does Every Inspection Really Need a Meeting?" *Empirical Software Engineering*, 3(1): 9-35.

Karlsson, J., Wohlin, C., and Regnell, B. 1998. "An Evaluation of Methods for Prioritizing Software Requirements" *Information and Software Technology*, 39(14-15).

Lott, C. M., and Rombach, H. D. 1996. “Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques”. *Empirical Software Engineering*, 1(3): 241-277.

Miller, J., Wood, M., and Roper, M. 1998. “Further Experiences with Scenarios and Checklists” *Empirical Software Engineering*, 3(1): 37-64.

Montgomery, D. C. 1997. *Design and Analysis of Experiments*. Fourth Edition. Wiley.

Porter, A., Votta, L., and Basili, V. R. 1995. “Comparing Detection Methods for Software Requirements Inspection: A Replicated Experiment” *IEEE Transactions on Software Engineering*, 21(6):563-575.

Porter, A., and Votta, L. 1998. “Comparing Detection Methods for Software Requirements Inspection: A Replication Using Professional Subjects” *Empirical Software Engineering*, 3(4):355-380.

Regnell, B., and Runeson, P. 1998 “Combining Scenario-based Requirements with Static Verification and Dynamic Testing”, *Proceedings of 4th Intl Workshop on Requirements Engineering - Foundation for Software Quality* (REFSQ'98), Pisa, Italy.

Regnell, B., Runeson, P., and Wohlin, C. 1999. “Towards Integration of Use Case Modelling and Usage-Based Testing”, to appear in *Journal of Systems and Software*, Elsevier.

Robson, C. 1993. *Real World Research*. Blackwell.

Sandahl, K., Blomkvist, O., Karlsson, J., Krysander, C., Lindvall, M., and Ohlsson, N. 1998. “An Extended Replication of an Experiment for Assessing Methods for Software Requirements” *Empirical Software Engineering*, 3(4):381-406.

Shull, F. 1998. *Developing Techniques for Using Software Documents: A Series of Empirical Studies.* PhD Thesis, Computer Science Department, University of Maryland, USA.

Siegel, S., and Castellan N. J. 1988. *Nonparametric Statistics for the Behavioral Sciences*. Second Edition. McGraw-Hill.

Sørumgård, S. 1997. *Verification of Process Conformance in Empirical Studies of Software Development*. PhD Thesis, Department of Computer and Information Science, The Norwegian University of Science and Technology, Norway.

Votta, L. G. 1993. “Does Every Inspection Need a Meeting?” *Proceedings of the ACM SIGSOFT 1993 Symposium on Foundations of Software Engineering, ACM Software Engineering Notes*, 18(5):107-114.

Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. 1998. “Scenarios in System Development: Current Practice”, *IEEE Software*, March/April 1998, pp. 34-45.

PAPER VI:

From Requirements to Design with Use Cases - Experiences from Industrial Pilot Projects

Björn Regnell and Åke Davidson

Published in Proceedings of REFSQ'97, *3rd Intl Workshop on Requirements Engineering - Foundation for Software Quality*, Barcelona, Spain, June 1997.

Abstract

In systems evolution, new requirements are distributed on existing architectures. This paper describes a method for modelling how new requirements are distributed on a hierarchy of existing system components. The method applies use case modelling in the transition from requirements to design, with focus on requirements traceability and dynamic system behaviour modelling. The method is based on a recursive process where functionality specification and distribution activities are applied at different abstraction levels in the component hierarchy. The method has been evaluated in three realistic projects, concerned with the evolution of a complex real-time cellular switching system. The subjective conclusions from these evaluations suggest that use case modelling is useful in requirements analysis and distribution within the studied domain.

1. Introduction

The normal case in most software development projects is that existing systems are extended with new capabilities [1]. When new requirements are implemented in an existing architecture, it is vital that these requirements are distributed on present architectural elements without destroying the system structure.

This paper describes motives for, and results from, a method development project at Ericsson Radio Systems, focused on the transformation from requirements to design in the context of system evolution. A method, called FRED (From REquirements to Design), is under development as a part of REME (Requirements Engineering Methods for Ericsson), which is a collection of method components applicable in the early phases of software development.

An important ingredient of FRED is *use case modelling* [2]. The main concepts of use case modelling are *actors* and *use cases*. An **actor** represents an entity (human or non-human) external to the system under development, that communicates with the system in order to achieve certain goals. A **use case** is a generalisation of a usage situation where one or many actors interact with the system to accomplish specific goals. One use case may cover several sequences of events (so called *scenarios* or *flow variants*). A use case may be described either from an external (*black-box*) point of view suitable for requirements, or from an internal (*white-box*) point of view suitable for design. Related work on use case modelling can be found in, e.g., [3, 4, 5, 6, 7].

The FRED method has been developed over a period of several years, taking previous method engineering efforts at Ericsson into account. FRED has been evaluated in three industrial pilot projects with real requirements, and experiences from each of these studies have resulted in incremental improvements of the method.

The goal of this paper is threefold: (1) to describe the problems motivating the presented work, (2) to describe the FRED method, and (3) to report on its evaluation.

The paper is organized as follows. Section 2 gives the context of Ericsson's software development and characterises its products, projects and practises. In Section 3, the problems motivating the development of FRED are stated. An overview of the method is given in Section 4, with its main concepts, models and activities. Section 5 describes the evolution and evaluation of FRED, and Section 6 presents some conclusions and issues of interest in the further development of FRED.

2. Context

Ericsson Radio Systems is the world-leading manufacturer of cellular switching systems. Software intensive products are provided for all major cellular standards. All systems are based on a common architecture, called AXE, which includes a large number of software components, distributed in a network.

The *products* derived from this architecture have the following characteristics:

- *Large and complex*. Systems typically include several million lines of code.
- *Very high reliability demands*. System failure often leads to substantial financial loss, and AXE-based switches are used in safety-critical application domains.
- *Distributed real-time systems*. Systems consist of many hardware and software components that are geographically distributed and communicate in real time with high performance demands.
- *Many installations on different markets*. AXE-based cellular switching systems serve approximately 50 million mobile subscribers in close to 100 countries.
- *Support for many different standards*. The telecommunication domain includes many different standards, which increase complexity.

The development of systems is carried out in development *projects* with the following characteristics:

- *Large and complex*. Projects involve a large number of people to a substantial cost.
- *Distributed development*. The development of AXE is distributed over many countries, with different languages and culture.

- *Evolution rather than create-from-scratch*. Development projects are rarely concerned with development of completely new systems. Instead, new features are added to the existing architecture.

The software engineering *practice* used by Ericsson in the development of AXE-based systems includes:

- *Incremental development*. A product management function collects requirements and defines increments of evolutionary changes in the AXE system. For each increment, a document-driven process, similar to the common waterfall-process [8] is applied. The early phases (prestudy and feasibility study) have a strong focus on assessment of risks, costs, and impact.
- *Inspection and testing*. Projects have a strong focus on quality, and formal inspections [9] and rigorous testing at different abstraction levels are applied.
- *Object-based design*. Data and behaviour are encapsulated in objects, called **blocks**, which communicate with asynchronous messages called **signals**.

The AXE architecture includes a large set of blocks, and it is not uncommon that blocks include over 10 000 lines of code. To manage a distributed development an facilitate effective configuration management, blocks are arranged in a hierarchy of system components (nodes, subsystems, block groups).

The AXE architecture has evolved over several decades and its components with belonging documentation constitute an extensive **design base**. Over time, the development processes, methods and standards have also evolved.

3. Problem Description

Ericsson is a highly successful software development organisation, but as new requirements arrive and complexity grows, there is a constant need for improvements. The objective of all improvement efforts at Ericsson are a better achievement of the following goals: (G1) high product quality, (G2) short time-to-market, and (G3) low development costs.

Ericsson's ability to achieve a good trade-off between these goals relies on (1) a system *architecture* that allows a controlled architectural evolution when new features are incorporated, and (2) a development *process* that can, *predictably*, achieve a good balance between G1-G3.

The block level has a comprehensive documentation, but the AXE design base includes little documentation on how requirements are distributed on high-level software components, and how components interact at higher abstraction levels (e.g. network, node, and subsystem level).

Improved requirements traceability [10] is assumed to give better support for impact analysis (the assessment of system change induced by new and changed requirements). Furthermore, improved impact analysis is assumed to improve increment planning, and effort prediction, as well as give a higher control over the system evolution. This in turn, is assumed to support a better achievement of G1-G3.

By capturing the responsibilities of architectural components, and providing high-level models of component interaction, it is assumed that system understandability is increased and that development becomes less dependent on experienced designers which know the architectural principles by heart. This is also assumed to positively influence G1-G2. It is also assumed that a shift of some effort from low-level design and implementation to requirements analysis and high-level design supports the achievement of G1-G3.

These assumptions have not been validated in controlled experiments, but are nevertheless a basis for improvement efforts at Ericsson. The method development work reported here is, in the consequence of these assumptions, focused on requirements traceability and high-level system modelling.

Some of the problems motivating the presented work are (1) *how to model (telecommunication) systems at a high abstraction level*, (2) *how to achieve traceability between requirements and different design entities*, (3) *how to model the distribution of new requirements on existing system components*.

The development of the FRED method represents a first step in the search of good solutions to these problems. The current version of FRED is briefly described in the following section.

4. Solution Proposal

The FRED[1] method tries to address the problems described previously, by combining use case modelling with a hierarchical system model. The models produced using FRED express how new requirements are distributed on existing software components. Before requirements are distributed on a set of components, new components may be introduced, and existing components may be changed.

The system **architecture** is viewed as a set of communicating components. A **component** is an entity on which requirements can be distributed. A component is defined in terms of its responsibilities [11], and its interface to its environment. Components may contain other components, in a hierarchical manner, creating a tree structure. The leaf components of this tree are called **target components**. Only target components may contain executable software, while **intermediate components** only act as containers for other components. The root of the tree is called **top-level component**, and represents the component for which the requirements distribution process is initiated. Entities outside the top-level component are represented by *actors*.

The *input* to FRED is (1) a numbered list of textual requirements, and (2) a top level component including its architecture. The *output* of FRED is comprised of three interrelated models that describe the result of the requirements distribution in a way suitable for low-level design of target components.

FRED focuses on the *modelling* of the result of requirements distribution, and does not prescribe *how* the architecture is to be changed or extended to meet the new requirements. Architectural evolution relies on design expertise and domain knowledge, which is not captured by FRED (except for a number of informal, telecommunication specific guidelines not elaborated here).

FRED assumes that a component's internal structure on the next hierarchical level is defined *before* requirements are distributed on its contained components, but a top-down definition of the architecture is *not* prescribed.

1. An extensive WWW-based guide for FRED is provided on Ericsson's intranet; the overview presented here is a simplification and only focuses on the main aspects of the method.

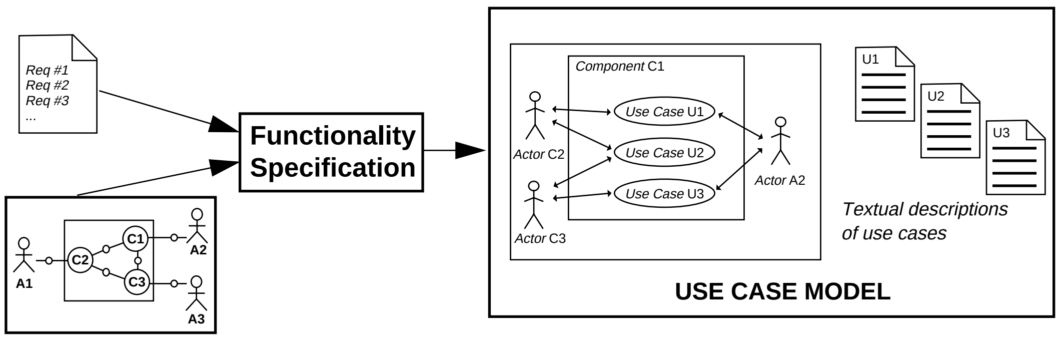


Figure 1. *The Functionality Specification activity creates a use case model for a component (C1) from a black-box point of view, based on textual requirements and its architectural environment (C2, C3, A2).*

FRED is mainly focused on the functional and dynamical aspects of real-time systems and does not prescribe a specific method or notation for static data modelling. (FRED projects may use a data dictionary to model passive data entities, but it is also possible to use E/R-diagrams [12] or some object-oriented modelling language, e.g. OMT [13].)

The models and activities of FRED are applied *recursively* for each non-target component in the component hierarchy. When a target component is reached, the recursion stops and a low-level design method succeeds FRED.

For each recursion the following three activities are performed in an iterative manner: Functionality Specification, Component Specification, and Functionality Distribution, as described in the following.

Functionality Specification activity. Use cases of a component are defined and described from a black-box point of view. *Input*: a list of numbered requirements and (if not a top-level recursion) a Distribution Model from a higher level recursion. *Output*: a validated Use Case Model including a set of textual, black-box use cases. The external components (specified in a previous recursion) that communicate with the component for which the use cases are defined, becomes actors from the viewpoint of the use cases at this recursion level. (See Figure 1.)

Component Specification activity. A component is defined in terms of its responsibilities and described in terms of its actors, contained components, and communication paths. *Input*: a list of numbered requirements and a Use Case Model from the Functionality Specification activity, and domain-specific architectural principles and guidelines. If this activity is not in a top-level recursion: a Component Model and a Distribution Model, including white-box use cases (output of the Functionality Distribution activity, described below) from a higher level recursion. *Output*: a validated Component Model, describing each contained component's responsibilities, and interfaces. (See Figure 1.)

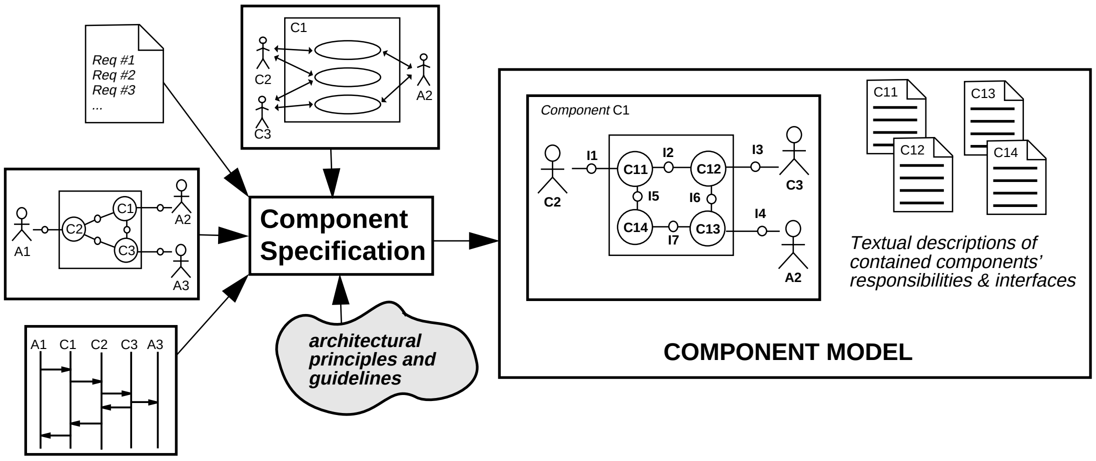


Figure 2. *The Component Specification activity creates a component model for a component (C1) by describing its contained components (C11-C14) and its internal and external communication structure in terms of interfaces (I1-I7).*

Functionality Distribution activity. Functional requirements for a component are distributed on its contained components. The distribution of the functionality specified in black-box use cases is expressed with white-box use cases using the standardised notation of Message Sequence Charts (MSC) [14]. Each white-box use case models a set of scenarios (flow variants), where sequences of signals between actors and contained components are specified. Each MSC is annotated with semi-formal pseudo-code, specifying operations in the contained components. *Input*: a Use Case Model and a Component Model from the activities above, together with architectural principles for component communication. If not at top-level recursion: white-box use cases regarding the concerned component from higher levels. *Output*: a validated Distribution Model, specifying requirements distribution in white-box use cases. (See Figure 1.)

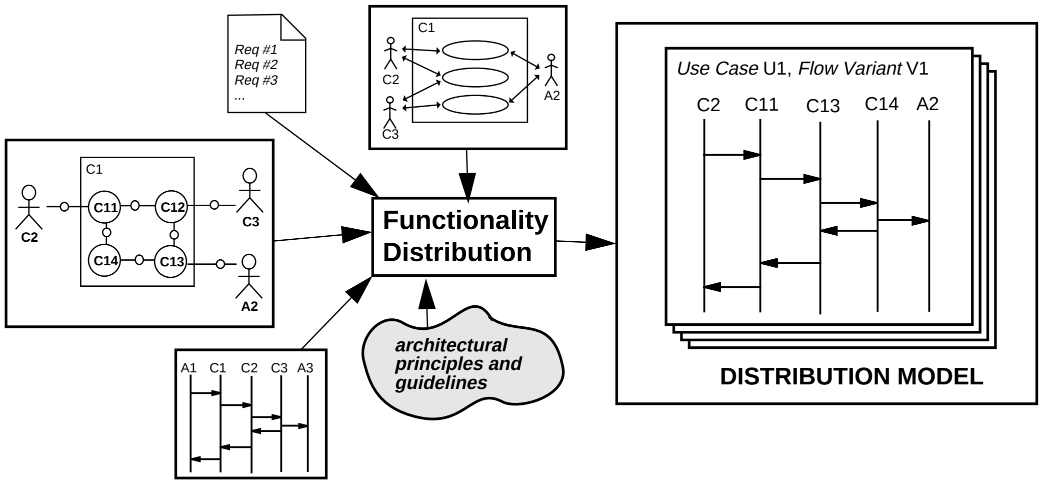

Req #1
Req #2
Req #3
...
C1
C2
C3
A2
C11
C12
C13
C14
Functionality Distribution
architectural principles and guidelines
A1 C1 C2 C3 A3
Use Case U1, Flow Variant V1
C2 C11 C13 C14 A2
DISTRIBUTION MODEL

Figure 3. *The Functionality Distribution activity creates, for each component's black-box use case, a number of white-box use cases using MSC diagrams, specifying how requirements are distributed on contained components.*

Different types of requirements. Functional Specification focuses on *functional requirements*, regarding dynamic system input, output, and control, but *quality requirements*[1], such as performance, reliability, and flexibility, may also be considered here. The activities of Functional Specification and Component Specification both assume a list of numbered requirements as input. Component Specification focuses on quality requirements and *architectural requirements*, meaning requirements that address the structure or interfaces of components, e.g. that some responsibilities shall be physically separated in a specific component, or that a specific standard protocol shall be supported.

Requirements traceability. During Functional Distribution, each textual requirement is assigned to one or more components on the subordinate level, by creating links between the numbered requirements and the components. These links are the main vehicle for requirements traceability, and most benefits are gained if these links are maintained automatically

1. The terms non-functional requirements [15], non-behavioural requirements [16], and quality attributes [17] have been used for this type of requirements. The classification of requirements into different types is useful in conceptual discussions, but is often difficult to make in practice.

by a tool. To support traceability between different recursion levels, FRED recommends to maintain typed links at least between:

- requirements and use cases, e.g. when a use case describes a requirement,
- use cases and use cases, e.g. when a use case relates to another use case on the next recursion level,
- use cases and components, e.g. when one white-box use case distributes requirements on a set of components,
- requirements and components, e.g. when requirements are assigned to a component,
- components and components, e.g. when components contain other components.

Vertical traceability [10] can be achieved by maintaining such links between entities in the models. The impact of, for example a changed requirement, may be investigated by following the links through the hierarchy *down* to target components. By following links from a target component *up* through the hierarchy, the requirements associated with this component can be found.

Process configuration. Before FRED is applied in a specific project, the recursive process needs to be configured according to the specific needs of the project. The *number of recursion levels* needs to be decided. For example, it might be decided that the recursion should be applied on three levels: network level, node level, and subsystem level. In another case it may be decided to stop the recursion at the node level, and directly continue with other modelling techniques used in low-level design.

It is also necessary to decide which of the existing components in the design base that should be included in the component models. It is possible to stop modelling a specific recursion branch before the lowest recursion level is reached, if the components of this branch has little or no relevance to the requirements considered by the project. For instance, some target components may be at the node level, while others are at the subsystem or block level.

In Figure 4, a principal example of one recursion branch is shown. FRED is used to model three abstraction levels. At the fourth level, target components are described with some low-level design technique, for example SDL process diagrams [18].

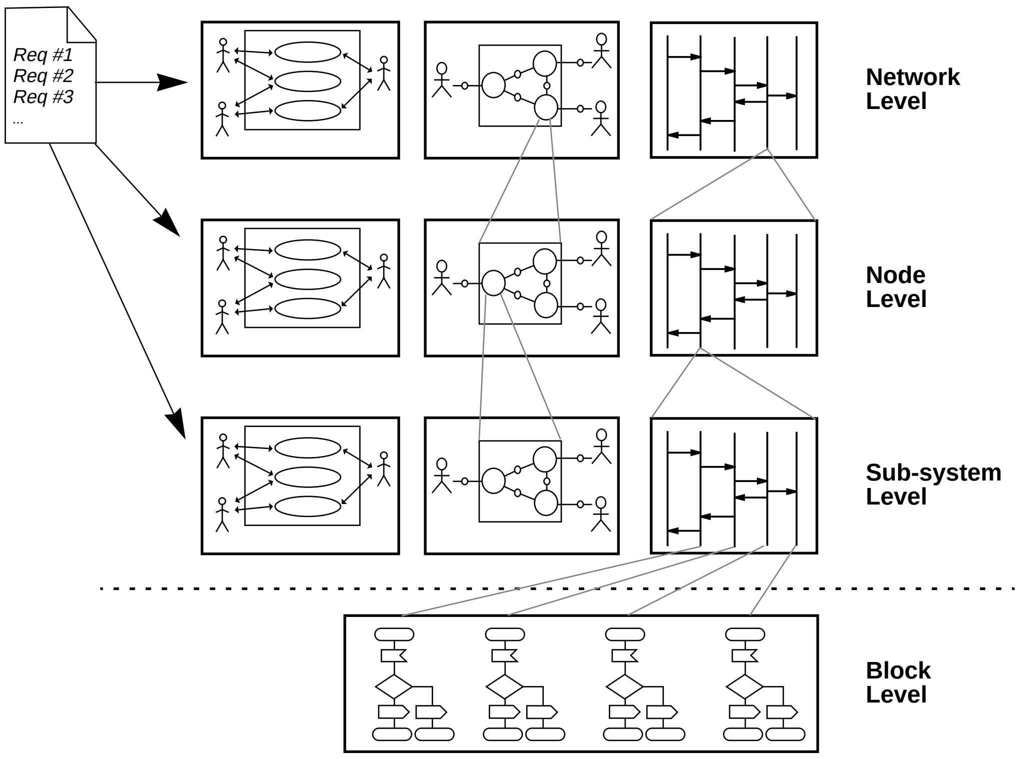


Figure 4. *An example recursion branch, where FRED is used on three levels and a state based technique is used on the fourth level.*

Requirements distribution vs. functional decomposition. In common applications of structured analysis [19], the architecture of a new system is determined in a strict top-down manner, by decomposition of functions according to a divide-and-conquer approach [20, 21]. Functions are organized in a strict hierarchy, and the system architecture corresponds to this hierarchy of functions.

Experiences at Ericsson indicate, however, that quality requirements (performance, reliability, flexibility, etc.) are more important to consider when creating a good system architecture, than an incidental decomposition of functions. In FRED, the functions are represented by use cases, but they are not decomposed in a top-down fashion. Use cases on a subordinate level is defined based on relevant information in all use cases from a higher level together with knowledge of target component responsibilities, and thus considering not only “previous” use cases in one recursion branch.

The component structure is still hierarchical, as this has proven to be an efficient way of managing complexity, but the responsibilities of components are determined through a distribution of quality requirements and architectural constraints, rather than on a decomposition of use cases. The functional requirements are distributed according to the responsibilities of components.

The use cases for a component *C* at recursion level *n* are defined based on its actor’s goals and its responsibilities, rather than on a decomposition of a use case from recursion level *n*-1. White-box use cases of level *n*-1, concerning *C*, are studied when defining level *n* black-box use cases, but the set of all use cases for all levels is not a decomposition hierarchy. An important rule constrains (white-box) use cases at different recursion levels: a use case on level *n*, should not be inconsistent (in terms of signals from/to its actors) with use cases on level *n*-1.

Before new requirements are distributed on an existing architecture, it may be necessary to adjust the component hierarchy, e.g. by splitting a large component into several smaller components, or transferring responsibilities from one component to another. Such architectural adjustments are not carried out in a top-down manner.

5. Validation

FRED has been developed and evaluated through a series of three method applications, concerned with new requirements on real features in the AXE architecture. This section describes the evaluation settings and report on some subjective conclusions from these pilot projects.

The pilot projects were not carried out as formal case studies [22, 23]. The reasons for this are: (1) the method was developed in parallel with the pilot projects, so it was impossible to find a stable treatment and compare with a control, and (2) no objective measurement were undertaken for use in statistical analysis (except for total person-hours in the last two projects). Instead, the pilot projects were used to get subjective opinions from the involved engineers, and based on these opinions further improve the FRED method.

The main result of the evaluation studies is thus represented by the method itself, as it is described in the previous section. This method version has been productified and more pilot projects are planned to further assess and improve the method and its tool support.

5.1 Evaluation Design

The development of FRED started with studies of existing literature. This survey, together with previous experience in requirements distribution, and the problem description in Section 3, formed the input to the design of FRED, version 0.1. Hereafter, the method has been improved in three increments, based on experience gained by method applications in realistic projects, as shown in Figure 5.

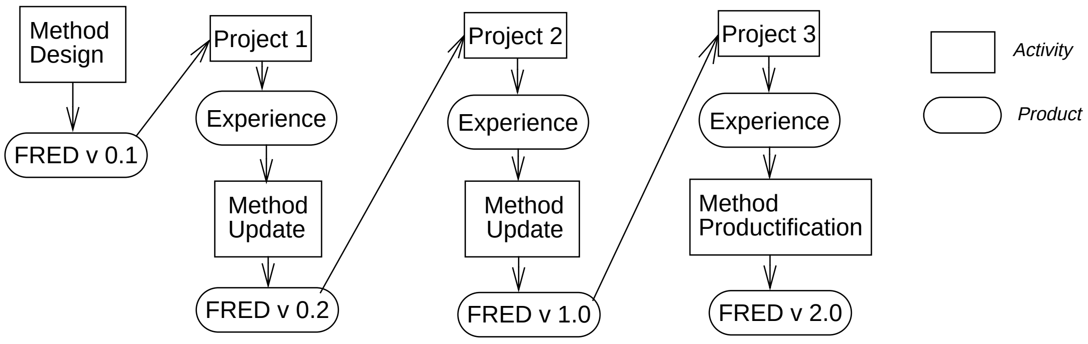


Figure 5. *The FRED method has been evaluated in three subsequent studies.*

Project 1 and 3 was carried out off-line, in parallel with a real project, while Project 2 was carried out as an on-line project, where the results were used in further development. During these projects, the method was changed, as it was sometimes necessary to find immediate work-arounds for unanticipated method problems.

FRED v 0.1 was based on some elements of Objectory [2], where use case modelling was applied on network and node level, while “true” object-oriented modelling with inheritance was not applied. Major revisions were made in FRED v 0.2, and the notion of a recursive process and a hierarchical component model were introduced. Project 2 applied two recursions before handing over the results to traditional development, which resulted in a real product. Project 3 used the results of Project 2 and continued with a new increment of the same product, and (in parallel with a normal project) carried out three recursions, reaching the block level (as in Figure 4). The reason for Project 3 not being a “sharp” project, was that the next release was delayed due to synchronisation with other projects and method development had either to wait or continue in parallel.

After each project, the experiences where documented and the method documentation was adjusted according to the findings of each project. After Project 3, the method was productified and released internally at Ericsson, and comprehensive method documentation were created, including work instructions, modelling guidelines and selected examples from Project 2 and 3.

The evaluation projects organisations included a project leader, a methodology team, and a product development team. The participating persons were located in the same building, to make continuous interaction between teams easy.

The main tasks of the *product development team* was to:

- use FRED on a new feature of AXE, and suggest method improvements,
- use a tool, specifically tailored for FRED, and suggest tool improvements,
- coordinate with related projects that use traditional methods,
- produce test specifications from the output of FRED.

The main tasks for the *methodology team* was to:

- develop the FRED method,
- support the product development team in the use of the method and the tool,
- handle dependencies with other methods and tools.

The development team held regular meetings at least two times a week. Each team member had a detailed work plan to follow, defining what to do each day. The methodology team had regular meetings once a week to follow the progress of the design team. A number of meetings together with the design team was held, in order to sort out problems and decide on how to cope with unanticipated situations.

Evaluation Criteria. The projects were designed to test the following hypothesis:

H1. It is possible to apply use case modelling in high-level design.

H2. Use case modelling can support requirements traceability.

H3. It is possible to model distribution of requirements with use cases.

5.2 Evaluation Conclusions

The hypotheses H1-H3 were not falsified by Project 1-3. It was possible to apply use case modelling on real features introduced in the AXE system. With the use of a CASE tool, specifically tailored for FRED, it was possible to trace requirements from high-level, black-box use cases to low-level target components. The high-level dynamic behaviour, specified by functional requirements and black-box use cases, was possible to model with white-box use cases expressed using Message Sequence Charts.

The total effort of the latter two projects, including some method development activities, were 7700 person-hours for Project 2, and 1800 person-hours for Project 3. These hours were distributed between the following activities:

Table 1. Distribution of person-hours in percentage over different activities.

Activity	**Fraction**
Product development	32%
Inspections and meetings	12%
Investigation of methods and tools	14%
Preparation of methodology documents	14%
FRED Support	9%
Project management	11%
Training courses (preparation and execution)	6%
Unspecified	2%

No measurements of productivity were undertaken, but the subjective opinion of project leaders was that it was similar to company standards, when adjusted by factors of staff inexperience with the new methodology.

Other subjective conclusions are reported in the following.

Work parallelisation. It was concluded that FRED allowed earlier parallelisation of work, as interfaces between components are well defined at an early stage. The low-level design of blocks can be made independently based on the distribution model. This opportunity may result in shorter lead-time.

Fragmentation of use cases. In Project 1, it was concluded that the way use cases are defined is crucial. Project 1 had great problems with a highly fragmented use case models, where more than 100 use cases were produced. Many use cases only comprised a small part of the functionality required by each actor. The "uses" and "extends" relations [2] was difficult to apply and increased fragmentation. It was found that there was a difficult trade-off between a separate description of reusable parts of use cases and self-contained, comprehensive use cases. In Project 2 and 3, greater emphasis was put on use case definitions reflecting the goals of actors, making use cases describe a complete goal achievement or an undivided function. This resulted in a significant reduction of the number of use cases. However, this also resulted in some redundancy between use cases. The introduction of the episode concept [5, 24], may be a solution to this problem.

Information-driven process. By applying CASE-tools, it may be possible to centre the process around different types of information elements in one information model instead of around a collection of documents. The development may focus on the *refinement* of an information model, rather than on the production of loosely connected documents. With the aid of tools, documents may be *generated automatically* from FRED models, according to pre-defined templates, when needed, for example, as input to inspections or decision meetings. In Project 3, a document called "Implementation Proposal" was, in part, generated automatically from the models of FRED, and sucessfully used as input to project planning and effort prediction.

If a transition from a document-driven process to an information-driven process is made, it is necessary to be able to relate to existing documents in the design base. The document generation facility may be one way to recreate changed parts in existing documents.

Reverse engineering of the design base. Project 2 and 3 showed that it was possible to model only those parts of the design base that was affected by the new requirements. Partial models of the system was created on a high abstraction level, making the affected parts of existing functionality easier to understand. A complete reverse engineering [25] of the entire design base is a very large cost, and the ability to migrate the design base incrementally is considered a big advantage.

Maintenance. The results of Project 2 have not yet been put in operation at customers, and consequently no failure reports from operation have been received. It is however believed that the improved ability of tracing requirements from implementation level to high-level models, will have a good impact on maintenance productivity.

Test Case Identification. It was possible to use the models of FRED to identify test cases. By focusing on actors and use cases, test cases were identified based on different flow variants of use cases, taking both successful, unsuccessful, end error cases into account. Expected results of test cases were identified based on signals in MSC diagrams. The different recursion levels were used to identify test cases at different abstraction levels.

6. Conclusions and Future Work

The FRED method combines use case modelling with a hierarchical component model to support requirements distribution. The transition from requirements to design is viewed in a system evolution context, where new requirements on an existing architecture are distributed through the modelling of affected parts only.

FRED applies a recursive process, where the activities of Functionality Specification, Component Specification, and Functionality Distribution are carried out recursively on different abstraction levels.

In conclusion, the main objectives of FRED are:

1. Improved modelling on a high abstraction level.
2. Improved methods for requirements distribution modelling.
3. Improved requirements traceability.
4. Early definition of interfaces between components, allowing parallelisation of work.
5. Coherent assignment of responsibilities to architectural components.

The FRED method has been developed and evaluated through three pilot projects in the domain of cellular switching systems. Real requirements concerning new features of a telecommunication system were distributed on an existing architecture.

The evaluation of FRED concluded that it was possible to apply use cases in the modelling of requirements distribution, with respect to the realistic requirements studied in the pilot projects. No quantitative measurement were conducted, but the subjective conclusions from pilot studies suggest that FRED improves requirements traceability and system understanding, in comparison with current methods applied in AXE development at Ericsson Radio Systems.

The method has been productified and released internally at Ericsson, and more pilot projects are planned to further assess and improve FRED.

In future developments of FRED, the issue of reuse between use cases will be studied. The concept of *episodes* (coherent parts of use cases) [24], and *operators* [26] (e.g., alternative, repetition, and exceptions) will be considered, as a support for describing how parts of use cases are related.

Other issues of future work are:

1. Applying use case models as input to *statistical usage testing* [27]. How can we select a set of test cases that properly reflects operational conditions?
2. Formulating *tool support* requirements, based on pilot studies conclusions. Which tool is the most appropriate in supporting FRED?
3. *Project management* in recursive, information-driven processes. How to best plan and control FRED projects?
4. *Step-wise introduction* of FRED in traditional processes, starting with a "light" version of the method, that do not require a process revolution and special tool support.

A major challenge is quantitative assessment of the virtues of the proposed method and to evaluate FRED in formal case studies with focus on system quality, time-to-market, and development effort.

Acknowledgements

Many people are and have been involved in the development and evaluation of FRED. The authors would like to give a special acknowledgement to Stefan Sandh, who is one of the main architects behind FRED and has carefully reviewed this paper. Thanks also to all people at ZeLab, Ericsson Radio Systems, Linköping, and the REME project. Special thanks to Kristian Sandahl, Claes Wohlin, and Erik Linderoth-Olson for their useful comments. The academic participation is partly funded by National Board of Industrial and Technical Development (NUTEK), Sweden, Ref. no. 93-2850.

References

[1] Lubars, M., Potts, C., Richter, C., "A Review of the State of Practice in Requirements Modelling", *IEEE International Symposium on Requirements Engineering*, Los Alamitos, USA, pp. 2-14, 1993.

[2] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., *Object-Oriented Software Engineering- A Use Case Driven Approach*, Addison-Wesley, 1992.

[3] Rumbaugh, J.,"Getting Started - Using Use Cases to Capture Requirements", *Journal of Object-Oriented Programming*, September 1994.

[4] Gough, P., Fodemski, F., Higgins, S., Ray, S., "Scenarios - an Industrial Case Study and Hypermedia Enhancements", *IEEE Second International Symposium on Requirements Engineering*, York, UK, March 1995.

[5] Potts, C., Takahashi, K., Anton, A., "Inquiry-Based Requirements Analysis", *IEEE Software*, March 1994.

[6] Hsia, P., Samuel J., Gao J., Kung, D., "Formal Approach to Scenario Analysis", *IEEE Software*, March 1994.

[7] Regnell, B., Kimbler, K., Wesslén, A., "Improving the Use Case Driven Approach to Requirements Engineering", *IEEE Second*

International Symposium on Requirements Engineering, York, UK, March 1995.

[8] Rook, P., "Controlling Software Projects", *Software Engineering Journal*, Vol. 1, No. 1, pp. 7-16, Jan. 1986.

[9] Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, 12(3), pp. 182-221, 1976.

[10] Lindvall, M., *A Study of Traceability in Object-Oriented Systems Development*, Licentiate Thesis 462, Dep. of Computer and Information Science, Linköping University, Sweden 1994.

[11] Wirfs-Brock, R., et. al., *Designing Object-Oriented Software*, Prentice Hall, 1990.

[12] Chen, P., "The Entity-Relationship Model - Toward a Unified View of Data", *Transactions on Database Systems*, no. 1, pp. 9-36, 1976.

[13] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[14] Message Sequence Chart (MSC), *ITU-T Recommendation Z.120*, International Telecommunication Union, 1993.

[15] Loucopoulos, P., Karakostas, V., *System Requirements Engineering*, p. 12, McGraw-Hill Book Company, UK, 1995.

[16] Davis, A. M., *Software Requirements - Objects, Functions and States*, p. 180, Prentice Hall, 1993.

[17] Wieringa, R., *Requirements Engineering - Frameworks for Understanding*, p. 20, John Wiley & Sons, 1996.

[18] Specification and Description Language (SDL), *ITU-T Standard Z.100*, International Telecommunication Union, 1992.

[19] Stevens, W. P., Myers, G. J., Constantine, L. L., "Structured Design", *IBM Systems Journal*, no. 13(2), pp. 115-139, 1974.

[20] Dijkstra, E.W., “Notes on Structured Programming”, in Dahl, Dijstra, Hoare, eds., Structured Programming, Academic Press, pp. 1-82, New Yourk 1972.

[21] Wirth, N., “Program Development by Stepwise Refinement”, *Communications of ACM*, no 14(4), pp. 221-227, Apr. 1971.

[22] Kitchenham, B., Pickard, L., Pfleeger S., “Case Studies for Method and Tool Evaluation”, *IEEE Software*, pp. 52-62, July 1995.

[23] Wohlin, C., Gustavsson, A., Höst, M., Mattsson, C., “A Framework for Technology Introduction in Software Organizations”, *International Conference on Software Process Improvement*, Brighton, UK, 1996.

[24] Regnell, B., Andersson, M., Bergstrand, J., “A Hierarchical Use Case Model with Graphical Representation”, *IEEE International Symposium and Workshop on Engineering of Computer-Based Systems*, Germany, March 1996.

[25] Sommerville, I., *Software Engineering*, p. 701, 5th Ed., Addison-Wesley, 1996.

[26] Regnell, B., *Hierarchical Use Case Modelling for Requirements Engineering*, Licentiate Thesis no. 120, Dep. of Communication Systems, Lund University, Sweden 1996.

[27] P. Runeson and C. Wohlin, “Statistical Usage Testing for Software Reliability Control”, *Informatica*, vol. 19, no 2, 1995.

Paper VII:

A Market-driven Requirements Engineering Process
- Results from an Industrial Process Improvement Programme

Björn Regnell, Per Beremark and Ola Eklundh

Published in Journal of *Requirements Engineering* 3:121-129, Springer-Verlag, 1998

Abstract

In market-driven software evolution, the objectives of a requirements engineering process include the envisioning and fostering of new requirements on existing packaged software products in a way that ensures competitiveness in the market place. This paper describes an industrial, market-driven requirements engineering process which incorporates continuous requirements elicitation and prioritisation together with expert cost estimation as a basis for release planning. The company has gained a measurable improvement in delivery precision and product quality of their packaged software. The described process will act as a baseline against which new promising techniques can be evaluated in the continuation of the improvement programme.

1. Introduction

Requirements Engineering (RE) for packaged software is different from RE for bespoke software. In tender projects, the customer is well-defined and the requirements specification often acts as a contract between the developer and the customer. When developing packaged software for a market place, the RE process should be able to invent requirements based on foreseen end-user needs and select a set of requirements resulting in a software product which can compete on the market. A packaged software product, sometimes called COTS (Commercial off-the-shelf) software, is often an integration of components. The product with its components is evolved in releases, with each release including new and improved features that, hopefully, ensure that the vendor stays ahead of competitors.

This paper describes a specific industrial RE process for packaged software, called REPEAT (Requirements Engineering ProcEss At Telelogic), which is enacted by the Swedish CASE-tool vendor Telelogic AB; a fast growing company, currently with 180 employees, more than 600 customers world-wide, and a predicted revenue for 1998 of circa 200 million SEK (increase from 107 million SEK, 1997).

REPEAT is used in-house at Telelogic for eliciting, selecting and managing requirements on a product family called Telelogic Tau; a software development environment for real-time systems, used by many of the world's largest telecommunication systems providers in their software development. Telelogic Tau supports standardised graphical languages, such as SDL [1], MSC [2], TTCN [3], and UML [4], and provides code generators for integration with several real-time operating systems[1].

Telelogic Tau is an integration of in-house developed COTS components and can be tailored for the specific needs of a customer or a market segment, and is available on UNIX and MS-Windows platforms. It is built using an architecture with an implicit invocation style [5], which enables changes with local impact.

The paper is structured as follows. Section 2 describes the current and first version of the RE process denoted REPEAT-1. The lessons learned from three subsequent enactments are used for the continuation of the RE process improvement program at Telelogic aiming at the definition of REPEAT-2. In this work, REPEAT-1 will act as a *baseline* against which process improvement proposals can be evaluated in case studies. The past experiences of the REPEAT process improvement programme are concluded in Section 3, together with proposals regarding its continuation.

1. More information on Telelogic Tau can be found at http://www.telelogic.se

2. REPEAT: A Market-Driven Requirements Engineering Process

REPEAT is an RE process that manages requirements throughout a whole release cycle. It covers typical RE activities [6], such as elicitation, documentation, and validation, and has a strong focus on requirements selection and release planning. Management of requirements changes due to, e.g., new market demands, is an important function.

The actors involved in REPEAT include:

- *Requirements Management Group* (RQMG). This group is responsible for requirements management, and makes decisions on which requirements to implement. It is also responsible for requirements change management. RQMG includes, among others, product and project managers together with the quality manager.
- *Issuer.* Any employee at Telelogic can submit a requirement to RQMG. An issuer is usually a person from marketing & sales or customer support, but can also be e.g. a developer or a tester.
- *Customer*s & *users* provide input and feedback to an issuer regarding user and market needs.
- *Requirements Team*. A team with the responsibility of analysing and specifying a set of requirements. RQMG has several of these teams at their disposal. A requirements team is cross-functional and includes persons participating in implementation, testing, marketing & sales, and customer support.
- *Construction Team*. A team with the responsibility of designing and implementing a set of requirements.
- *Test Team*. A team with the responsibility of verifying a set of requirements.
- *Expert*. A person that is assigned to evaluate a specific requirement in depth, concerned with e.g. cost estimation and impact analysis.
- *Requirements Database* (RQDB). All requirements are stored in this in-house-built database system. RQDB has a web-interface that can be accessed by Telelogic employees from a multi-continent intranet.

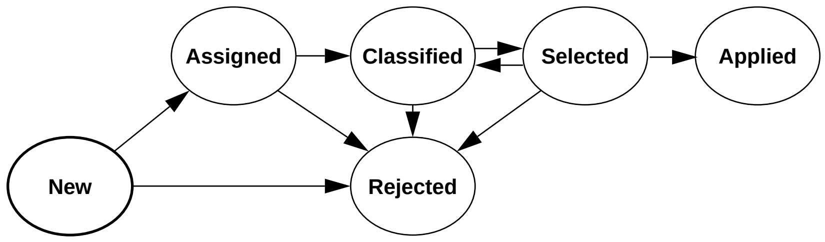


Figure 1. *The states of a requirement in the REPEAT process.*

Elicitation is continuously active, and a requirement can be issued at any time by an issuer that foresees a market need. Each requirement is stored in RQDB as an entity described in natural language with unique identity. Throughout the continuous enactment of REPEAT process instances, each unique requirement has a life-cycle progressing through specific states as shown in Fig. 1.

The semantics of the states are explained below. The RQMG with support from experts is responsible for deciding on requirement state transitions.

- *New*. The initial state of a requirement after it has been issued and given an initial priority.
- *Assigned*. The requirement has been assigned to an expert for classification.
- *Classified*. A rough estimate of cost and impact is attached to the requirement. Comments and implementation ideas may also be stated.
- *Rejected*. An end-state indicating that the requirement has been rejected, e.g. because it is a duplicate, already implemented, or it does not comply with the long-term product strategy.
- *Selected*. The requirement has been selected for implementation with a certain priority attached to it combined with results from detailed cost and impact estimations. There is also a more detailed specification of the requirement available. A selected requirement may be de-selected, due to requirements changes, and then re-enters the classification state or gets rejected.
- *Applied*. An end-state indicating that the requirement has been implemented and verified.

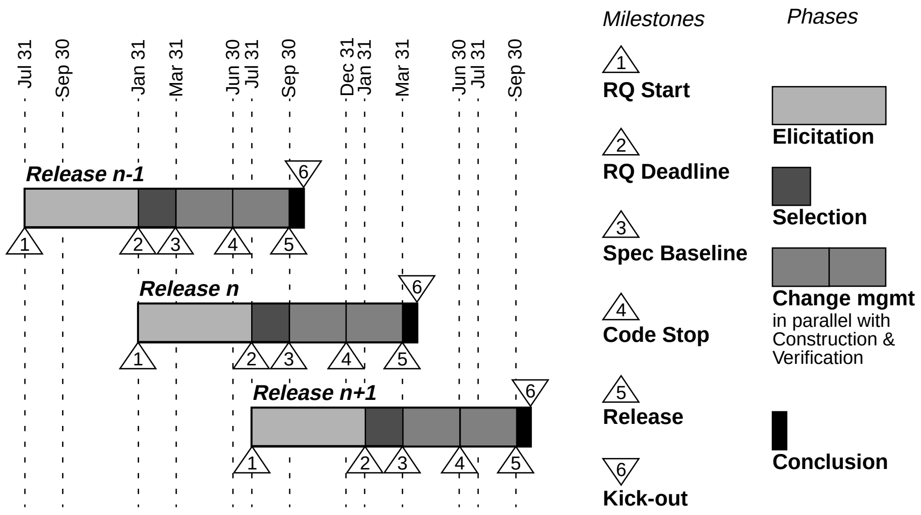


Figure 2. *The milestones, phases, durations and parallelisation of REPEAT process instances.*

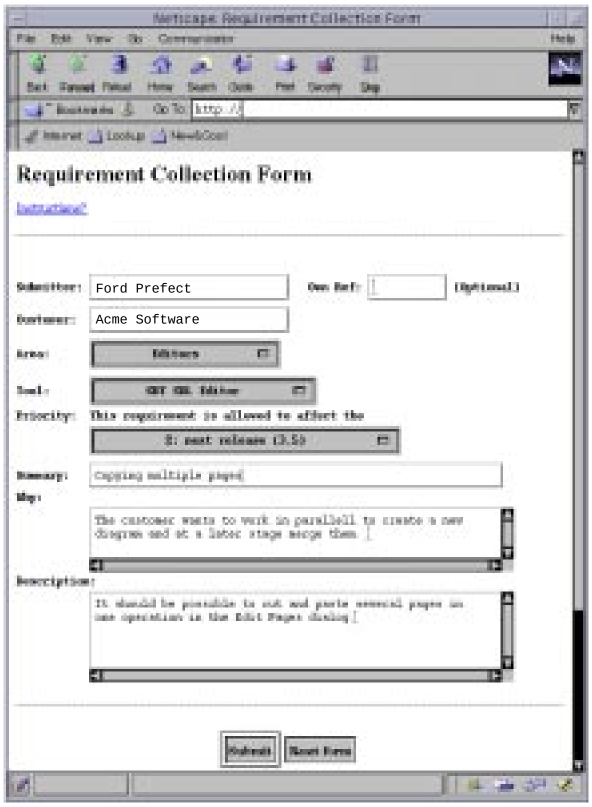
REPEAT is instantiated for each release, and each process instance has a fixed duration of 14 months. A new product version is released at fixed dates every sixth month, which implies that different REPEAT instances overlap with at most three simultaneous enactments, as shown in Fig. 2. The REPEAT process instance *n* denotes the *current* release project. Each REPEAT instance consists of five different phases separated by milestones at pre-defined dates. The different phases are described subsequently.

2.1 Elicitation Phase

The elicitation phase includes two activities: *collection* and *classification*. Collection of requirements is made by an issuer that fills in a web-form and submits the requirement for storage in RQDB (see Fig. 3).

Requirements are described using natural language and given a summary name by the issuer. An explanation of *why* the requirement is needed is also given. The issuer gives the requirement an initial priority P, which suggests in which release it may be implemented. P is a subjective measure reflecting the view of the issuer, and is measured on an ordinal scale with three levels, as shown in Table 1.

The requirement is initially in the *new* state, and a first check is made by RQMG to see if it is detailed enough; if not it is returned to the issuer for clarification of its description.


Netscape: Requirement Collection Form
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Go To: http://
Internet Lookup New&Cool

Requirement Collection Form

Instructions?

Submitter: Ford Prefect
Own Ref: (Optional)
Customer: Acme Software
Area: Editors
Tool: SDT SDL Editor
Priority: This requirement is allowed to affect the
3: next release (3.5)
Summary: Copying multiple pages
Why: The customer wants to work in parallell to create a new diagram and at a later stage merge them.
Description: It should be possible to cut and paste several pages in one operation in the Edit Pages dialog

Submit Reset Form

Figure 3. *The web-form for issuing requirements that are stored in the RQDB.*

Table 1. The ordinal scale of the priority P.

Level	**Semantics**
1	The requirement is allowed to impact on-going construction.
2	The requirement is incorporated in the current release planning.
3	The requirement is postponed to a later release.

When a new requirement has arrived, RQMG issues a classification of the requirement by assigning it to an expert. The expert classifies the requirement by assigning to it a rough estimate of its cost (C) and impact (I). The cost estimate C is given on an ordinal scale of implementation effort from 1 to 5, as shown in Table 2.

The impact estimate I is given to assess how many architectural components that are affected by the requirement. The I measure is given on an ordinal scale from 1 to 5 as shown in Table 3.

Table 2. The ordinal scale of the cost estimate C.

Level	Semantics
1	Less than 1 day.
2	Less than 5 days.
3	Less than 5 weeks.
4	Less than 3 months.
5	More than 3 months.

Table 3. The ordinal scale of the impact estimate I.

Level	Semantics
1	Impact is isolated to one component.
2	A few components are impacted.
3	Less than half of all components are impacted.
4	More than half of all components are impacted.
5	Nearly all components are impacted.

The expert also reconsiders the priority P and may recommend RQMG to change P. Further comments and implementation ideas may also be given.

The classification (i.e. estimating C and I, and reconsidering P) should take about 15-30 minutes. If more effort is needed, the expert should recommend the RQMG to issue a pre-study, where the requirement can be decomposed to more fine-grained requirements that are easier to classify.

When the priority P, cost estimate C, and impact estimate I, have been given, the requirement enters the *classified* state, and will be further treated when the subsequent selection phase is started.

2.2 Selection Phase

The goals of this phase are: (1) to select which requirements to implement in the current release, (2) to specify the selected requirements in more detail, and (3) to validate the requirements document.

The output of this phase is a Requirements Document (RD) which includes a selected-list, a detailed specification of all selected requirements, and a not-selected-list including the requirements that are postponed to the next release (see Fig. 4).

In the RD, there are a total of *m*+*w* requirements in a selected-list, and a total of *n* requirements in a not-selected-list. The selected-list is divided

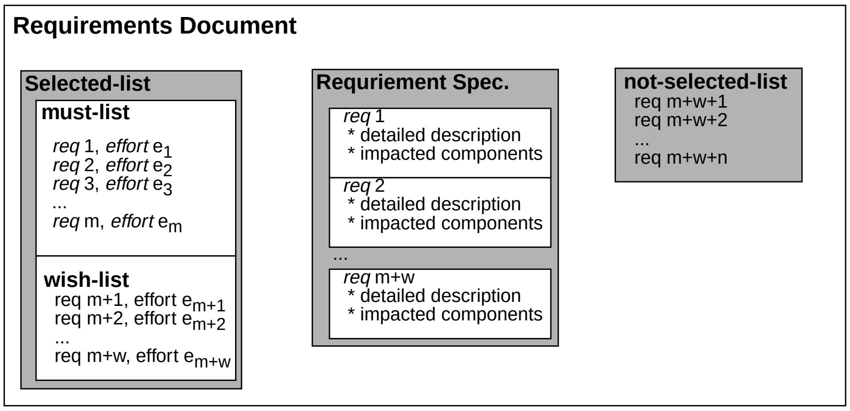


Figure 4. *The requirements document including a list of selected requirements sorted in priority order.*

into two parts: *m* requirements in the must-list and *w* requirements in the wish-list. The select-list is sorted in priority-order.

For each requirement *i* on the selected-list, a detailed effort estimation e_i is given, measured on a ratio-scale of hours. Given that there is a total effort of *E* hours available for implementing the planned release, the *selection-rule* given in Fig. 5 must be fulfilled by the RD.

For all req on the selected-list:	For all req on the must-list:	For all req on the wish-list:
$\left(\sum_{i=1}^{m+w} e_i\right) \le 1.3E$	$\left(\sum_{i=1}^{m} e_i\right) \le 0.7E$	$\left(\sum_{i=m+1}^{m+w} e_i\right) \le 0.6E$

Figure 5. *The selection-rule.*

The selected requirements are estimated to take 130% of the available effort *E*. The must-list comprises 70% of *E* (i. e. a 30% risk buffer) and the wish-list comprises 60% of *E*. This implies that up to half of the wish-list will be implemented.

The effort estimation and detailed specification of all selected requirements are made by requirements teams, and more effort is put on specifying the high-priority-requirements. The sorting of the selected requirements in priority order, is made by RQMG with support from the requirements teams, using the P, C, I, and e_i measures and the detailed specifications as input information.

When the RD is completed, it is validated in an inspection before it is put in the Specification Baseline. The not-selected-list is used in the validation of the RD so that no requirements are unintentionally omitted. The not-selected requirements are in state classified, and are normally placed in the selected-list in the RD of the next release.

2.3 Change Management, Construction, Verification and Conclusion

After Specification Baseline, the REPEAT process instance enters the Change Management Phase. When this happens a new REPEAT process instance is started in the Elicitation Phase (see Fig. 2). During change management the RQMG takes decisions on changing the RD caused by new incoming requirements with P=1, i.e. high-priority requirements that are suggested to impact the current *development process* (including construction and verification) running in conjunction with the change management phase of REPEAT.

When the RD is changed, and a new requirement is allowed to enter the must-list, the selection-rule in Fig. 5 must still hold, and a set of requirements amounting to the same effort as the new requirement must be de-selected. The new requirement is inserted by RQMG at a position in the selected-list that reflects its decided priority. Feedback is given to the issuer on the decisions taken to the change request.

The Code Stop milestone separates *construction* from *verification*. Construction is made using an iterative design and implementation process with a weekly build and unit test. In the verification activity, the requirements in the selected-list that where really implemented are verified against the RD using a requirements-based testing method. When the implementation is correct with respect to RD, the new release is delivered to marketing & sales and the implemented requirements enter the applied state. A Conclusion Phase is then entered, where metrics are collected and a final report is written that summarises the lessons learned from this REPEAT enactment.

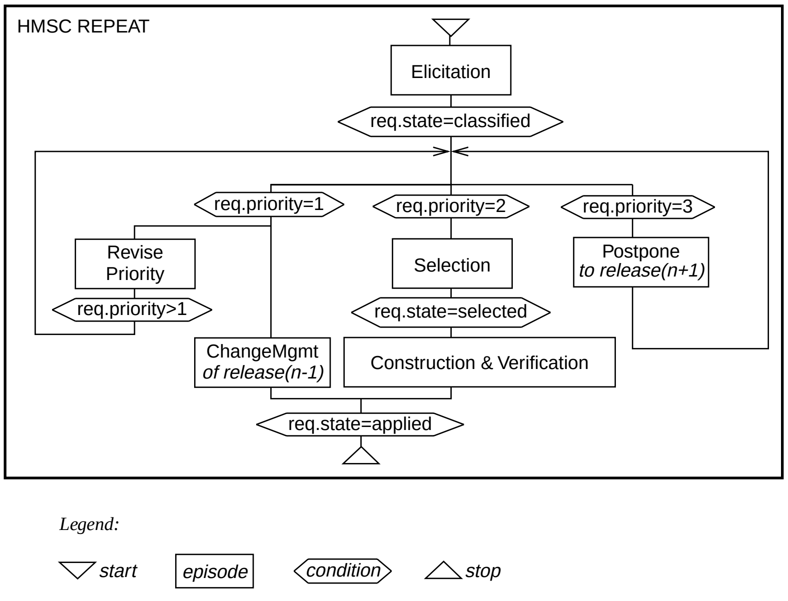


Figure 6. *Different ways of handling a requirement depending on its classification. (Time progresses down-wards.)*

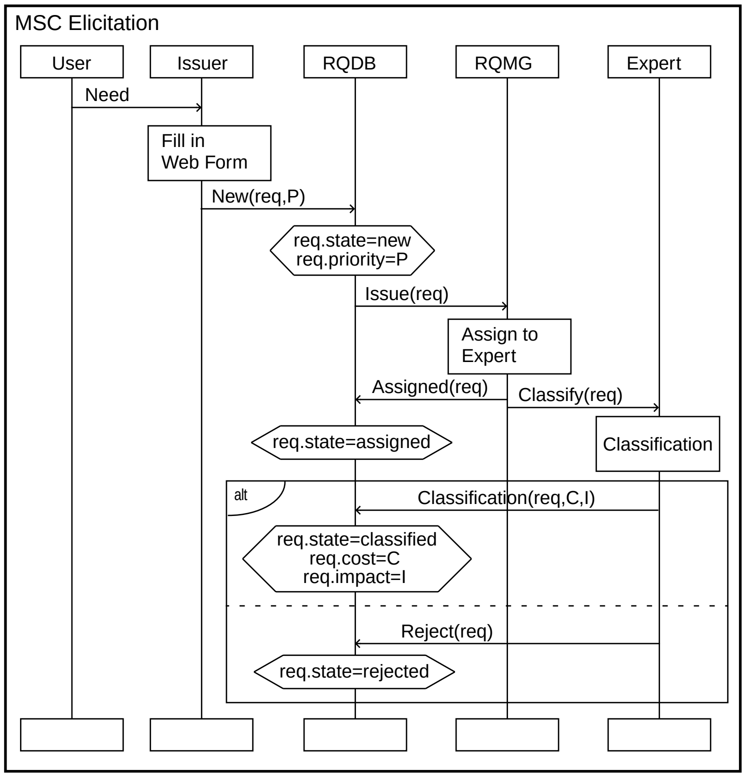
2.4 Some Process Enactment Scenarios of REPEAT

To further explain how REPEAT is enacted in its different phases, we present a partial process scenario model, using Message Sequence Charts (MSC) [2]. Fig. 6 shows a High-level MSC (HMSC) [8], that describes the events related to *one* requirement.

Assume that we are in the elicitation phase of the current release *n*, and we issue a requirement *req* that is classified according to the classification scheme described in Section 2.1. In Fig. 7, a typical process scenario for an elicitation episode is depicted. (The case where *req* is rejected before it is assigned is not included, c.f. Fig. 1.)

When the RQ deadline is reached (see Fig. 2) and the elicitation phase is ended the priority of *req* determines which release it will affect. Thus, P suggests to which release it should be “routed” for further treatment.

If *req.priority*=1 then release *n*-1 may incorporate *req* in its change management phase. As it is rather expensive to incorporate late changes, it is not unusual to enact the Revise Priority episode, so that ongoing construction is unaffected (see Fig. 6).



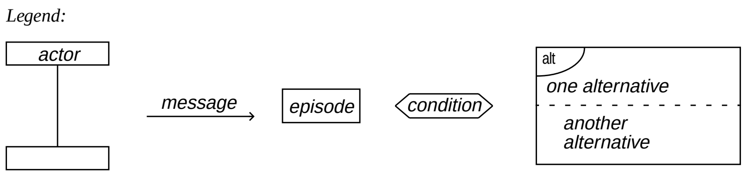


Figure 7. *A partial description of the events in the elicitation phase of REPEAT. (Time progresses down-wards.)*

If a change management of release *n*-1 is enacted, different actions are taken depending on how far release *n*-1 has reached, as shown in Fig. 8.

If the REPEAT process of release *n*-1 is in the selection phase (i.e. pre Specification Baseline), the Change Selection episode is enacted where *req* is allowed to change the selected-list. This change is not so expensive as the other case, where the Specification Deadline milestone is passed. Then *req* implies that a Change Construction episode is enacted, causing expensive re-design and, if Code Stop is reached, re-testing.

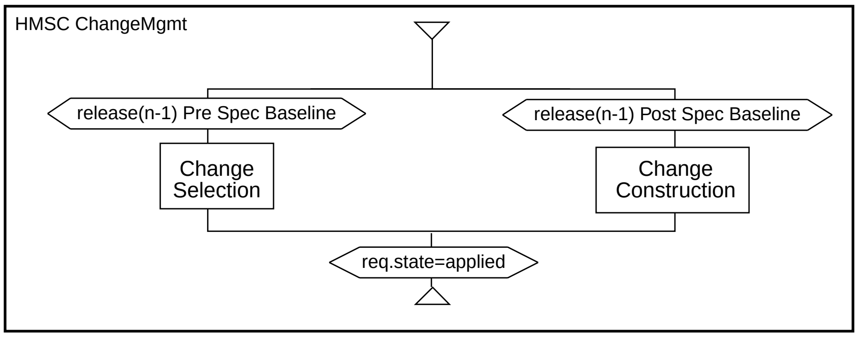


Figure 8. *Different change episodes depending on the advance of release n-1.*

If *req.priority*=2 then the treatment of *req* will continue with the selection phase in release *n*, where *req* is specified in more detail and subjected to detailed effort and impact estimation. By the end of the selection phase, the must-list and wish-list are constructed (as described in Section 2.2) including *req* at its decided priority level. When req is on the must- or a wish-list, it is in the *selected* state and when it has been implemented and verified it is in the *applied* state.

If *req.priority*=3 then *req* is kept in the *classified* state and postponed to a later release, where its classification (P, C, and I) is reconsidered. If *req* at this stage is given priority 1 or 2 it will eventually change its state to *selected*, and at some future stage become *applied*.

3. Conclusions from the REPEAT Process Improvement Programme

During 1995, Telelogic realised that they needed a repeatable and defined RE process and the work started on the formulation of an RE Process Improvement Programme resulting in REPEAT-1.

Prior to REPEAT-1, Telelogic had an ad hoc process for managing requirements and faced a number of challenges related to release precision and product quality. Version 3.0 of the product family was released 8 months later than planned, and version 3.1 was released with a 3 months delay. Between 3.0 and 3.1, seven intermediate releases were needed in order to mend quality problems and add on extra requirements. In May 1995, a CMM assessment [7], conducted by an external software engineering consultancy, concluded that very few of the Key Processes Areas of CMM-Level 2 were in place.

REPEAT-1 was introduced in January 1996. In February 1998, a second CMM assessment showed that Telelogic had almost all Key Processes Areas of CMM-Level 2 in place. When REPEAT-1 was applied, product version 3.2 was released with a small delay of 15 days, and the subsequent version 3.3 was released three days *ahead* of schedule. The current version 3.4 under construction is to date on schedule, and is predicted to be released on time. The product quality has increased as indicated by the monotonic decrease of reported failures in operation measured from version 3.1 to 3.3. Almost no requirements were unintentionally missed in the latest two versions. The authors are convinced that the introduction of REPEAT-1 is the major explanation for these achievements.

The major elements of REPEAT-1 that are believed to cause the dramatic improvements in release precision and product quality, are the prioritisation of requirements, the effort estimation, the detailed requirements specification, and the continuous change management throughout design, implementation and verification. The classification activity gives experts the opportunity to carefully consider which requirements to be implemented in which release, so that the requirements that are believed to give the highest value to the lowest cost are implemented first. The must-, and wish-lists are strong tools for enforcing that a release project does not take in more requirements than can be achieved within 6 months. Customer support and marketing can easily issue requirements as a reaction to their observation of end-user and market needs.

However, a number of challenges have been identified in the past enactment of REPEAT. Some of these challenges are outlined below:

- *Overload control*. With the web-interfaced requirements database, it is very easy to issue new requirements. Every new requirement has a cost, even if its never implemented. Requirements that are in state classified must eventually be either applied or rejected. Currently, the number of classified requirements in RQDB is increasing for every release, which is about to cause REPEAT-1 to be overloaded. A mechanism is needed to avoid congestion in the RE process.
- *Connecting fragments*. The requirements entities are not related to each other. They are only grouped in relation to implementation components. Requirements fragments need to be packaged into coherent bundles, in order to give them a structure that reflects the functionality as seen by the user. This is necessary for managing dependencies and making prioritisation of sets of requirements.
- *Bridging the chasm between elicitation and selection*. Related to the above challenges, the authors have made the qualitative observation that the performance of REPEAT-1 is low in the gap between elicitation and selection (see Fig. 9), due to congestion caused by too many incoming, unrelated requirements fragments with sometimes poor description quality. The requirements fragments are described at very different levels of abstraction and classification gets difficult.
- *Long-term product strategy for a diversity of market segments*. As REPEAT-1 triggers on the issuing of new requirements, RE becomes to some extent reactive rather than pro-active. There is a foreseen need of promoting activities related to the existing long term product strategy and prioritisation in relation to a range of market segments.

During the continuation of the REPEAT Process Improvement Programme, REPEAT-1 will act as a *baseline* against which promising techniques, that are believed to meet the above challenges, can be evaluated using expert surveys, case studies and experiments [9]. Two techniques that are candidates for introduction in REPEAT-2 are:

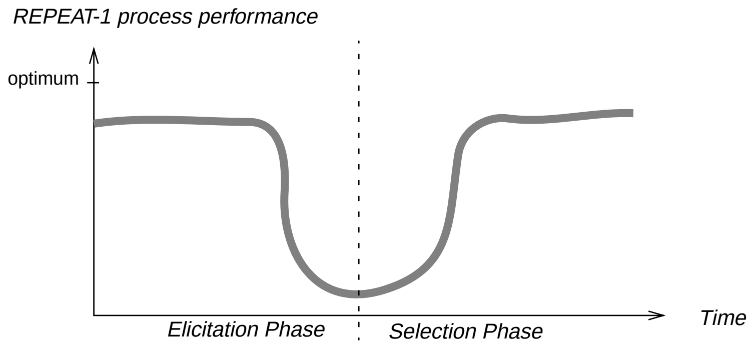


Figure 9. *An informal depiction of the REPEAT process performance challenge of bridging the chasm between elicitation and selection.*

- *Hierarchical use case modelling* [8, 10, 11]. A hierarchy of informal or semi-formal use cases may help to connect requirement fragments and provide an integrated model of the product's "functionality architecture" as seen by its users. Hopefully, a long-term product strategy with the priorities of different market segments can be integrated with such a use case model.
- *Cost-value use case prioritisation*. In order to increase process performance and avoid congestion, a more efficient approach to the sorting of requirements is need. Currently the selection is made using expert judgement. A smart grouping of requirements based on the use cases to which they are related, combined with a systematic cost-value prioritisation approach [12, 13] applied to use cases instead of single requirements, may speed up the selection process.

Acknowledgements

The authors would like to give a special acknowledgement to Anders Ek at Telelogic, who participates in the development of REPEAT. Thanks also to Per Runeson, Lars Bratthall, and Claes Wohlin at Dept. of Communication Systems, who have carefully reviewed this paper. The academic participation in this work is partly funded by the National Board of Industrial and Technical Development (NUTEK), Sweden, grant 1K1P-97-09690.

References

1. Specification and Description Language (SDL), *ITU-T Standard Z.100*, International Telecommunication Union, 1992.

2. Message Sequence Chart (MSC), *ITU-T Recommendation Z.120*, International Telecommunication Union, 1996.

3. Tree and Tabular Combined Notation (TTCN), ISO/IEC Recommendation 9646-3, International Standardisation Organisation, 1992.

4. Fowler, M., Scott, K., *UML Distilled - Applying the Standard Object Modelling Language*, Addison-Wesley, 1997.

5. Shaw, M., Garlan, D., *Software Architecture - Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

6. Sommerville, I., Sawyer, P., *Requirements Engineering - A Good Practice Guide*, Wiley, 1997.

7. Humphrey, W. S., *Managing the Software Process*, Addison-Wesley, 1989.

8. Regnell, B., Andersson, M., Bergstrand, J., "A Hierarchical Use Case Model with Graphical Representation", *IEEE International Symposium and Workshop on Engineering of Computer-Based Systems*, Germany, March 1996.

9. Wohlin, C., Gustavsson, A., Höst, M., Mattsson, C., “A Framework for Technology Introduction in Software Organizations”, *International Conference on Software Process Improvement*, Brighton, UK, 1996.

10. Regnell, B., Kimbler, K., Wesslén, A., “Improving the Use Case Driven Approach to Requirements Engineering”, *IEEE Second International Symposium on Requirements Engineering*, York, UK, March 1995.

11. Regnell, B., *Hierarchical Use Case Modelling for Requirements Engineering*, Licentiate Thesis No. 120, Dept. of Communication Systems, Lund University, Sweden 1996.

12. Karlsson, J., Ryan, K., “A Cost-Value Approach for Prioritizing Requirements”, *IEEE Software*, September/October 1997.

13. Karlsson, J., *A Systematic Approach for Prioritizing Software Requirements*, Doctoral Dissertation No. 526, Dept. of Computer and Information Science, Linköping University, Sweden, 1998.