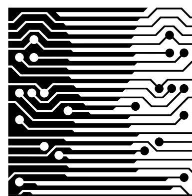
IMPROVING REQUIREMENTS ENGINEERING: AN ENHANCED REQUIREMENTS MODELLING AND ANALYSIS METHOD

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Ksenia Ryndina
March 2005

Supervised by
Professor Pieter Kritzinger

University of Cape Town



The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

© Copyright 2005
by
Ksenia Ryndina

Abstract

Inadequate requirements engineering is considered to be one of the top causes for software development project failure today. One of the major problems is the lack of processes, techniques and automated tool support available to developers for specifying requirements. We thus set out in our research to improve requirements specification methodology by enhancing the approach that is most popular at the moment - use case modelling. Despite their popularity, use case models lack structure and precision, which makes formal analysis of such models impossible. In our proposal, we amend traditional use case models with formal structure and semantics to make them suitable for automated analysis.

The enhanced use case modelling method that we propose is called Susan ("S"ymbolic "us"e case "an"alysis), which facilitates analysis of use case models using model checking. We also developed a software tool called SusanX to construct, manipulate and analyse Susan models. The analysis feature of the tool is implemented using the publicly available NuSMV model checker, which allows verification of finite state systems for behavioural properties expressed in temporal logic. A number of generic properties that can be used for verification of any Susan model are built into the SusanX tool. Additionally, SusanX permits the user to define model-specific properties for verification. This is done through property specification patterns, which allow one to express logic properties without knowing the details of the underlying formalism.

In order to evaluate how valuable Susan and the SusanX tool are in solving real-world problems, we performed a case study of a Cash Management System (CMS). The case study was done in collaboration with an established South African software development company, which provided us with the requirements specifications for the system. We successfully used the Susan notation to model the CMS requirements and performed various analyses on the models with SusanX. The state of the requirements specifications was considerably improved through this process and numerous errors were discovered during the SusanX analyses.

Acknowledgements

The research project resulting in this dissertation has been a challenging and rewarding endeavour. Despite learning a great deal about requirements modelling and analysis, I also acquired skills of much higher importance such as dealing with criticism, time management and communicating ideas professionally. I would like to extend my sincere thanks to the following people for their support that assisted me in completing this work.

- My supervisor, Professor Pieter Kritzinger, for his wise guidance and encouragement throughout the project. Dr Andrew Hutchison, for constructive advice and suggestions during progress meetings.
- Justin Kelleher and the Software Futures team for making the Cash Management System case study possible. A special thanks goes to Monica Murray from Software Futures, for being so efficient with all the feedback and meeting arrangements.
- Fellow student members of the DNA group, for sitting through my numerous progress presentations. It has been a pleasure working with all of you. In particular, I would like to note that Simon Lukell, Nico de Wet, Ben Tobler and Johannes Appenzeller are the best company one can wish for during coffee breaks and SATNAC conferences!
- Other Masters students from the “contending” group - Marshini Chetty, Ilda Ladeira, Sarah Brown, Cara Winterbottom and Barry Steyn. Our support network made it so much easier to cope with the tough days on the way to the finish line.
- Two more friends - Yumna Omar and Gerard Petersen, for keeping me company in the deserted Room 300 during many dark nights spent in front of my computer.
- Finally, my brother Denis, Mom and Dad. For supporting me morally and always having encouraging words in difficult and stressful times.

Contents

Abstract **iii**

Acknowledgements **iv**

1 Introduction **4**

1.1 Synopsis 7

1.2 Original Contribution to the Research Field 9

1.3 Objectives, Scope and Limitations 10

1.4 Dissertation Outline 11

2 Methods for Requirements Modelling and Analysis **13**

2.1 Importance of Modelling and Analysis of Requirements 13

2.2 Notations, Analysis Techniques and Tool Support 15

2.2.1 Notation 15

2.2.2 Analysis technique 16

2.2.3 Tool support 17

2.3 Existing Requirements Methods 18

3 Use Case Modelling **23**

3.1 Fundamentals of Use Case Modelling 23

3.2 Pros and Cons of Use Case Modelling 26

3.3 Illustration of Use Case Modelling 28

3.4 Our Approach to Improving Use Case Modelling 32

3.5 Existing Approaches to Formalising Use Case Modelling 35

4 **Model Checking and NuSMV** **38**
4.1 Fundamentals of Model Checking . . . 38
4.2 Pros and Cons of Model Checking . . . 41
4.3 Existing Model Checkers . . . 43
4.4 NuSMV Input Language and Generated Counter-Examples . . . 44

5 **Susan Models and Verification** **49**
5.1 Susan Metamodel . . . 50
5.1.1 A Simple Susan Model . . . 55
5.1.2 Flattening Use Case Models for Susan . . . 60
5.2 Mapping Susan Models to NuSMV . . . 62
5.3 Overview of Susan Verification . . . 69
5.4 Generic Verification of Susan Models . . . 70
5.4.1 Liveness of Use Cases . . . 70
5.4.2 Reversibility of Conditions . . . 72
5.5 Model-Specific Verification of Susan Models . . . 74
5.5.1 Property Specification Patterns . . . 74
5.5.2 Property Specification Patterns in Susan . . . 76

6 **The SusanX Tool** **80**
6.1 Overview of Features Supported by SusanX . . . 80
6.2 Graphical User Interface of SusanX . . . 82
6.2.1 Building a Model in SusanX . . . 84
6.2.2 Static Checking on a Susan Model . . . 86
6.2.3 Running Batch Generic Verification . . . 88
6.2.4 Using Guided Generic Verification . . . 89
6.2.5 Verifying Model-specific Properties . . . 90
6.3 Implementation Details . . . 92
6.3.1 Architectural Overview of SusanX . . . 93
6.3.2 SusanAnalyser Package . . . 96

7 **Case Study: A Cash Management System** **100**
7.1 Case Study Overview . . . 100
7.2 Modelling and Analysing CMS Requirements with Susan . . . 103

7.3 Stage 1: Examining the Provided Documentation 104
7.4 Stage 2: Modelling and Verifying Use Case Clusters 107
7.5 Stages 3 and 4: Integrating Cluster Models for Each Increment and Integrating Increment Models 113
7.6 Performance Measures for Analysis with SusanX 115
7.7 Summary and Evaluation of Results 116

8 Conclusions and Future Work **119**
8.1 Future Work 120

A Susan Model for CMS **121**

Bibliography **135**

List of Figures

1	Enhanced Method for Requirements Modelling and Analysis	8
2	Relating Elements in Use Case Models	25
3	Use Case Diagram for MuTI Requirements	31
4	Details of Place Call Use Case	33
5	Unwinding a Kripke Structure	39
6	CTL Properties in a Computation Tree	41
7	State Changes of a Traffic Light	45
8	State Changes of Two Synchronised Traffic Lights	46
9	Interaction Between Actor and System in Susan	49
10	Susan metamodel.	51
11	A Simple Use Case Diagram	55
12	Flattened MuTI Use Case Diagram.	61
13	Overview of Verification in Susan	70
14	Dwyer's Property Specification Pattern Hierarchy	75
15	Susan Property Specification Pattern Hierarchy	76
16	Graphical Editor Window of SusanX	82
17	Pull-down Menus and Toolbar in SusanX	83
18	Properties of Condition and Use Case Elements in SusanX	85
19	Static Model Check Window in SusanX	87
20	Batch Generic Verification Window in SusanX	89
21	Guided Generic Verification Window in SusanX	90
22	Model-specific Verification Window in SusanX	91

23 Component View of SusanX 94
24 Package View of SusanX 95
25 Classes in SusanAnalyser Package 97
26 Sequence of Actions During Batch Use Case Liveness Verification 98

27 CMS Architectural Overview 102
28 The Four Stages of the Case Study Process 103
29 (a) Original and (b) Revised Use Case Diagrams for *Manual Receipts* Cluster 109
30 Use Case Diagram for *Manual Receipts* Cluster Flattened for Susan Model . 110

31 CMS Model in SusanX 122

List of Tables

1 Overview of Existing Methods 20
2 Assessment of Existing Methods 21

3 Times for CMS Model Verification with SusanX 115

Chapter 1

Introduction

It is fairly common knowledge that today only one out of every three software development projects is completed successfully. The latest CHAOS Surveys by the Standish Group [sta03] report that 15% of projects fail outright, and 51% are late, run over budget or provide reduced functionality. On average only 54% of the initial project requirements are delivered to the client. Inadequate definition and understanding of project requirements by developers is considered to be one of the main causes for project failure.

Before any software system can be built, it is necessary to establish its exact requirements. System requirements include a precise description of the functionality that the system must provide, as well as the design and implementation constraints imposed by its environment [KS98]. Requirements that define the desired functionality and behaviour of the system are called *functional*, while other qualities and constraints to which the system must conform are given by its *non-functional* requirements [BS03]. In a software development project, a combined definition of functional and non-functional requirements is usually called a *Software Requirements Specification (SRS)* [Kru01]. There are several desirable characteristics for a SRS, the most important of which are unambiguity, correctness, completeness and consistency [IEE84]. Producing a SRS that possesses these characteristics has proven to be a very challenging and yet crucial task. Without a reliable requirements definition accurate project planning is impossible, system design and implementation are bound to contain errors, and the final system cannot be validated to determine whether it satisfies the needs of the client.

For many types of systems, missing or incorrectly implemented functionality is more devastating to the client than the system not conforming to all its non-functional requirements such as performance or usability. For example, for non-real-time systems the client

can usually tolerate the fact that the system response-time is not as quick as was originally desired. In this research we address the issues concerning the handling of functional requirements of software systems. In the rest of the dissertation, we refer to "functional requirements" as simply "requirements" in order to make our discussion more succinct.

A number of interrelated software engineering processes deal with system requirements, which are collectively called *requirements engineering*. The names and definitions of these processes vary slightly from one source to the other, but the most widely used ones are as follows [KS98, NE00, MC92].

- **Requirements elicitation** is concerned with collecting information from the organisation of the client and using it to identify system requirements. This involves studying the application domain and existing systems, establishing the stakeholders for the system and their individual needs, negotiating with them where necessary. Elicitation is the most communication-intensive process of requirements engineering.

- **Modelling and analysis** are used to visualise, structure and verify requirements once they have been elicited from various sources. In software engineering, a *model* is an abstract description of a system from a particular perspective, used to enhance the developers' understanding of that system [BS03]. Techniques for modelling software requirements vary greatly and include informal graphical modelling and rigorous mathematical approaches. Requirements analysis is concerned with reasoning about and examination of the captured requirements to ensure that they fulfil certain desirable properties. Analysis of requirements is essential for identifying missing requirements, inconsistencies and logical errors early in the development cycle.

- **Documentation** of requirements results in a SRS, which is textual but often augmented with models created during the modelling process. In fact, the latest trends in software engineering such as Model Driven Architecture [mda02, Fra03], place models rather than documents at the centre of all the development processes.

- **Requirements management** is necessary for controlling the changes of requirements that almost inevitably occur during the course of any development project. It involves continuous monitoring of requirements right up to the end of implementation of the system. When changes are encountered, they must be reflected in all the development work done up to that point.

All of the requirements engineering processes described above are not yet well understood and are prone to error. Furthermore, related activities such as traceability from requirements to design and implementation also present developers with many challenges. The two requirements processes that are overlooked the most by the industry are modelling and analysis of requirements, and requirements management. In our research we focus on the former - modelling and analysis of requirements.

Several methods for modelling and analysing requirements have been proposed by researchers, but few have been accepted in the industry. Even though it has now been established that a reliable definition of requirements is essential for project success, developers are still reluctant to invest a sufficient amount of time in requirements engineering. As a result, requirements modelling and analysis are often completely left out or not performed effectively. In turn, this results in developers not having proper insight into systems being developed and poor requirements leading to poor final products. This problem can be remedied by making requirements modelling and analysis methods more adequate for their intended tasks and more accessible for the industry developers.

The work described in this dissertation is concerned with improving the existing methods for modelling and analysis of requirements. Three main aspects characterise a requirements modelling and analysis method:

- A **notation** describes how a requirements model is to be constructed and interpreted, in other words its syntax and semantics. A requirements notation can be textual, symbolic, diagrammatic or a mixture of these three. It can also vary in its formal foundation.
- An **analysis technique** defines a way of checking a requirements model against certain properties. It either predefines these properties or provides a means for the user to define them. The modelling notation influences the way in which analysis can be performed. Informal requirements models can only be checked manually, while formal models can often undergo rigorous automated analysis.
- **Tool support** automating requirements modelling and analysis tasks is essential for effective and efficient use of a particular method [Ebe97].

We set out in our research to study and bridge the gap posed by the lack of an adequate method for requirements modelling and analysis. The following section describes the

approach that we took in order to achieve this and gives an overview of our solution to the problem.

1.1 Synopsis

The focus of our initial background research was on the existing methods for modelling and analysis of software system requirements. It was necessary to establish the desirable criteria for such methods before we could propose improvements in the area. An overview of the criteria that we identified is given below.

1. *Suitability:* The modelling notation must be suitable for representing information about software requirements and sufficiently expressive for capturing requirements at a detailed level.

2. *Understandability:* Requirements models must be comprehensible and sufficiently easy to navigate.

3. *Unambiguity:* Models in the notation must have a precise interpretation.

4. *Rigorous analysis:* The analysis technique must be rigorous producing useful and reliable results. The goals of the requirements analysis must be clearly defined and acknowledged by the user of the method.

5. *Tool support:* Construction and analysis of models must be automated by one or more software tools. Analysis of models must provide the user with results in a reasonable amount of time. Guidance with modelling and analysis of requirements must also be provided by the tools supporting the method.

Our evaluation of the available methods revealed that while some of them have many advantages, few can be highly rated on all of the above criteria. Consequently, we decided to bring together the best elements of several existing approaches to formulate an enhanced solution for requirements modelling and analysis. The diagram in Figure 1 depicts the solution that we propose in this dissertation.

Figure 1 shows how we use some existing techniques as building blocks to construct our enhanced method for modelling and analysis of software requirements. We base our notation on the approach to capturing and modelling requirements that is the most popular

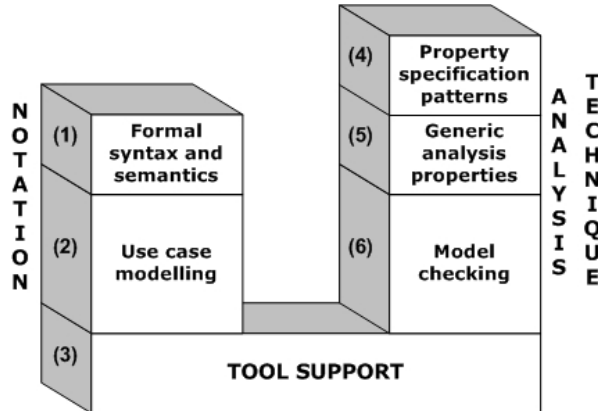


Figure 1: Enhanced Method for Requirements Modelling and Analysis

at the moment - *use case modelling* [BS03, BRJ99], shown in block (2) in the diagram. This technique was first presented by Ivar Jacobson [Jac92], but it is now considered to be a part of the *Unified Modelling Language (UML)* [BRJ99]. Requirements models in the use case notation are semi-formal and usually consist of diagrams supplemented by text. They are well-suited for representing functional requirements for software systems. Despite their popularity, use case models lack structure and exact semantics, which makes rigorous analysis of such models impossible. In our proposal, we amend traditional use case models with a *formal syntax and semantics* to make them suitable for automated analysis, as represented by block (2) in Figure 1.

Rigorous analysis of formalised use cases is enabled with *model checking* [CES86, QS81] in our solution, as illustrated in block (6) in Figure 1. Model checking is the process of algorithmically determining whether a behavioural model satisfies certain specification properties, which are usually expressed in some form of temporal logic [CES86]. Our amendment of the use case notation facilitates creation of high-level behavioural models that capture the desired functionality of a system, and these are then analysed with model checking.

In this research we utilised the NuSMV model checker [CCGR99, CCG+02] as the analysis engine for our requirements models. NuSMV is a state-of-the-art *symbolic* model checker, which is based on Binary Decision Diagrams. It verifies finite state-transition models expressed in a prescribed NuSMV input language. In order to make use of this tool, we defined a mapping from our formalised use case models to NuSMV programs. Specification properties for NuSMV analysis must be expressed in Computational Tree

Logic (CTL) [CE81] or Linear Temporal Logic (LTL) [Pri57, Pri67, Pnu77]. Our solution shelters the developer from the complexities of temporal logic in two ways. First, we define a number of *generic analysis properties* shown in block (5) in Figure 1. These can be used to analyse any requirements model, which allows the developer to check models without providing any extra input. Second, we make use of *property specification patterns* [DAC98, DAC99] that allow one to construct simple analysis properties in terms behavioural patterns and model elements. Specification patterns appear in block (4) in Figure 1 as part of the proposed analysis technique.

We called the proposed notation and analysis technique described above *Susan*, which stands for "*S*"ymbolic "*us*"e case "*an*"alysis. The construction, manipulation and analysis of the formalised use case or Susan models is automated by the software tool that we created called SusanX, which is represented by the *tool support* block (3) in Figure 1. SusanX translates Susan models to the NuSMV input language for analysis and also interprets results produced by the model checker in terms of the original models.

1.2 Original Contribution to the Research Field

While using several existing techniques and tools, our approach is novel from a number of different perspectives described below.

Our method of formalising and analysing use case models is original. Several attempts have been made to improve and formalise use case models. For example, Hausmann *et al* [HHT02] propose refining use cases with UML activity diagrams and expressing their pre- and post-conditions in terms of UML collaboration diagrams. This approach allows for static analysis of conflicts and dependencies in use case models. Back *et al* [BPPP99] formalise use cases with *contracts* defined in refinement calculus, which facilitates rigorous analysis of use case models for properties such as "achievability" and safety. The complexity of the mathematical notation underlying this approach and the absence of tool support automating the analysis makes this technique impractical. Our proposed solution enhances use case modelling by facilitating automated dynamic analysis of the models while keeping the complexities of the analysis hidden from the developer.

We applied model checking in a domain where it has not been used before. The use of model checking has proved to be very successful in verifying hardware designs, and recently its application to software models has notably increased [CAB+98, RDH03,

EP02]. Cheng *et al* [CCMS02, MC01] have developed a framework and a number of tools for translating UML class and state diagrams to formal specifications that can be simulated and analysed by model checkers. The NuSMV model checker has been used for verification of business process models, where state automata representing business models were translated to the NuSMV input language [KTK02]. We introduce a different application of model checking - analysis of high-level behavioural use case models, and we investigate the suitability of NuSMV for this purpose.

We proposed and investigated the usefulness of a generic analysis technique for requirements models. The advantages of rigorous analysis have always been offset by the effort and special skills required from the developers. A lot of work has gone into making these rigorous techniques more accessible. Our generic analysis option can be run by the developer with a "push of a button". The characteristics or properties that models are checked for during this type of analysis have not been applied in use case modelling before. In our work, we also examine another way of reducing the obstacles faced by developers when using formal analysis techniques - employing specification patterns in construction of analysis properties for use case models.

1.3 Objectives, Scope and Limitations

As explained before, the problem that initiated this research was the apparent inadequate state of requirements modelling and analysis methods available to the industry. As we learnt more about this problem and reviewed the work relevant to the field, we formulated the precise objectives for our research. These were to,

1. Enhance the standard use case modelling approach, so that unambiguous use case models can be constructed that are suitable for rigorous analysis by model checking.

2. Select a suitable model checker for the analysis of formalised use case models and define a mapping from the use case models to the input language of the model checker. In addition, determine how results produced by the model checker can be interpreted in terms of the original use case models.

3. Develop a means for the developer to control the analysis of the formalised use case models, while hiding the details pertaining to model checking. This includes providing an accessible way for specifying behavioural properties for the analysis.

4. Build a prototype software tool to construct, manipulate and analyse formalised use case models.
5. Carry out a case study whereby a real system is modelled and analysed according to the proposed method using the developed tool. Evaluate the results of the case study and draw conclusions on the usefulness of the proposed approach and the tool.
6. Make recommendations for future work in the field of requirements modelling and analysis.

The scope of this research work has been limited to the duration of a Master of Science programme at the University of Cape Town. More specifically, the following scope restrictions and assumptions have been made.

- Although some aspects of our work can be applied to systems in any general engineering discipline, in this dissertation we concentrate only on the development of software systems.
- Only the problems pertaining to modelling and analysis of requirements are considered in this work, not the problems of the entire requirements engineering field. We assume that solving these problems in isolation is reasonable.
- This research does not specifically address non-functional requirements of software systems.

1.4 Dissertation Outline

This dissertation first gives essential background on requirements engineering aspects relevant to our research and the fundamentals of model checking. Then it describes in detail our proposed requirements modelling and analysis method, Susan. The development of the SusanX software tool is explained with reference to the important design and implementation details. Finally we share our experience in the performed case study, draw conclusions and make recommendations for future work. The chapter outline for the dissertation is as follows.

Chapter 2 provides the reader with background on the field of requirements modelling and analysis and surveys the existing work in the area.

Chapter 3 first explains the standard use case modelling approach, including its strengths and weaknesses. The main illustrative example of a telemedicine system used throughout the dissertation is introduced. This is followed by an overview of the strategy that we employed to formulate our improved requirements modelling and analysis method. A comparison of our work with previous efforts in developing use case modelling is also provided.

Chapter 4 contains important background theory on model checking and an overview of the existing model checking tools. Our choice of the NuSMV model checker is justified, after which the essential details of the NuSMV input language are provided.

Chapter 5 describes the Susan notation in detail. The mapping from Susan to the NuSMV input language and the interpretation of verification results produced by the model checker are explained. This is followed by a description of the different analyses of Susan models that we propose.

Chapter 6 shows how all the theory previously described was implemented in the SusanX tool.

Chatper 7 presents the completed case study of a Cash Management System (CMS). The case study experience and the obtained results are evaluated in this chapter.

Chapter 8 concludes the dissertation and suggests directions for further work.

Chapter 2

Methods for Requirements Modelling and Analysis

In this chapter we first discuss the importance of modelling and analysis in the field of requirements engineering. Subsequently, we explain the possible notations, analysis techniques and tool support that can be employed by a requirements modelling and analysis method. Then we describe and draw a comparison of the existing methods and justify our choice of use case modelling as the basis for Susan.

2.1 Importance of Modelling and Analysis of Requirements

We begin by providing a few definitions that are essential to the discussions in the remainder of this dissertation.

Modelling: As in other domains, in software engineering, modelling is primarily done to create a representation of a system that allows one to reason about and analyse that system. A good model expresses complex aspects of a system in a manner that makes them more comprehensible and is less expensive to create than the system itself [Ebe97]. A requirements model for a software system is any abstract description of what is required from that system. These models often consist of a combination of diagrams, symbols and text.

Structural and behavioural models: A structural model emphasises static parts of a system, such as its architecture for example. A Unified Modelling Language (UML) class diagram is a typical example of a structural model. A system's functionality or behaviour over time and space is depicted in a behavioural model. Such models

generally show communication between the different parts of the system, as well as between the system and its environment. Behavioural models often contain some structural information about the system as well.

Analysis and verification: In the domain of requirements engineering, analysis is the process of studying and refining requirements [Ins90]. Verification is an analysis activity that confirms that a system is being built correctly. The goal of requirements verification is to check that requirements for a system are correct, complete, consistent and unambiguous [Ins90, Ebe97].

At present, the main trends in software engineering are concerned with replacing approaches based on documentation with model-driven development [mda02]. The main goal of the modelling trends is automation of as many of the software development activities as possible. The long-term vision is to be able to generate a complete system implementation from design or even requirements models without having to manually program the solution. Several tools such as IBM Rational Rose and Rational Software Architect already provide a certain degree of code generation from software design models.

Unlike design for example, requirements engineering activities have always been based primarily on documentation rather than models. One of the reasons for this is that when requirements are elicited from the client, they are expressed verbally. Hence, it is easier for the developers to write down their interpretation of these requirements than transform them into modelling concepts. On the other hand, software designs are naturally created using diagrams or other modelling constructs to explain how a system should be built. However, modelling as a technique for capturing information has a number of advantages that apply to design as well as requirements engineering.

Firstly, model representations are usually more concise than textual descriptions and can convey large amounts of information succinctly. Consequently, models are easier to analyse than lengthy documents. Secondly, models are more traversable or “traceable”, as it is more often referred to in software engineering. This once again makes models easier to understand and navigate than text. Thirdly, models have the power to express requirements more precisely than natural language, which is inherently ambiguous. A precise model can be analysed rigorously by formal techniques such as model checking and theorem proving, which are described further on page 17. Analysis of requirements definitions is essential to ensure that they capture requirements in a correct, consistent and complete manner. Several

methods have been proposed to improve requirements engineering with these benefits of modelling. However, few of these methods have succeeded in practice as we explain shortly.

2.2 Notations, Analysis Techniques and Tool Support

In Chapter 1, page 5 we distinguished between three aspects characterising a requirements modelling and analysis method: notation, analysis technique and tool support. Some additional detail on these is given below.

2.2.1 Notation

As mentioned already, in the context of requirements modelling a notation describes how a requirements model is to be constructed and interpreted. A model can consist of diagrams and textual elements such as characters, symbols, abbreviated expressions and annotations. Modelling notations can be classified into the following categories.

- **Informal.** Open-ended and flexible notations without precise interpretations fall into this category. Models in such notations usually consist of ad-hoc diagrams produced by developers to communicate their ideas to others or supplement documents. Informal models can often be understood by non-technical stakeholders. However these models cannot be used to convey information in an unambiguous manner, are not suited to rigorous analysis and are prone to contain inconsistencies and incompleteness [Ebe97].

- **Semi-formal.** Notations that define a set of valid elements to be used in a model and restrict the ways in which these can be put together are semi-formal, provided that overall models still do not have a precise interpretation. Semi-formal models are mid-way between informal and formal notations. They are more suitable for analysis than the informal models and more comprehensible than the formal models.

- **Formal.** Formal modelling notations have a well-defined syntax and semantics. Algebraic and logic notations are prime examples of notations in this category. The main advantage of formal models is that they provide an unambiguous way to express information and can be rigorously analysed for properties such as correctness, completeness and consistency. However, considerable effort and skill are usually required to build formal models, which is why formal notations are not often used in practice.

2.2.2 Analysis technique

Analysis of requirements models can be performed with a number of different goals, and it is essential that the developer understands these goals. Requirements model analysis can be carried out to,

1. *Provide developers with a better understanding of the system.* Examining individual model elements, how they relate to each other and work together can be very effective in improving the developers' understanding of the system and its requirements.

2. *Expose errors, inconsistencies and incompleteness in the model.* The first version of a requirements model is very unlikely to be error-free, consistent and complete. A number of analysis and adjustment cycles are necessary to get a model into an adequate state. In spite of this, proving absolute correctness, consistency and completeness of a requirements model is a conceivably impossible task.

3. *Show that certain constraints hold in the model.* Typically a system is required to provide functionality under certain constraints that are defined as non-functional requirements for the system. In this case, a model of the functional requirements can be analysed to check that it satisfies these additional constraints.

Currently, there are a number of analysis techniques that have application in different areas of software engineering including requirements engineering. The salient of these are briefly described next.

- **Manual inspection.** This category refers to examining models by-hand either in an ad-hoc manner or according to a structured process.

- **Static analysis.** All the techniques for checking the structure of models fall into this category. This type of analysis can be applied to models in semi-formal and formal notations. For behavioural models, static analysis is typically performed as a prelude to further analyses involving model execution.

- **Animation.** Automated walk-through of behavioural models is called animation, simulation or meta-execution. The manner in which animation is performed largely depends on the modelling notation used. Most of the time, the effect of each step through the system's behaviour is shown during the trace and the user instructs the

animation tool when and how the next step should be executed. Animation is very useful in providing developers with insight into behavioural models that are expressed in semi-formal or formal notations.

- **Model checking.** Formal behavioural models expressed as systems of state transitions can be analysed using model checking techniques [CES86, QS81]. Model checking determines whether a model satisfies a given property by exhaustively searching all of its possible behaviours. In addition to a formal model description, model checkers require the user to provide a specification of verification properties stated in temporal logic [BCG88]. At the end of the analysis, a model checker reports to the user whether the given property is satisfied by the model. In case of a negative result, most model checkers generate a counter-example trace showing how the property can be violated. This analysis technique is very effective in verifying models for desirable properties or constraints, but its major drawback is that the time taken by analysis grows exponentially with the size of the model.

- **Theorem proving.** Similar to model checking, theorem proving can be used to determine whether a formal behavioural model satisfies a certain property [BM84, Fit96]. However, for this analysis technique models need to be described by a series of axioms and hypotheses expressed in a form of logic, which is also used to write the verification properties. The analysis process shows whether it is possible to derive a given property from the model by using the principles of the underlying logic. This process requires guidance of an expert to be effective and efficient, which is why theorem proving is not commonly used in practice.

2.2.3 Tool support

In the vigorous software development industry of today a requirements modelling and analysis method must be supported by a software tool if it is to be used in practice. Many editors are currently available for different requirements modelling notations. Some of these additionally provide organisation and management features, such as grouping of models into packages, configuration management and version control. Tools for modelling notations that are suitable for analysis provide different degrees of analysis automation. On the one hand, a tool can automate the complex analysis computations but still require close guidance by the user. On the other hand, certain tools fully automate the entire analysis process

needing no input from the user other than the actual requirements model. Guided analysis requires more work from the user and possibly special skills to make effective use of the process. However while fully automated analysis is much less demanding from the user's point of view, it can be inflexible and limited when compared to the guided option. Tools that offer both analysis options can be very advantageous.

2.3 Existing Requirements Methods

We surveyed numerous methods that are currently available for modelling and analysis of software requirements. Many of these methods focus mainly on the modelling aspect, without defining or suggesting analysis techniques to be used on requirements models. We suggest two possible reasons for this: either modelling notations underlying the methods are not suitable for analysis purposes or the methods were not sufficiently developed to include analysis techniques. The methods surveyed varied greatly in the measure of guidance provided to the user in applying the method. Below is a list of the most significant of these methods and a brief description of each.

ALBERT: The Agent-oriented Language for Building and Eliciting Real-Time requirements was developed at the University of Namur, Belgium [DBDP94]. This method can be used to model in detail the intended behaviour of real-time systems. ALBERT prescribes a formal modelling notation based on temporal logic and in addition to that provides users with less formal diagrams to capture the high-level aspects of models. A software tool called GraphTalk (registered trademark of Rank Xerox) can be used as an editor for certain parts of ALBERT models.

CORE: The COntrolled Requirements Expression method was introduced by Mullery in 1979 [Mul79]. This method models requirements in terms of *viewpoints* that are used to represent all entities that interact with, affect or have an indirect interest in the system [KS98]. CORE models consist of tables and simple structural and data flow diagrams.

KAOS: Knowledge Acquisition in autOmated Specification is a goal-driven method proposed in 1991 by van Lamsweerde *et al* [vLDDD91, DvLF93]. It borrows several notions from the theories of Artificial Intelligence, the most important one of these is that of an *agent*. An agent is an entity that has characteristics like autonomy, social

ability, reactivity and proactivity. The notation underlying KAOS is predominantly textual, where certain aspects of models are formalised with first order temporal logic. The method is supported by an editor called GRAIL [DDMvL97].

RML: The Requirements Modelling Language was first presented in 1982 [GMB82]. This language prescribes a textual or code-like notation for modelling system behaviour, which has a formal semantics and can be mapped to first order predicate calculus. Modelling with RML is supported with a software tool called ACME [GFST91]. After some experimentation RML was criticised for its rigidity and an improved version called Telos was developed to make it more flexible [MBJ90].

Sequence diagrams: UML offers sequence diagrams to capture required system behaviour by showing interactions between the system and its environment. Interactions are expressed as messages passed between the system and entities in its environment with emphasis on the time ordering of events. UML advocates using sequence diagrams in combination with use case models. Most available UML modelling tools provide editor support and sometimes animation facilities for sequence diagrams. Rational Rose and Telelogic Tau UML Suite are just two examples.

Tropos: Tropos is an agent-oriented methodology that is a recent development proposed in 2001 by Bresciani *et al* [BPG+01, GMP02]. This methodology encompasses the entire software development process, but places a great emphasis on modelling and analysis of requirements which is why we include it in our discussion. Tropos requirements models are based on agents and notions such as beliefs, goals, actions and plans. Tropos prescribes semi-formal diagrammatic notations for requirements models during early and late requirements analysis stages. A number of analysis techniques based on principles from Artificial Intelligence are defined for Tropos requirements models.

Use case modelling: The main idea behind the use case approach was first proposed by Jacobson [Jac92], but now use case modelling is considered to be a part of UML. Use case models provide a way to demonstrate what is required from a system without specifying any of its internal behaviour. The use case notation consists of simple diagrams that are often supplemented by textual descriptions. Typically all tools that provide UML modelling facilities support use case modelling. In the recent years this method has become increasingly popular in the industry.

Z: The Z method was developed by the Programming Research Group at the Oxford University Computing Laboratory in the late 1970s [Spi92]. It provides a way for precisely expressing behaviour of software systems in a notation based on set theory and first order logic. The main elements of Z models are *schemas* that are intended to be supplemented by informal explanations to make them more comprehensible. Z schemas can be used to describe state variables, operations within a system and invariant conditions that must not be violated. These descriptions get lengthy and difficult to understand for a system of a moderate size. Numerous tools have been developed for Z modelling and analysis, FuZZ is one example. A similar approach is the Vienna Development Method (VDM) [Jon90, JS90] that was developed prior to Z at the IBM Laboratory in Vienna.

Table 1 classifies each one of the methods in terms of the modelling notation that they use, the type of analysis they allow and their tool support. We use the following categories for these three characteristic aspects of a requirements modelling and analysis method:

- **Notation:** *informal*, *semi-formal*, *formal*
- **Analysis technique:** *manual inspection*, *static analysis*, *animation*, *model checking*, *theorem proving*, *other*
- **Tool support:** *editor*, *management*, *guided analysis*, *fully automated analysis*

Method	**Notation**	**Analysis technique**	**Tool support**
ALBERT	*formal*	-	*editor*
CORE	*semi-formal*	*static analysis*	-
KAOS	*formal*	-	*editor*
RML	*formal*	*static analysis*	*editor*, *guided analysis*
Sequence diagrams	*semi-formal*	*animation*	*editor*, *management*, *guided analysis*
Tropos	*semi-formal*	*other*	-
Use case modelling	*semi-formal*	*manual inspection*	*editor*, *management*
Z	*formal*	*static analysis*, *theorem proving*	*editor*, *guided analysis*

Table 1: Overview of Existing Methods

The methods that we looked at are so diverse that it is difficult to draw a direct comparison between them. Nevertheless we assessed them on the basis of the desirable criteria

for a requirements modelling and analysis method defined in Chapter 1 on page 7, in an attempt to evaluate their relative merit. These criteria are suitability (SUIT), understandability (UNDERSTAND), unambiguity (UNAMBIG), rigorous analysis (ANALYSIS) and tool support (TOOL). This assessment was done using documented critiques of the methods as well as our own judgement. The results are shown in Table 2. In the table we use ■ to indicate that the method satisfies a criterion and □ that it does not.

Method	**SUIT**	**UNDERSTAND**	**UNAMBIG**	**ANALYSIS**	**TOOL**
ALBERT	□	□	■	□	■
CORE	■	■	□	□	□
KAOS	□	□	■	□	■
RML	□	□	■	□	■
Sequence diagrams	□	■	□	□	■
Tropos	□	■	□	□	□
Use case modelling	■	■	□	□	■
Z	□	□	■	■	■

Table 2: Assessment of Existing Methods

The most important of the criteria used in our comparison above is suitability. Many powerful and well-established methods exist for capturing system behaviour at a detailed level, such as Statecharts [Har87] for example. However, they cannot be used effectively to represent software requirements that state *what* a system must do rather than *how* it must to it. As can be seen from Table 2, only two out of all the methods are suitable for representing information about software requirements. The first of the two methods is CORE, which has not achieved acceptance by developers in the industry. The other method is use case modelling, which unlike CORE is being widely used in practice for modelling software requirements.

The original objective of our work was to bridge the gap posed by the lack of an adequate method for modelling and analysis of software requirements. In order to do this, we could have formulated a completely novel approach that was superior to the existing methods discussed in this section. Instead of re-inventing the wheel however, we decided to improve the most successful of the surveyed methods - use case modelling. As shown in Table 2, the use case approach lacks an unambiguous notation and a rigorous analysis technique. We extend the standard use case modelling with these in our enhanced method called Susan.

This concludes our discussion of the methods for requirements modelling and analysis. In this chapter, we explained the importance of the modelling and analysis activities during requirements engineering. Furthermore, we introduced the predominant techniques for modelling and analysing software requirements and compared them on the basis of several characteristics and criteria.

Chapter 3

Use Case Modelling

In this chapter we first provide background on use case modelling and explain the benefits and drawbacks of this technique. Next we introduce an example of a telemedicine system, to be used throughout the dissertation, and show how its requirements can be modelled with use cases. Furthermore, we explain how the Susan method improves the standard use case modelling approach and compare Susan to related work.

3.1 Fundamentals of Use Case Modelling

Use case modelling is one of the only approaches to defining software requirements that allows one to represent the intended behaviour of a system without specifying how it should be implemented. Booch *et al* [BRJ99] state that use case models can be used *"to visualize, specify, construct, and document the intended behavior of your system during requirements capture and analysis"*. In fact, the use case approach can be used to model behaviour of sub-systems, components and even individual classes, in addition to entire software systems.

Requirements captured in use case models provide an outside view of the interaction between a system and its environment. The main elements of these models are *actors* and *use cases*. Actors are used to represent entities that interact with the system, while use cases define services that the system must provide. Before embarking on a detailed discussion of this method, it is important to define and distinguish between the following three concepts.

Use case: A use case describes a service that the system is required to provide to an actor. This service is expressed as a collection of possible scenarios of interaction between the system and its environment needed to yield the value of the service to the actor.

A use case must have a name that describes the service underlying the use case, and this name is used to refer to it.

Use case diagram: Actors and use cases have graphical representations, which are used to capture the relationships between these modelling elements in a use case diagram. This diagram provides an overview of the people and other entities that interact with the system and their requirements.

Use case model: A complete use case model consists of one or more use case diagrams and additional descriptions of the scenarios associated with each use case in the model. A use case diagram alone does not contain sufficient detail to serve as a requirements model on its own.

Actors in use case modelling can be human or other automated systems that interact with the modelled system, but are not part of the system themselves. In fact actors do not just represent entities in a system's environment, but rather correspond to a coherent set of roles that these entities play when interacting with the system. For example, a particular person can be embodied as several actors in a use case model if he plays different roles when using the system.

A use case description usually consists of a number of *flows*, which are essentially sequences of events depicting scenarios of the system interacting with its environment. Typically, a use case has one *main* flow and a number of *alternative* or *exceptional* flows. Flows for a use case can be expressed in natural language, captured in sequence diagrams or state machines, or in another appropriate manner. Additionally other details such as priority, trigger event, pre-conditions and post-conditions are often defined for use cases.

Diagrammatically, use cases are shown as ellipses or bubbles and actors as stick figures. Figure 2 shows how the elements of use case models can be related to each other and how these relationships are reflected in diagrams. These relationships are explained below.

(a) **Association:** Actor receives value from the system through Use case. Associations also represent interaction between actors and the system. This is the only valid relationship between an actor and a use case in use case modelling.

(b) **Actor generalisation:** Actor 1 inherits all the use case associations from Actor 2. This relationship can be used to distinguish between general and more specialised roles represented by actors.

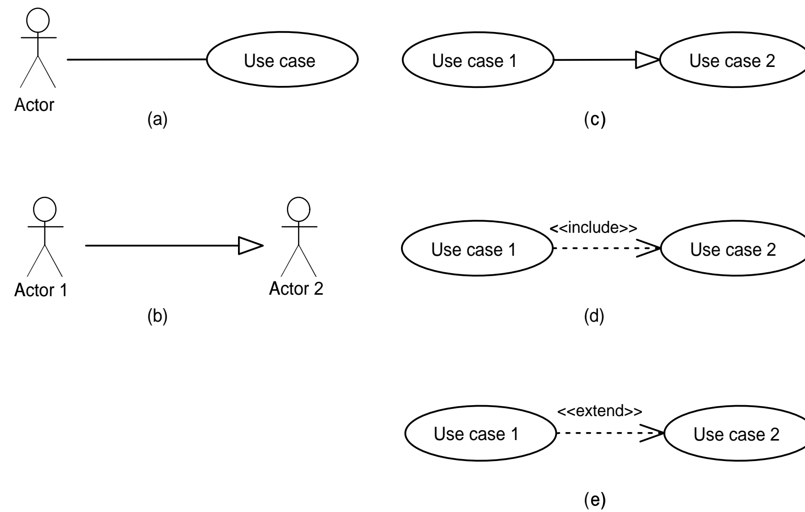


Figure 2: Relating Elements in Use Case Models

(c) **Use case generalisation:** Use case 1 inherits behaviour from Use case 2, in a similar way to class generalisation.

(d) **Use case include:** Flows of Use case 2 are contained within the flows of Use case 1. The include relationship can be used to show that the same behaviour is shared by a number of use cases. By organising use cases with this relationship, one can avoid describing the same behaviour several times.

(e) **Use case extend:** Use case 1 can be optionally extended by the behaviour of Use case 2. The flow of Use case 1 "can but does not have to" contain the behaviour of Use case 2. Contrast this with include relationship where Use case 1 "must" include the behaviour of Use case 2.

The most important relationship that underlies use case modelling is the association between an actor and a use case. Surprisingly, this relationship has the least comprehensible definition and is most misunderstood by users of the method. Booch *et al* provide this ambiguous definition: *"An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages"*. The confusion arises from the open-ended nature of the use case modelling approach. Originally, use cases represented the different ways in which actors could use the system - hence the name "use case". Subsequently, people began to associate one use case

with multiple actors to show all entities that the system interacted with during execution of the use case. Today, development teams usually adopt and agree on a definition of the actor and use case association that they find the most suitable to their particular context.

Several sets of guidelines for use case modelling have been proposed over the years [Coc00, ASJ01], all of which are informal and often contradictory. The Rational Unified Process (RUP) [Kru01] that prescribes effective usage of the Unified Modelling Language (UML) during different development phases provides, limited support for use case modelling as such. Furthermore, no specific technique for analysis of use case models accompanies the method. This can be primarily attributed to the informal nature of use case modelling.

In the next section we provide an evaluation of the use case modelling approach by considering its strengths and weaknesses. This is followed by an elaborate example of a use case model in Section 3.3.

3.2 Pros and Cons of Use Case Modelling

Booch *et al* advocate three main purposes for use case modelling. Firstly, use case models provide a means of communication between all of the system stakeholders because they can be understood even by non-technical people. Secondly, use case modelling and analysis allow developers to get better insight into a system's requirements. Lastly, use cases can be used to validate a system all the way during the development process. In this dissertation, we primarily look at use case modelling as a method for modelling and analysis of requirements. With this particular purpose in mind, the main strengths of the use case modelling method are as follows.

1. *The approach is relatively simple and flexible.* The fundamentals of use case modelling are straightforward, which makes it easy to learn the basics of the approach. The flexibility arises from the fact that there is no unique way prescribed for constructing or using use case models. Additionally, the level of detail that use case models contain is flexible.

2. *Use case models show the "what" and the "who" without the "how".* The goal of requirements specification is to identify "who" the stakeholders for the system are and "what" they require from the system. Use case models are well-suited to capture this type of information without showing "how" the system needs to be built, which is a design concern.

3. *Stakeholders can understand use case models.* Use case models are high-level and their graphical nature makes them intuitive and easy to grasp for clients and other stakeholders. Hence, developers can use these models as a basis for discussion with stakeholders, as well as get stakeholders to review models that capture functional software requirements.

4. *Use case modelling is well-integrated into the development process.* It naturally integrates into the software engineering process if UML is used during the other development phases.

Above and beyond these strengths of use case modelling, the approach suffers from several weaknesses explained next.

1. *Effective use case modelling is challenging.* Although it is easy to learn the basics of use case modelling, effective use of this approach is not a simple task. The lack of a standard and consistent set of guidelines and the extensive flexibility of the approach make construction and manipulation of use case models difficult.

2. *Textual use case descriptions lack structure.* There is no prescribed structure for textual use case descriptions. In other words, there is no stipulated set of attributes that must be specified for a use case, neither are there formats set down for commonly used use case descriptions. Consequently, one can never be assured of the level of detail, type of content or presentation of use case descriptions.

3. *Use case models are ambiguous.* Supplementary use case descriptions are usually given in natural language, which is inherently imprecise, making use case models ambiguous. Additionally, certain graphical elements of use case models are poorly defined. The ambiguous nature of the association between an actor and a use case makes it difficult to identify use cases, and also obscures interpretation of already created use case models.

4. *It is impossible to analyse use case models for correctness, completeness or consistency.* Use case models are not based on a formal syntax or semantics. Textual supplements to diagrams are usually in natural language, and as a result cannot be analysed in any formal way. This means that the only way a use case model can be analysed for qualities such as correctness, consistency and completeness is by being

checked by a human reader. Naturally, this becomes more difficult and unreliable as the amount of information in the model increases.

3.3 Illustration of Use Case Modelling

In the following chapters that describe our proposed requirements modelling and analysis method, we use a telemedicine system called MuTI as an example. We begin by presenting a use case model for this system to demonstrate the standard use case modelling method. MuTI stands for Multimodal Telemedicine Intercommunicator and is a system that was developed during a research project at the Computer Science department of the University of Cape Town by Chetty *et al* for use in rural areas of South Africa [CTB04]. The main purpose of the system is to facilitate communication and exchange of data between doctors and nurses situated in remote locations in rural areas. For example, MuTI allows a nurse at a clinic to consult a doctor at a remote hospital before sending a patient there. Nurses and doctors can exchange images and other data during a consultation, which allows for some patients to be diagnosed without the physical presence of a doctor. In this way, the limited resources of the rural community are used more efficiently and health-care becomes accessible to more people.

MuTI is a custom telemedicine system that was developed for and tested in a specific area in the Eastern Cape province of South Africa. This area is not covered by a fixed network and the only available network is wireless, which is not reliable at all times. These special conditions are taken into account in the MuTI system, as it provides two modes of communication: synchronous and asynchronous. In the synchronous mode, users can interact in real-time such as for example with Voice over the Internet Protocol (IP). The asynchronous mode allows a user to prepare data that needs to be sent across to another user, after which that data is stored locally and forwarded to its destination when the network connection is available. With this mode of communication, users can exchange medical data in a manner similar to electronic mail. The medical data that nurses and doctors deal with consist of patient profiles and medical records, which can include text, images and audio.

Each MuTI user is given a unique identifier name, a User ID. When one user wishes to communicate with another, the User ID of the target user needs to be provided as well as the location where he is running a MuTI application. Each location is identified by a MuTI

address.

The main features of MuTI are outlined below.

- *Secure login.* All users need to log in and be validated by the MuTI system before they can have access to any of its services.

- *Address book.* Each user can maintain an address book that allows quick access to addresses and other details of the user's contacts.

- *Dynamic online status for contacts.* At all times, the user can see which contacts in the address book are online and which are offline. If response time is important, seeing the online status of the contacts helps the user to decide which contact to consult.

- *Voice calls.* The users can call each other using the Voice over IP technology, which compensates for the absence of traditional telephones in the area. In order to place a call, the caller must provide the User ID of the callee and the MuTI address of the callee's location. The caller can either type in these details or select one of the contacts from the address book.

- *Storage of patient data.* A MuTI user has a number of patients, whose personal details are stored in the system. Each medical record created by the user must be attached to a certain patient. A medical record for a patient can consist of text files, images and audio recordings.

- *Exchange of medical records between users.* Once a medical record is created, it can be sent to remote MuTI user. Received medical records get stored under the profile of the corresponding patient. If no such patient exists on the receiver side, a new patient profile is created.

- *Arrangement of doctor appointments.* A nurse may refer a patient to a doctor, in which case she can request an appointment with that doctor through MuTI. Once the doctor accepts or rejects the appointment, the nurse is informed.

The use case diagram in Figure 3 captures the requirements for the MuTI system. As can be seen, it depicts actors, use cases and their relationships. All the MuTI features described before are refined and represented by use cases in the diagram. For example, the *Voice calls* feature is supported with the *Place call* and *End call* use cases. Sometimes,

use cases are drawn inside a rectangle representing the system boundary to emphasise that actors are entities outside the system. However since the behaviour of only one system can be modelled in a single use case diagram, showing the boundary rectangle is redundant.

There are two main users for this system - *Nurse* and *Doctor*. Both need to have access to most of the services provided by the system, but have some distinct requirements for the system as well. For example, only a nurse can request an appointment with a doctor and only that doctor can accept or reject it. In order to represent the shared requirements, we created an additional actor called *Any user* and linked both *Nurse* and *Doctor* actors to it with the generalisation relationship. In this way, all the use case associations of the *Any user* actor are inherited by the *Nurse* and *Doctor* actors as necessary.

The MuTI use case diagram shows an example of each of the three valid use case relationships. The *Validate user* use case is included in the *Log in* use case to show that the validation procedure takes place every time a user logs into MuTI. On the other hand, the extend relationships used on the *Log in* use case show that the user is required to log in before he can make use of the services represented by the extending use cases. Finally, the use case generalisation is used to group related use cases. For instance, there are three ways in which a user can *Manage contacts* represented by the three use cases specialising this use case.

As an example of a supplementary use case description, consider the details of the *Place call* use case.

Main flow of events:

1. Caller provides User ID of callee and MuTI address of callee's location.
2. System checks whether callee is currently online at specified location.
3. Callee is currently reachable at specified location.
4. System sends call request to callee's location.
5. Callee is not busy and accepts call.

Pre-conditions:

Caller is not currently on another call.

Post-conditions:

Call established between caller and callee.

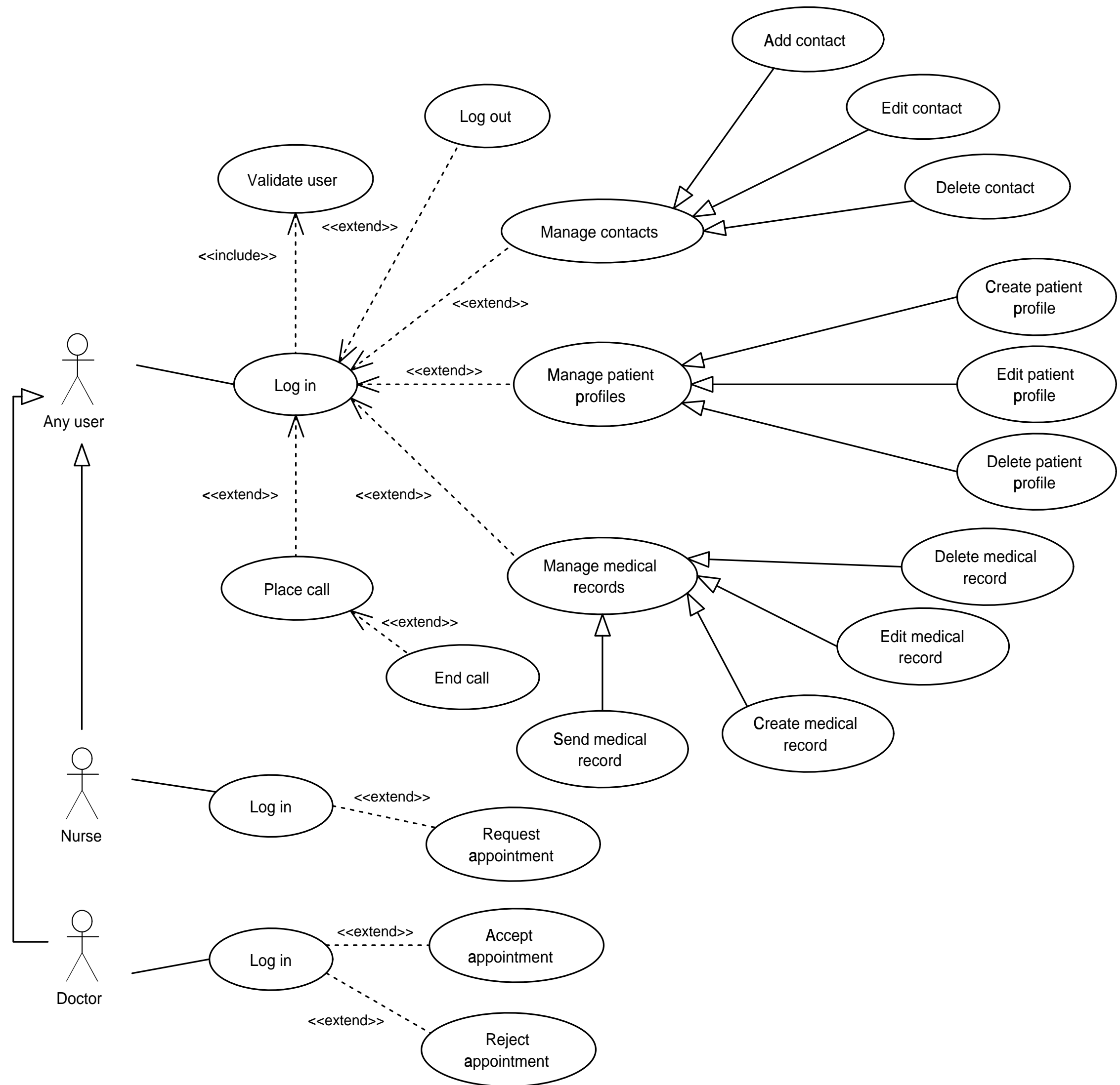


Figure 3: Use Case Diagram for MuTI Requirements

Alternative flow of events:

1. Caller provides User ID of callee and MuTI address of callee's location.
2. System checks whether callee is currently online at specified location.
3. Callee is currently reachable at specified location.
4. System sends call request to callee's location.
5. Callee is busy.
6. System asks caller to leave voice mail.
7. Caller records voice mail for callee.
8. System sends voice mail to callee's location.
9. System receives confirmation that voice mail was delivered.
9. System notifies caller that voice mail was delivered successfully.

Pre-conditions:

Caller is not currently on another call.

Post-conditions:

Voice mail from caller waiting for callee.

In this case, the use case description consists of the main flow, alternative flow and pre- and post-conditions associated with each of the flows. Here the flows are expressed in natural language as numbered sequences of steps, but could have also been written in a more informal manner in prose. As already mentioned, sequence or state transition diagrams could have also been used to capture these flows.

3.4 Our Approach to Improving Use Case Modelling

Our objective in improving the standard use case modelling approach was to alleviate the weaknesses described in Section 3.2 and at the same time retain as many of the benefits of the approach as possible. Essentially, we wanted to formalise use case models and define an appropriate technique for analysing the formalised models. In this section we explain how we proceeded in developing the enhanced use case modelling and analysis method, Susan.

We first inspect the important behavioural aspects of each use case in a use case model. Let us consider the *Place call* use case described in the previous section as an example. The diagram in Figure 4 shows all the pieces of information that we have for this use case, disregarding its relationship with other use cases in the model.

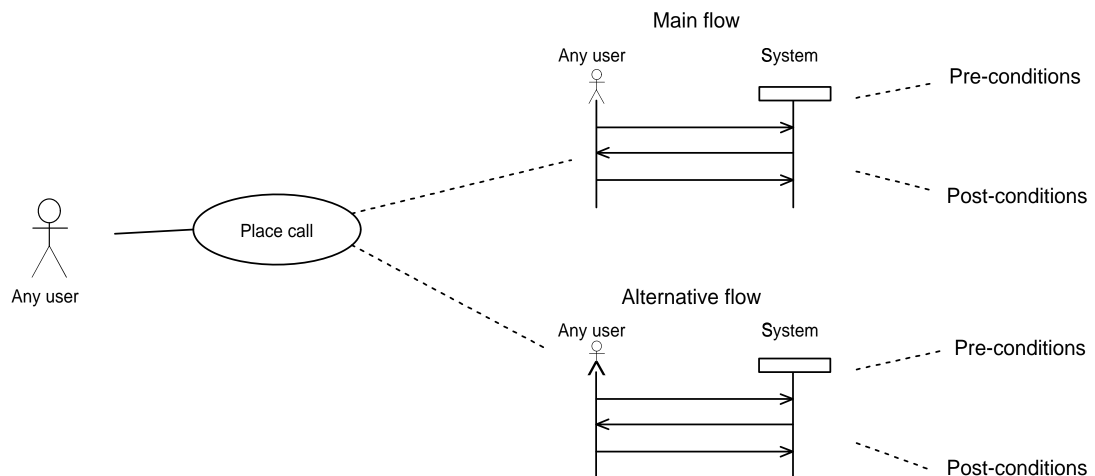


Figure 4: Details of Place Call Use Case

There are two flows defined for the *Place call* use case, each of which describes a scenario of interaction between the system and a MuTI user represented by the *Any user* actor. Additionally, each of the two flows has pre- and post-conditions associated with it. On closer inspection of each of the flows, we notice that the flow steps contain a mixture of information about *what* the system and user do and *how* they do it. Requirements modelling should only concern the *what* aspects, while the *how* details should be taken care of in design activities. Given a use case flow, we can extract the pure requirements information from it by determining *what* overall effect the flow has on the system, under *what* circumstances this effect takes place and *what* information is required by the system during the flow. Then the flows for the *Place call* use case can be interpreted as follows.

Main flow of events:
What information is required? Input: Callee User ID, MuTI address for callee's location.
Under what circumstances? If: Caller is not on another call, callee is reachable at the specified location, callee is not busy and accepts call.
What effect? Then: Call established between caller and callee.

Alternative flow of events:
What information is required? Input: Callee User ID, MuTI address for callee's location, voice mail.

Under what circumstances? If: Caller is not on another call, callee is reachable at the specified location, callee is busy and caller decides to leave voice mail.
What effect? Then: Voice mail from caller waiting for callee.

The use case details captured above contain the essence of the *Place call* use case. Altering the commonly used concept of pre- and post-condition slightly, we can include the entire *If* contents into the flow's pre-conditions and *Then* contents into the post-conditions. Here we describe these details informally, but if they are formalised then rigorous analysis can be performed of the required behaviour of the modelled system. Our proposed Susan method is geared towards capturing precisely this type of information about use cases in an unambiguous way.

We wanted our improved method to meet all the desirable criteria for a requirements modelling and analysis method that we identified in Chapter 1, page 7. The formulation of the Susan method was based on the lessons we learned while evaluating the various efforts in the requirements modelling and analysis field. For example, in several cases we observed how intricacies of a formal notation rendered methods uncomprehensible and unusable for typical developers. From this we inferred that the trade-off between formality and usability of a method cannot be underestimated.

In our view the most effective way to deal with the formality versus usability issue was by providing the user with tools that simplify the tasks involved in building and analysing formal models as much as possible. Hence the driving force behind the development of our enhanced use case modelling method was automation. During every step that we took in formalising use case models and devising analyses for them, consideration was given to how this step would be supported by the SusanX tool accompanying our method.

Before formalising the use case modelling notation, it was important to identify the type of analysis technique we wanted to apply to the resultant models. The type of analysis that can be applied to a particular model is principally influenced by two factors: the nature of the model, in other words whether it is structural or behavioural, and the formal foundation of the underlying notation. Structural models can essentially only be analysed by static means, while some behavioural models are appropriate for more advanced analysis techniques such as model checking and theorem proving. Most rigorous types of analysis require models to have formal syntax and semantics.

Use case models fall into the category of behavioural models, therefore there is a range

of analyses that could potentially be performed on them. We decided to investigate how model checking could be used for the purpose of analysing use case models. We chose model checking because it is a very powerful automated analysis technique that has been used successfully in a number of projects during various development activities [CAB+98, EP02, CCMS02, MC01]. The only technique that offers the capability of computing the same results is theorem proving, but it requires very specialised skills from expert users and for this reason is not as widely used.

3.5 Existing Approaches to Formalising Use Case Modelling

Several attempts have been made to address the drawbacks of use case modelling and formalise this method.

Hausmann *et al* [HHT02] propose an approach to modelling and analysis of software requirements based on formalising relationships between several UML diagrams with graph transformation theory. This approach suggests that there are two main types of requirements models: static and dynamic. Static aspects of a system are modelled using class diagrams, while use case diagrams capture its dynamic requirements. Inconsistencies are often introduced when static and dynamic models are integrated, as currently there is no adequate mechanism to check an integrated model for consistency. Hausmann *et al* tackle this particular problem by defining explicit relationships between static and dynamic requirements models and proposing a means of analysing them for consistency.

Behaviour of individual use cases is described by activity diagrams in this approach. Activity diagrams give an overview of the sequential or branching flow of a set of operations within a system. For each operation appearing in an activity diagram, pre- and post-conditions are defined as collaborations. In UML, a collaboration refers to a set of classes or other elements that work together to achieve some common objective. Hausmann *et al* draw collaborations as object diagrams showing only the elements relevant to the particular operation. A pre-condition collaboration shows a snapshot of the system before the operation is executed, and a post-condition collaboration shows how the objects and their relationships change after the operation execution. These collaborations serve as a link between static and dynamic requirements models.

Graph transformation theory is used to formalise collaborations in models and this facilitates rigorous consistency analysis. The aim of the analysis is to uncover potential

consistency problems and not to prove absolute model consistency. It is performed statically on the basis of *critical pair analysis*, and implemented in a tool called AGG. AGG is a tool for graph manipulations, and hence cannot be applied directly to UML models. For practical use of this approach, an interface between a UML tool and AGG needs to be defined and implemented. Such an interface would allow one to export UML models to AGG for analysis and then view analysis results in terms of the original UML models.

The method proposed by Hausmann *et al* is appealing, as it has the potential of allowing developers to build models using familiar visual techniques and at the same time benefit from formal analysis of these models. However as presented in [HHT02], the work seems to be incomplete especially since it is not substantiated with any application of the method in practice. Additionally, certain elements of use case models such as generalisation, include and extend relationships between use cases are not yet addressed by the method.

Back *et al* [BPPP99] formalise use case models with a precise mathematical notation called refinement calculus [BW98], which is an extension of Dijkstra's weakest precondition calculus. As in the method advocated by Hausmann *et al*, an effort is made to bring together modelling of classes and dynamic requirements for a system. Additionally, rigorous analysis for achievability of actor goals is proposed in this method.

According to this approach, classes with attributes and methods are defined in a formal textual notation. The collection of all class attributes describes the state of the system that can be changed by execution of use cases. Use cases are expressed as *contract* statements that essentially state how their execution affects the system state. Once again relationships between use cases are not taken into account by this method. In the prescribed notation, class and use case descriptions resemble computer programs.

Achievability analysis can be performed by first defining a goal formally and then performing weakest pre-condition computations to determine whether the goal can be achieved in the given model.

This approach certainly extends use case modelling with a formal notation and an analysis technique, however whether this is an enhancement to use case modelling is very questionable. The work is currently not supported by any tool, which makes it difficult to judge the potential usability of the method. However, the nature of the underlying notation and analysis technique do not lend themselves to much automation, so our conclusion is that although interesting, this approach is impractical.

The Susan method that we propose is based on concepts similar to those used in the

techniques developed by Hausmann *et al* and Back *et al*, such as pre- and post-conditions. As proposed in this dissertation, Susan equally does not incorporate the relationships between use cases and actor generalisations. However, Susan surpasses these techniques in improving use case modelling for two reasons. First, it maintains a relatively simple modelling notation while formalising use cases. Second, it is supported by a rigorous analysis technique and a tool that automates model analysis.

In their Masters dissertation [AB95], Andersson and Bergstrand formalise use case models with extended Message Sequence Charts (MSC) [ITU96]. Their work focuses on developing a graphical yet formal notation for describing use case flows. There are no suggestions for analysing the proposed extended models. As for tool support, only several suggestions are given in the future work section of the dissertation.

In general, numerous efforts have been made to formalise different aspects of UML. The UML Version 2.0 [uml03a, uml03b] has been a work in progress by the Object Management Group for the past few years. The aim of UML 2.0 is to provide a more complete and formal specification of the language with a special emphasis on its semantics. With respect to use case modelling however, the new version of UML provides only a few insignificant changes.

The Object Constraint Language (OCL) [ocl03] can be used to annotate certain UML diagrams with formal descriptions of constraints. OCL defines a relatively simple syntax and can be used to express invariants, queries, as well as pre- and post-conditions for UML modelling elements. The language is primarily based on the object-oriented concepts such as classes, associations and role names. Use case models deal with entities on a higher conceptual level, and hence applying OCL to use cases would not be practical. However, several constructs in the devised Susan notation resemble OCL.

With this we complete the background discussion of use case modelling. In this chapter, we presented the fundamentals of this the use case technique, illustrated its use with an example and discussed its strengths and weaknesses. Furthemore, an introduction to our proposed Susan method was given and related existing work on enhancing the use case approach was described. The following chapter concentrates on model checking and the NuSMV tool.

Chapter 4

Model Checking and NuSMV

In this chapter we explain model checking, discuss the strengths and weaknesses of this verification technique and compare existing model checkers. We support our choice of the NuSMV model checker for this project and provide the essentials of the NuSMV input language.

4.1 Fundamentals of Model Checking

Model checking is a technique for automatically verifying finite state concurrent systems, developed independently by Clarke and Emerson [CES86] and by Queille and Sifakis [QS81] in the early 1980s. In this context, verification refers to showing that a finite state system model satisfies a behavioural property expressed in a formal logic notation. This is performed by an exhaustive search of all the possible behaviours of the modelled system, implemented by tools called *model checkers*. When a model checker determines that a property is not satisfied by the model provided, it generates a counter-example trace of system behaviour that violates the property.

Systems are viewed as *Kripke structures* [Kri63] in model checking. A Kripke structure is essentially a nondeterministic finite state machine, where the states are labelled with *propositions* that hold in that state. A proposition is simply a statement that can either be true or false. A more formal definition is given below.

Let AP be a set of atomic propositions. Then a Kripke structure is represented by a tuple $M = < S, I, R, L >$, where:

- S is a finite set of states.

- $I \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a binary relation that gives all possible transitions from one state to another state. This relation is *total*, which means that $\forall x \in S\ \exists y \in S\ [(x, y) \in R]$ or each state must have a transition defined to itself or another state in the model.
- $L : S \rightarrow \mathcal{P}(AP)$ is a function that labels each state with atomic propositions that hold in that state.

During model checking, all the possible sequences of state transitions in a Kripke structure are considered. Figure 5 shows how a Kripke structure is "unwound" into a *computation tree* showing all the possible behaviours of the system.

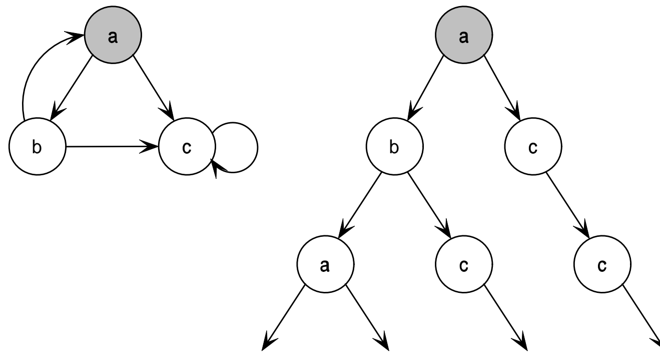


Figure 5: Unwinding a Kripke Structure

A simple Kripke structure with three states is represented by a state transition graph on the left in the diagram in Figure 5. There is one initial state, which is shaded and labelled *a*. The computation tree on the right is obtained by beginning with the initial state and repeatedly adding on states reachable through a valid transition. Since the transition relation must be total in a Kripke structure, all the branches in a computation tree are always infinite. A *path* is an infinite sequence of states such that every pair of adjacent states in the sequence are joined by a valid transition. It is important to note that although branches and paths in a computation tree are infinite, the set of all reachable states or the *state space* is always finite for Kripke structures.

The notation used for expressing behavioural properties for model checking is *propositional temporal logic* [BCG88]. This type of logic allows one to reason about system behaviour over time. Time is viewed as a sequence of system states, which are described by

combinations of propositions. The logic that is most commonly used in the field of model checking is Computational Tree Logic (CTL) [CE81]. In CTL the properties are composed of the constructors explained below. Suppose that p and q are atomic propositions.

- Two *path quantifiers* that describe for which paths a property must hold:
  - Ap *(forall)* : p holds for all computation paths.
  - Ep *(exists)* : there is least one computation path where p holds.
- Four main *temporal operators* that describe where within a particular path the property must hold:
  - Xp *(next)* : property p holds in the second state of the path.
  - Fp *(future)* : p holds on some state within the path.
  - Gp *(globally)* : p holds on all the states of the path.
  - $p\ U\ q$ *(until)* : q holds on some state within the path and p holds on all the preceding states.

In a CTL property, a temporal operator must always be preceded by a path quantifier. Figure 6 gives a visual representation of a number of simple but commonly used CTL properties. The shaded circles represent states where a proposition p holds.

In the diagram in Figure 6, *(a)* represents the property $AG(p)$ meaning that p holds on every state of every path. In this case, the property p is called an *invariant*. The computational tree shown in *(b)* depicts the property $AF(p)$, which means that p will inevitably hold on every path. $EG(p)$ implies that proposition p holds on every state of at least one path, as shown in *(c)*. Lastly, *(d)* represents the property $EF(p)$ meaning that p holds on some state of at least one path.

CTL is a *branching time* logic, which means that several possible futures are considered at any point in time. Alternatively to CTL, model checking can also be done using *Linear Temporal Logic (LTL)* [Pri57, Pri67, Pnu77]. Only one possible future exists in LTL, thus it reasons about paths rather than computation trees. As a result, the "possibility" of some event cannot be expressed in a LTL property. Furthermore, several other variations of these logics exist and are used by different model checkers. Extended Temporal Logic (ETL) [VW94], Linear μ-calculus [HH87] and Quantified Propositional Temporal Logic (QPTL) [SVW87] are just a few examples of alternative temporal logics for model checking.

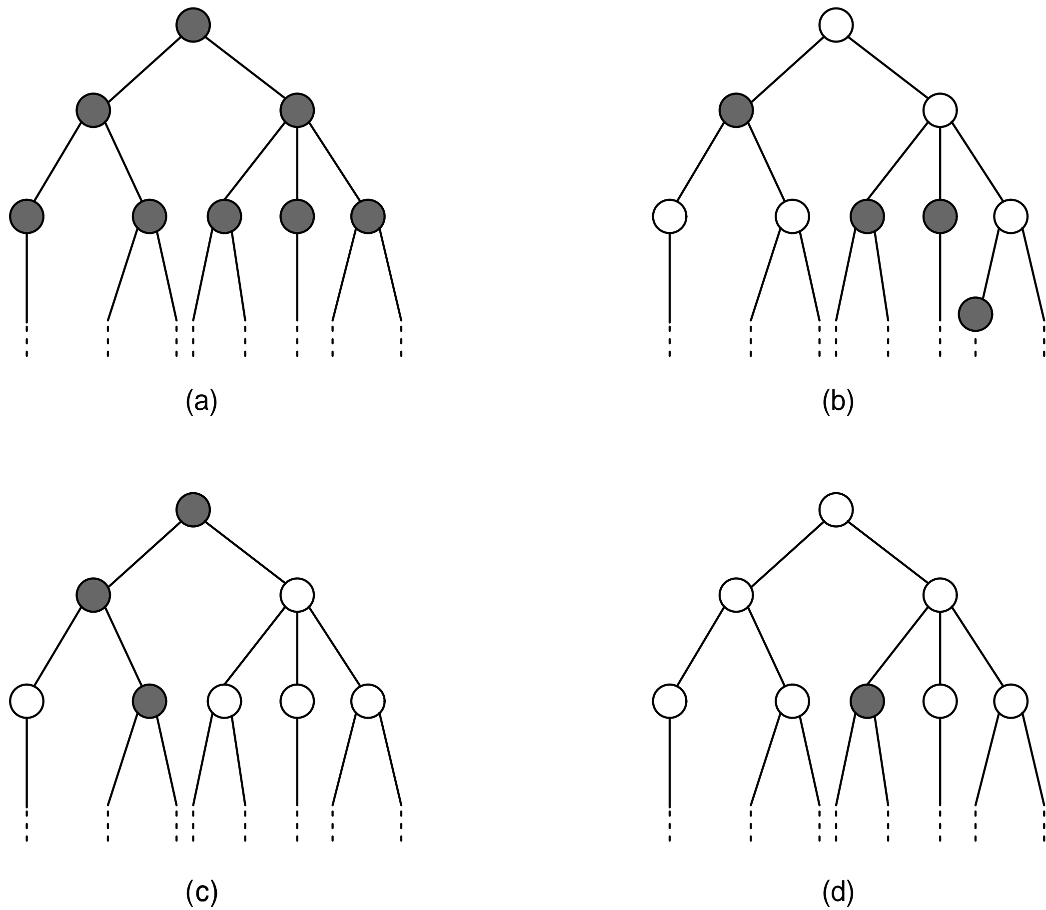


Figure 6: CTL Properties in a Computation Tree

4.2 Pros and Cons of Model Checking

Since invented, model checking has attracted a great deal of interest from both research and industry arenas. The use of model checking has proved to be very successful in verifying hardware designs and recently its application to software models has notably increased [CAB+98, EP02, CCMS02, MC01]. For example in [CCMS02, MC01], Cheng *et al* present a framework and a number of tools for translating class and state diagrams from the Unified Modelling Language (UML) to formal specifications that can be simulated and analysed by model checkers.

The appeal of model checking is justified by several advantages of this verification technique. First of all, it is fully automated and requires minimal effort on the part of the

user. All the user needs to do is supply the model checker with a system model and provide it with properties that need to be verified. After the commencement of the verification process, no further user interaction is necessary until the final results are computed. One of the important implications of this characteristic of model checking is the possibility of integrating model checkers as verification engines into other modelling tools. Such integration can provide users with a more accessible interface to verification features offered by model checkers. For instance, consider a UML modelling tool that interfaces with a model checker by translating a UML model into the input language required by the model checker and subsequently interpreting verification results in terms of the original graphical UML model. In such a scenario, the software developer can reap the benefits of powerful verification without having to know anything about the underlying complexities. The SusanX tool that we developed integrates the NuSMV model checker in this manner.

Another advantage of model checking is that if verification of a property fails, a counter-example is produced to demonstrate how the property is violated within the model. The counter-example trace can be very helpful in tracking down the source of errors in a model. Although, there are certain properties for which a counter-example cannot be produced. For instance, a model checker cannot produce a trace showing how a system violates a property that expresses the possibility of some event.

Lastly, the richness of temporal logic used to specify verification properties for model checking allows the user to express a wide variety of behavioural characteristics against which a model can be checked. A skilled user capable of constructing complex logic properties can utilise model checkers to their full extent. However, even if a model checker is used as a verification engine in another tool where the user is shielded from the details of the logic notation, it is still possible to allow the user to verify commonly sought behavioural properties. SusanX provides the user with several manageable options for controlling model verification performed with the NuSMV engine, as explained in Chapter 6.

The main drawback of model checking is its performance, in other words the time it takes to compute verification results. Since the model checking algorithm performs an exhaustive search of all the possible execution paths of the given model, verification time increases exponentially with the size of the model. This well-known phenomenon is named the *state explosion problem*.

A great deal of work aiming to improve the performance and scalability of model checking is currently in progress and significant advancements have already been made [MP04,

DKK02]. Many application areas such as verification of software systems could benefit from scalable and efficient model checkers.

4.3 Existing Model Checkers

Three main types of model checkers currently exist: *explicit* checkers, *symbolic* checkers based on *Binary Decision Diagrams (BDDs)* and symbolic checkers based on *satisfiability procedures (SAT)*. When first introduced, model checking algorithms were intended for explicit representations of system states stored in memory as lists and hash tables. During verification, access to state and transition information is very efficient in explicit model checking. However, model checkers with this type of state representation are greatly affected by the state explosion problem and memory requirements for verification quickly become intractable with increase in model size. The most eminent explicit model checkers are Spin [Hol97] developed at Bell Labs, Java PathFinder [VHB+03, LV01] from NASA and Murϕ [DDHY92] made at Stanford.

The invention of the symbolic approach [McM93, JEK+90] to model checking constituted a significant breakthrough in the field. Instead of explicitly representing each reachable state in memory, these model checkers work with sets of states instead. The most recognised structure for holding state information in symbolic model checking is BDDs. A BDD provides a compact means of representing boolean functions, and these functions are used to characterise multiple states from the system model. This approach considerably reduces the amount of memory required by verification and allows one to model check larger models. The first tool called Symbolic Model Verifier (SMV) implementing this technique was developed by McMillan at the Carnegie Mellon University [McM93]. It was then improved by two independent projects at Cadence Berkley Labs and ITC-IRST. The members of the ITC-IRST project called their enhanced model checker NuSMV [CCGR99, CCG+02] and are continuously developing and extending this tool with new features. Other symbolic model checkers include PRobabilistIc Symbolic Model checker (PRISM) [KNP02, KNP04] from the University of Birmingham and RuleBase [BBDEL96] from the IBM Haifa Research Labs.

Symbolic model checking with SAT presents another technique of obtaining verification results more efficiently. Propositional satisfiability procedures are used to check computation paths of bounded length instead of exploring the entire reachable state space. Bounded

Model Checkers (BMC) [BCC+99] that implement algorithms using SAT can be used for discovering errors in models, but not for proving that a model conforms to a specification property.

The Susan method is based on use case modelling and hence it represents the required behaviour of a system for different scenarios of use. We wanted to allow verification of Susan models, where the possibility of use case activations could be checked. Specification properties describing the possibility of events can be expressed with CTL, which is the predominant temporal logic used in model checking. Therefore, we needed to select a model checker that supported verification of CTL formulae for our research. Furthermore, we wanted to use an established tool that has been extensively used in practice. The NuSMV model checker met these requirements and thus we used it as our verification engine for Susan.

In our work we utilise NuSMV for verification of CTL properties only, however the tool also supports LTL specifications. Furthermore, recently NuSMV has been extended with BMC using SAT algorithms. We did not experiment with this feature of NuSMV, but rather used the standard symbolic model checking capabilities of NuSMV.

One may note that the "symbolic" part of the name Susan, which stands for "S"ymbolic "us" case "an"alysis, refers to our particular choice of the model checking technique. If an explicit model checker were to be used instead of a symbolic one in future research, this method name would need to be generalised.

4.4 NuSMV Input Language and Generated Counter-Examples

As explained already, verification with model checkers requires a description of a finite state system in some formal language and a specification of verification properties in temporal logic. The NuSMV tool that we used in our work prescribes its own input language for model description and accepts verification properties expressed in terms of CTL and LTL formulae. The NuSMV input language allows one to define valid transition relations between the states of a system. The system state itself is represented by *state variables*, which can only be of finite data types such as boolean, scalar and fixed array. During verification, the transitions defined in a NuSMV model are used to determine legal evolutions of the system and thereby compute verification results.

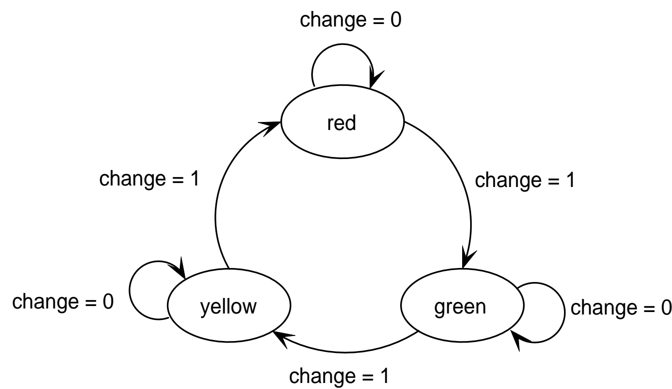


Figure 7: State Changes of a Traffic Light

We introduce the NuSMV language syntax with a few simple examples. Our first example is a basic model of state transitions for a traffic light, which is shown as a state transition diagram in Figure 7. The colour of our traffic light changes from green to yellow, yellow to red and then red to green again. In this model, the traffic light changes its colour when the trigger called *change* is set to 1. We labelled the transitions in the diagram above in order to make it more understandable, however it could easily be turned into a Kripke structure where only the states are labelled. This traffic light system can be represented by the following NuSMV model, or program as it is more commonly called.

```
1 MODULE main
2 VAR
3   change : boolean;
4   traffic_light : {red, yellow, green};

5 ASSIGN
6  init(traffic_light) := red;
7  next(traffic_light) :=
8    case
9        traffic_light = red & change = 1 : green;
10       traffic_light = yellow & change = 1 : red;
11       traffic_light = green & change = 1 : yellow;
12       1 : traffic_light;
13   esac;
```

A NuSMV program can consist of one or more modules, where exactly one *main* module must be present. The order of the modules in a program is irrelevant, but the entry point into the program is always defined by the main module. Our example program above comprises only a main module, which is declared in line 1. Inside the main module, there

are variable declarations starting with line 2 and assignment statements starting with line 5. Two variables are declared in this program: a boolean variable `change` and a scalar variable `traffic_light` that can take on three values (`red`, `yellow` and `green`). During model checking, the values of these two variables fully describe the state of this particular system.

The transition relations are expressed by defining the changes on the system's state variables in the assignment statements section of a module. Initial values are assigned to variables with the `init` keyword as in line 6, while the `next` keyword is used to specify how the value of a variable should change in the next system state as in line 7. In the example, initially `traffic_light` is assigned to `red`. After that, a case statement is used to conditionally assign the next value to this state variable. For instance, line 9 indicates that if the current value of `traffic_light` is `red` and `change` is equal to `1`, then the new value assigned to `traffic_light` is `green`. Line 12 simply suggests that in all other cases not explicitly listed in the case statement, the value of the `traffic_light` variable should remain unchanged. Note that no initial value is given to the `change` state variable. NuSMV assigns an initial value to such non-initialised variables nondeterministically.

Modular programming in NuSMV is very useful when describing more complicated systems. The language allows for parameterised modules that can be reused within the program. Our second example illustrates this and some additional NuSMV language features.

We now consider two synchronised traffic lights, which we label A and B. Both traffic lights change colour at the same time, but A is always one colour "ahead" of B since it is situated some distance before B on the road. For instance, if A is green then B must be red. The system controlling these two traffic lights is modelled in the state transition diagram in Figure 8.

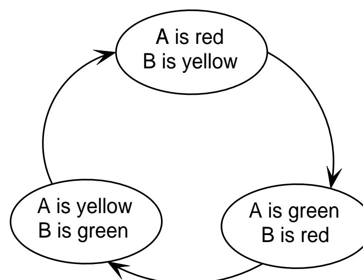


Figure 8: State Changes of Two Synchronised Traffic Lights

Note that no trigger is used in this example. The NuSMV program corresponding to this system is given below.

```
1 MODULE next_colour(current_colour)
2 VAR
3    colour : {red, yellow, green};

4 ASSIGN
5    colour :=
6        case
7            current_colour = red : green;
8            current_colour = yellow : red;
9            current_colour = green  : yellow;
10       esac;

11 MODULE main
12 VAR
13   traffic_light_A : {red, yellow, green};
14   traffic_light_B : {red, yellow, green};

15   change_B : next_colour(traffic_light_B);
16   change_A : next_colour(change_B.colour);

17 ASSIGN
18   init(traffic_light_A) := green;
19   next(traffic_light_A) := change_A.colour;

20   init(traffic_light_B) := red;
21   next(traffic_light_B) := change_B.colour;
```

In the above program, there is one reusable module called `next_colour` that captures the legal colour changes for one traffic light. It is instantiated twice by `change_B` and `change_A` in lines 15 and 16 respectively. Note that the `next_colour` module has one formal parameter called `current_colour`, which is assigned to the value of the `traffic_light_B` variable in its instantiation in line 15. The second instantiation uses `change_B.colour` as the actual parameter for the reusable module. This actual parameter references the `colour` variable declared and assigned inside the `next_colour` module, which is allowed in NuSMV. Hence the `change_A` module instance is essentially equivalent to `next_colour(next_colour (traffic_light_B).colour)`. In this way, we express the synchronisation of the two traffic lights.

Some additional NuSMV language features are introduced in the next chapter when they are used for the Susan to NuSMV mapping. Comments begin with `--` in NuSMV.

CTL specifications are inserted into the assignment statements section of a NuSMV program with a keyword `SPEC` preceding each property. For our second example, we may want to check that the two traffic lights in the model do not get "out of sync". One of the properties that need to be satisfied for this is `AG !( (traffic_light_A = red) & (traffic_light_B = green) )`. The NuSMV model checker reports that this property is in fact satisfied in the model.

When a property is not satisfied in the model, NuSMV produces a counter-example trace demonstrating the violation of the property. Let us construct a property that should be reported as false by the model checker. For example, `!EF (traffic_light_A = red & traffic_light_B = yellow)` states that it is not possible for A to be red and B yellow at the same time. The following trace is produced by NuSMV.

```
-- specification !EF (traffic_light_A = red & traffic_light_B =
yellow) is false
-- as demonstrated by the following execution
sequence -> State 1.1 <-
    traffic_light_A = green
    traffic_light_B = red
    change_B.colour = green
    change_A.colour = yellow
-> State 1.2 <-
    traffic_light_A = yellow
    traffic_light_B = green
    change_B.colour = yellow
    change_A.colour = red
-> State 1.3 <-
    traffic_light_A = red
    traffic_light_B = yellow
    change_B.colour = red
    change_A.colour = green
```

The counter-example trace consists of two state transitions and shows three states. For each state, the values of state variables are given. `State 1.1` is the initial state of the system, as can be seen `traffic_light_A` is assigned to `green` and `traffic_light_B` is assigned to `red`. After two changes of the lights, our property is violated in `State 1.3`.

We have now explained the fundamentals of both use case modelling and model checking. In the next chapter we describe how we bring these two techniques together to formulate Susan, our method for modelling and analysing software requirements.

Chapter 5

Susan Models and Verification

In this chapter we first present the Susan modelling notation and demonstrate it with a few examples. Next, we explain how Susan models are mapped to the NuSMV input language and how the generated NuSMV counter-example traces are interpreted in terms of Susan models. Finally, we discuss the verification options offered in the Susan method.

Before explaining how the actual use case notation is extended in Susan, we first present the general view on modelling system behaviour requirements that is assumed in the Susan method. In agreement with the standard use case approach, the system under consideration is treated as a "black box" in Susan. The diagram in Figure 9 illustrates the perspective on actor-system interaction taken by Susan, which is fundamental to the method.

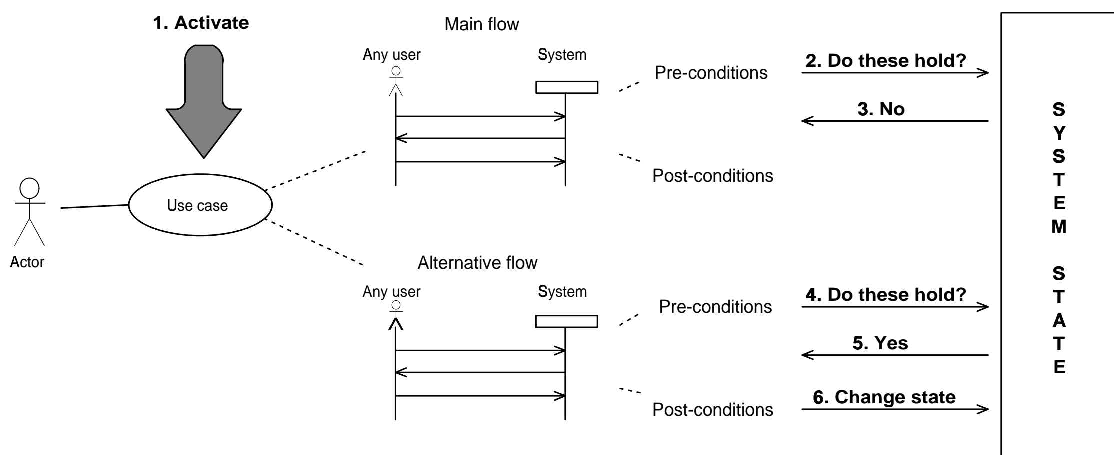


Figure 9: Interaction Between Actor and System in Susan

The actor can call upon the system's services by *activating* use cases. The system itself is described by the *system state*, which is dynamic and changes in time as a result of use case activations. Predicates referred to as *conditions* in Susan are used to collectively represent the state of the system, where at any time the system can be queried for the value of any one of these conditions. For instance, one of the features of the MuTI telemedicine system introduced in Chapter 3 is the storage of patients' medical records. One of the conditions describing the state of the MuTI system would specify whether a particular medical record exists within the system.

As already described in Chapter 3 and shown in Figure 9, each use case is associated with a number of flows and each flow has pre- and post-conditions. The diagram shows what happens during a use case activation with six numbered steps. After a use case is activated by an actor (1), the pre-conditions of its main flow are queried against the current system state (2). Suppose that these pre-conditions do not hold (3), then the alternative flow of the use case is considered. Pre-conditions of the alternative flow are queried (3) and this time they are satisfied in the state of the system (4). Since the pre-conditions are satisfied, the post-conditions of that flow are used to change the system state (5). When pre-conditions for one of the flows hold, the use case activation is said to be *successful*. Note that pre-conditions for only one or none of the flows could be satisfied by the system state at one time, hence the order in which the flows are checked is not essential.

As explained above, use case models become more dynamic in Susan than in the standard use case modelling approach. This new view of system requirements modelling allows Susan to incorporate verification with model checking that explores all the possible interactions between the actors and the system for a particular model.

5.1 Susan Metamodel

A "metamodel" is a model that describes semantics of some modelling language. This method for expressing the constructs of a language has become common in recent years as a result of increased popularity of graphical and object-oriented modelling [mof03]. We use a metamodel represented as a Unified Modelling Language (UML) class diagram in order to explain the Susan modelling notation.

We took the fundamental building blocks of models from the standard use case approach and appended them with additional elements to facilitate construction of executable Susan

models. The diagram in Figure 10 shows the Susan metamodel. Classes are used to represent the modelling elements and associations denote relationships between the elements. The labels in italics identify class roles, these show how one element can play different roles in different relationships. For example, the association between "Actor" and "Variable" should be interpreted as follows: "an Actor has zero or more attributes that are instances of the Variable class".

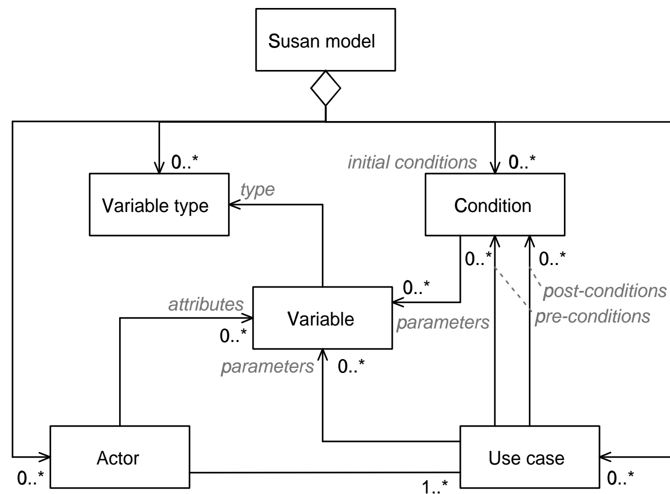


Figure 10: Susan metamodel.

The aggregation relationships in Figure 10 show that a Susan model comprises four different types of elements: actors, use cases, conditions and variable types. For each modelling element the Susan metamodel prescribes a number of *properties*, which are similar to class attributes in UML. The remaining element, variable, is auxiliary; it assists in defining properties for the main four elements.

At the center of a Susan model there is a use case diagram that shows a graphical representation of actors, use cases and their associations. For each actor and use case in the diagram, textual properties are additionally defined. Conditions and variable types do not have graphical representations; these elements are completely textual. In the examples throughout the dissertation we consistently use a readable syntactical form for Susan modelling elements. However, our emphasis here is on the conceptual and semantic level of Susan and hence we do not discuss the details of concrete syntax for the notation.

Each part of the Susan metamodel is described next. Since the Susan semantics is closely related to the way in which verification is performed on the models, we refer to verification

or model execution on a conceptual level during the following discussion.

Variable types: Each variable type in Susan is associated with a finite number of *symbolic values*, which are essentially string literals that can only be compared for equivalence. Two symbolic variables are equal if their values are set to identical string literals. Consider the following two variable type definitions from the MuTI example.

Variable type 1.
Name: User ID
Values: Doctor Bosman, Sister Mary, Sister Nhlanga

Variable type 2.
Name: MuTI address
Values: Clinic A, Clinic B, Hospital

As can be seen above, a variable type is described by the name and *values* properties. The cardinality of a variable type or the number of values assigned to it must be kept to a relatively small number, as it affects the time taken by verification with model checking. For our MuTI example, we chose a set of test values for each of the required variable types.

Actors: The Susan actor element is based on actors in the standard use case modelling. However, the actor-use case association is slightly more restrictive in Susan. The multiplicity for this association is "one to many" (1 to 1..*), which means that a use case can only be associated with one actor.

Susan defines two properties for an actor: a name and a list of *attributes*. Attributes describe an actor's particulars that the system needs to access in order to deliver services to that actor. In MuTI, every user is identified by a User ID and hence we have the following definition for the *Any user* actor.

Actor 1.
Name: Any user
Attributes: ID *of type* User ID

Each actor attribute is regarded as a variable in Susan, and each variable has an associated *type* as shown in the metamodel in Figure 10. In the definition of Actor 1 above, the ID attribute is associated with the User ID variable type defined earlier.

Conditions: Conditions are used to describe the global state of the system and to declare use case pre- and post-conditions. Three properties are defined for a Susan condition: a name, a *parameter* list and a *truth-value*. Consider the following two conditions that are declared in the MuTI model and used to describe its state.

Condition 1.
Name: User logged in
Parameters: User *of type* User ID, Location *of type* MuTI address

Condition 2.
Name: Network available
Parameters: Location *of type* MuTI address

The reader may note that in the two definitions above there are no truth-values given to the conditions. This is because a condition only becomes true or false at the time of system verification, but at declaration time we only specify its name and parameters. During verification, condition parameters are also assigned literal values. A condition with a truth-value and all its parameters assigned to literal values is called a *condition instance*. From the above definitions, Condition 2 has three instances during model verification: *Network available(Clinic A)*, *Network available(Clinic B)* and *Network available(Hospital)*. By considering the truth-values of each of these instances we can determine the network availability at these three locations.

A number of *initial conditions* may be defined in a Susan model. These are condition instances that hold or are true at the very beginning of system execution. By using initial conditions we can specify that the network is initially available at all three locations in our model.

Use cases: As in the standard approach, use cases represent functional system requirements. In Susan, a use case has five properties: a name, its associated actor, a parameter list, pre-condition and post-conditions lists. Use case parameters describe information that is required by the system to provide the corresponding service. When a use case is activated, a literal value for each of its parameters is passed to the system.

Here are the properties that we define for the *Log in* use case in our Susan MuTI model.

Use case 1.
Name: Log in
Actor: User

Parameters: Location *of type* MuTI address
Pre-conditions: User logged in(*#self* ID, *#uc* Location) *is* false
Post-conditions: User logged in(*#self* ID, *#uc* Location) *is* true

As can be seen in the above definition, the *User logged in* pre- and post-conditions correspond to a condition that we declared earlier for the state description. As these pre- and post-conditions are queried against the system state during verification, it is necessary to indicate what values must be used during their evaluation. Two possible options for this are demonstrated in the example above. The *#self* prefix indicates that this parameter must take on the value of the actor attribute with the same name. Hence, *#self ID* refers to the value of the *ID* attribute of the *User* actor. The prefix *#uc* indicates that the parameter must take on the value of the parameter for this use case with the corresponding name. In the example, *#uc Location* specifies that the value of the *Location* parameter for this use case must be used for the evaluation of the condition.

There are two further options for assigning pre- and post-condition parameters for a use case. Firstly, a parameter can be given a literal value from the corresponding type. Secondly, the *#forall* prefix can be used to indicate that the condition must take effect for all the values in this variable type. For instance, a pre-condition *Network available(#forall MuTI address) is true* means that the network must be available at all locations.

The concept of condition parameters in Susan is comparable to *formal* and *actual* parameters in programming languages. For example, a Java method definition contains a formal parameter list, where the type of each parameter is specified. A method call supplies actual parameters to the method. Eventually, at runtime all the parameters are bound to literal values. In Susan, a condition declaration defines formal parameters for that condition and their variable types. When that condition is used as a pre- or post-condition for a use case, the user assigns each of the formal parameters to an actual parameter as described above. During the verification of the system model, all the possible use case activations are simulated. When a use case activation is simulated, the attributes of the associated actor and use case parameters are assigned literal values. These values are then propagated to fill the pre- and post-condition parameters of the use case. Once the pre- and post-conditions have all their parameters assigned, pre-conditions can be queried against the current system state and post-conditions used to alter it. A use case with values assigned to its parameters and the attributes of its associated actor is called a *use case instance*.

In the *Log in* use case example above, there was only one flow and thus one set of pre-

and post-conditions. If a use case has alternative flows, a pre- and post-condition set for each flow is included in the use case definition in Susan. All the pre- and post-condition sets for a use case are implicitly joined with the OR logical operator when they are used during verification.

From the diagram in Figure 10 it can be seen that Susan does not support relationships among use cases or actor generalisation relationships. In its current state the Susan method is built around the fundamental features of use case models only, as our goal was to test the approach first before incorporating the additional use case modelling features. Before explaining how Susan currently deals with use case relationships and actor generalisations, we first present a simple example of a complete Susan model.

5.1.1 A Simple Susan Model

In the previous section some individual elements from the Susan model for the MuTI system were shown. We now take just three MuTI use cases to create a rudimentary and yet complete Susan model for illustrative purposes. The use case diagram for this model is shown in Figure 11.

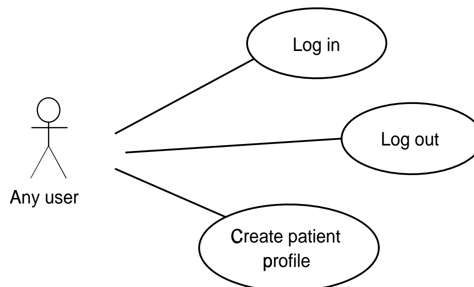


Figure 11: A Simple Use Case Diagram

The following textual descriptions together with the above use case diagram form the Susan model for this example.

Variable type 1.
Name: User ID
Values: Dr Bosman, Sister Mary, Sister Nhlanga

Variable type 2.
Name: MuTI address

Values: Clinic A, Clinic B, Hospital

Variable type 3.
Name: Patient ID
Values: Teboho Luan, Nicolas Brooks

Condition 1.
Name: User logged in
Parameters: User *of type* User ID, Location *of type* MuTI address

Condition 2.
Name: Patient profile exists
Parameters: Owner *of type* User ID, Location *of type* MuTI address, Patient *of type* Patient ID

Actor 1.
Name: Any user
Attributes: ID *of type* User ID

Use case 1.
Name: Log in
Actor: Any user
Parameters: Location *of type* MuTI address
Pre-conditions: User logged in (#*self* ID, #*uc* Location) *is* false
Post-conditions: User logged in (#*self* ID, #*uc* Location) *is* true

Use case 2.
Name: Log out
Actor: Any user
Parameters: Location *of type* MuTI address
Pre-conditions: User logged in (#*self* ID, #*uc* Location) *is* true
Post-conditions: User logged in (#*self* ID, #*uc* Location) *is* false

Use case 3.
Name: Create patient profile
Actor: Any user
Parameters: Patient *of type* Patient ID, Location *of type* MuTI address
Pre-conditions: User logged in (#*self* ID, #*uc* Location) *is* true,
Patient profile exists (#*self* ID, #*uc* Location, #*uc* Patient) *is* false
Post-conditions: Patient profile exists (#*self* ID, #*uc* Location, #*uc* Patient) *is* true

Most of the elements in the model above have been discussed before during the Susan metamodel explanation. There is one new variable type called *Patient ID*, which is used to distinguish between the different patients in the system. The *Patient profile exists* condition allows one to monitor the patient profiles existing within the system. Each patient profile is identified within the system by its *Owner*, *Location* and the *Patient* in question. The *Log out* use case is the same as the *Log in* use case, except the pre- and post-conditions are reversed. Lastly, the *Create patient profile* use case requires the identity of the *Patient*

under consideration as well as the *Location* where the profile is to be created. The pre-conditions of this use case indicate that the user must be logged in and a profile with the same details must not already exist within the system. After a successful activation of this use case, a new patient profile is created as indicated by the post-condition.

Note that in a complete Susan model the type property of all the actor attributes, condition parameters and use case parameters must be set to a variable type declared in the model. All the use case pre- and post-conditions must correspond to declared condition elements. In this particular example there are no initial conditions.

In the model for the MuTI system we additionally need to differentiate between a user being logged in and a user being online, where the latter is only possible if the network is available at the user's location. This distinction is necessary as some services such as call placement can only be accessed if the user is online. We add the following condition definitions to our simple Susan model discussed above.

Condition 3.
Name: User online
Parameters: User *of type* User ID, Location *of type* MuTI address

Condition 4.
Name: Network available
Parameters: Location *of type* MuTI address

Furthermore, we adjust the definition of the *Log in* use case in such a way that if the network is available at the time this use case is activated, the user's status is automatically set to "online". An alternative flow and hence a set of pre- and post-conditions is used to reflect this in the model. The post-conditions of the *Log out* use case are also altered to ensure that the user's status changes to "offline" when he logs out.

Use case 1.
Name: Log in
Actor: Any user
Parameters: Location *of type* MuTI address
Flow 1, Pre-conditions: User logged in (#*self* ID, #*uc* Location) *is* false,
Network available (#*uc* Location) *is* true
Flow1, Post-conditions: User logged in (#*self* ID, #*uc* Location) *is* true,
User online (#*self* ID, #*uc* Location) *is* true
Flow 2, Pre-conditions: User logged in (#*self* ID, #*uc* Location) *is* false,
Network available (#*uc* Location) *is* false

Flow 2, Post-conditions: User logged in (*#self* ID, *#uc* Location) *is* true

Use case 2.
Name: Log out
Actor: Any user
Parameters: Location *of type* MuTI address
Pre-conditions: User logged in (*#self* ID, *#uc* Location) *is* true
Post-conditions: User logged in (*#self* ID, *#uc* Location) *is* false,
User online (*#self* ID, *#uc* Location) *is* false

For the verification, we would also want to model the changes in network availability at different locations. In Susan we can do this by introducing a *Dummy* actor into the model and associating a use case with it that changes the value of the *Network available* condition. We add the following definitions to our model.

Actor 2.
Name: Dummy

Use case 4.
Name: Change network availability
Actor: Dummy
Parameters: Location *of type* MuTI address, User *of type* User ID
Flow 1, Pre-conditions: Network available (*#uc* Location) *is* true
Flow 1, Post-conditions: Network available (*#uc* Location) *is* false,
User online (*#uc* User) *is* false
Flow 2, Pre-conditions: Network available (*#uc* Location) *is* false,
User logged in (*#uc* User, *#uc* Location) *is* true
Flow 2, Post-conditions: Network available (*#uc* Location) *is* true,
User online (*#uc* User) *is* true
Flow 3, Pre-conditions: Network available (*#uc* Location) *is* false
Flow 3, Post-conditions: Network available (*#uc* Location) *is* true

In the use case definition above, there are three flows and hence three sets of pre- and post-conditions. The first flow represents the scenario where the network is available at the time of use case activation. The post-conditions for this flow state that the network becomes unavailable and the status of the user logged in at that location changes to “offline”. Note that if there is no user logged in at the location at the time of use case activation, the second post-condition has not effect on the system state. The second use case flow deals with the scenario where the network is not available and there is a user logged in at that specific location. The last flow represents the situation where the network is unavailable, but no user is logged in. Having the *Dummy* actor and such a *Change network availability*

use case in the model, allows us to test the behaviour of the system under different network availability conditions. Conditions that change their value as a result of some event external to the model can be handled in this way in Susan.

5.1.2 Flattening Use Case Models for Susan

In order to model our full MuTI system in Susan, it is necessary to remove the use case relationships and actor generalisations from the original use case diagram that we proposed in Figure 3, Section 3.3. At the same time, we would like to retain as much semantics of these relationships in the model as possible. We next describe how to eliminate each of these four relationships from a use case model.

Use case generalisation: In such a relationship, the general use case is abstract while the concrete behaviour of the system is captured by the use case specialising it. For instance, consider the general use case *Manage patient profiles* and the three use cases specialising it in Figure 3, Section 3.3. This relationship adds structure to the use case diagram, but the model without it still represents the same behaviour. When creating a Susan model from a standard use case diagram, only specialised use cases are included.

Use case include: Included use cases are eliminated in Susan models and hence one cannot show shared behaviour between use cases. In the MuTI example, the *Validate user* use case is removed from the use case model while the *Log in* use case remains.

Use case extend: When two use cases are joined with the extend relationship, they are both included in the Susan model. The semantics of the relationship are preserved through declaring pre- and post-conditions on these use cases that state that the extending use case can only be activated after the extended one. For example, in the MuTI system the *End call* use case extends the *Place call* use case. For the Susan model, we remove the extends relationship and ensure that a post-condition of the main flow through the *Place call* use case states that a call gets established and make this also the pre-condition for the *End call* use case flows. There is no other way of establishing calls within the MuTI model and therefore the *End call* use case can only be activated after a successful *Place call* activation. It is also possible that the extension point for the extend relationship appears somewhere in the middle of a use case rather then at the end. In this case, the extended use case has to be split into two use cases in the Susan model and then once again pre- and post-conditions can be used to capture the semantics of the relationship.

Actor generalisation: When actor generalisation is used in a use case model, only the general actor is carried through into the Susan model. A condition is then defined in the model that has the identifying actor attributes as parameters and can be used to distinguish which of the specialised actors a particular general actor instance represents. Each of the use cases connected with the specialised actors get associated with the general actor, but a pre-condition that checks the identity of the actor each time is added to each of these use cases. For our MuTI example, we add a *User is doctor* condition to the model with a parameter of *User ID* type. The *Doctor* actor is then removed from the model and the *Accept appointment* use case gets associated with the *Any user* actor. We add the following pre-condition to this use case: *User is doctor(#self ID) is true.* The *Nurse* actor is removed from the model in the same way.

We call this process of eliminating the unsupported relationships from a use case model "flattening". The flattened use case diagram for the MuTI Susan model is shown in Figure 12 below.

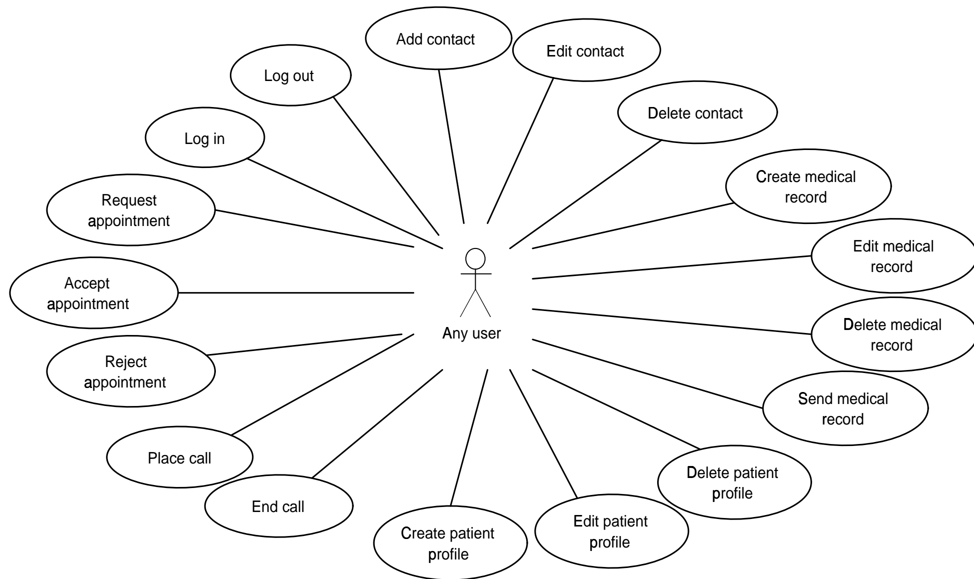


Figure 12: Flattened MuTI Use Case Diagram.

We have now fully presented the Susan notation and its semantics. The next section describes how a Susan model is mapped to a NuSMV program.

5.2 Mapping Susan Models to NuSMV

A Susan model can be seen as a finite state machine, where values of condition instances define the system state and state transitions are defined by activation of use case instances. Initial conditions in a Susan model give the initial state of the system. In our mapping to NuSMV, we represent each condition instance in a Susan model as a state variable. These condition variables are initialised in accordance to the initial conditions in the Susan model. For each use case instance, a NuSMV module is defined inside which "next" values are assigned to the condition variables as indicated by the pre- and post-conditions of the use case. The overall structure of a NuSMV program representing a Susan model is given below.

```
MODULE main
VAR
-- (1) Declare condition variables.
-- (2) Instantiate a process for each use case instance module.
-- (3) Instantiate a process for the dummy module.

ASSIGN
-- (4) Initialise condition variables.
-- (5) Specify CTL properties.

-- (6) Define a module for each use case instance.
-- (7) Include dummy module.
```

The main module of the program contains condition variable declarations (1) and initialisations (4). In addition to the main module, a module for each use case instance is defined (6). Each use case instance module is instantiated as a *process* in the main module (2). Using processes in NuSMV ensures that during verification, these modules are instantiated nondeterministically. Each instantiation represents a use case instance activation. Nondeterministic choice between activations allows us to check all the possible ways in which the system can be used.

Sometimes a Susan model can contain condition instances that remain constant throughout all possible use case activations. In such a case, the condition variable in the NuSMV program is re-assigned to new values nondeterministically by the NuSMV engine during verification. We counteract this undesirable situation by defining a `dummy` module that simply re-assigns each of these constant condition variables to its initial value (7). The `dummy` module is also instantiated as a process in the main module (3).

Finally, the NuSMV program needs logic specification properties to perform verification. These properties appear in the assignment section of the main module (5). The way in which these are generated is explained in Sections 5.4 and 5.5.

Each use case instance module is structured as follows.

```
MODULE Use_case_instance(condition_variable_1,condition_variable_2,...)
VAR
    return : boolean;

ASSIGN
    init(return) := 0;
    next(return) :=
    -- check pre-conditions using condition variables passed as parameters
    -- if preconditions hold, assign next return value to 1

    -- for each post-condition
    --      check pre-conditions using condition variables passed as parameters
    --      if preconditions hold, assign next condition variable value according
    --      to post-condition

FAIRNESS
    running;
```

Inside a use case instance module, the pre-conditions of the use case instance are checked. This is done by considering the values of the corresponding condition variables. Passing the appropriate condition variables to each use case instance module as parameters provides the modules access to the values of these variables. Additionally, condition variables for the post-conditions of a use case instance also need to be passed to its module as they get re-assigned there. In the module structure above, the passing of parameters into the module is shown by `(condition_variable_1,condition_variable_2,...)` next to the module name.

As can be seen in the above module structure, a boolean variable called `return` is declared inside each use case instance module. This variable is used in the main module to determine whether a use case activation represented by the use case instance module is successful or not. This variable is first initialised to `0` and if the pre-conditions of the use case are met then its value is re-assigned to `1`. Hence after a process is created for the `Use_case_instance` module during verification, we can find out whether this activation succeeded or failed by checking the value of `Use_case_instance.return`.

The NuSMV `FAIRNESS` clause is included in each use case instance module and the `dummy` module to ensure that during verification each of these modules is chosen equally

often for instantiation.

Names for condition variables and use case instance modules are generated in the following way. Firstly, the spaces are taken out of the condition or use case name. For example, an instance of the *User logged in* condition first becomes `Userloggedin`. Secondly, values for condition and use case parameters are substituted by the indices in their variable types and appended to the generated name with dollar sign separators ($). The condition instance *User logged in (Sister Mary, Clinic A)* will be represented by `Userloggedin$1$0$` since *Sister Mary* is the second value in the *User ID* type and *Clinic A* is the first value in the MuTI address type (indices begin with `0`). For a use case instance, the values of the associated actor's attributes are also appended to the name. For the instance that represents *Dr Bosman* logging in at the *Hospital*, the use case module name is `Login$0$2$`. The actor attributes indices appear before the indices for the use case parameters. This scheme provides for generation of concise NuSMV names that are unique.

For the simple Susan model example presented in Section 5.1.1 on page 55 we get the following NuSMV program.

```
MODULE main
VAR
-- (1) Declare condition variables.

    Userloggedin$0$0$ : boolean;
    ...
    Userloggedin$2$2$ : boolean;

    Patientprofileexists$0$0$0$ : boolean;
    ...
    Patientprofileexists$2$2$1$ : boolean;

-- (2) Instantiate a process for each use case instance module.

    activated_Login$0$0$ : process Login$0$0$(Userloggedin$0$0$);
    ...
    activated_Login$2$2$ : process Login$2$2$(Userloggedin$2$2$);

    activated_Logout$0$0$ : process Logout$0$0$(Userloggedin$0$0$);
    ...
    activated_Logout$2$2$ : process Logout$2$2$(Userloggedin$2$2$);

    activated_Createpatientprofile$0$0$0$ : process
    Createpatientprofile$0$0$0$(Userloggedin$0$0$,Patientprofileexists$0$0$0$);
    ...
    activated_Createpatientprofile$2$1$2$ : process
```

```
    Createpatientprofile$2$1$2$(Userloggedin$2$2$,Patientprofileexists$2$2$1$);

-- (3) Instantiate a process for the dummy module.

    activated_dummy : process dummy_module();

ASSIGN
-- (4) Initialise condition variables.

    init(Userloggedin$0$0$) := 0;
    ...
    init(Userloggedin$2$2$) := 0;

-- (5) Specify CTL properties.
    ...

-- (6) Define a module for each use case instance.

MODULE Login$0$0$(Userloggedin$0$0$)
VAR
    return : boolean;
ASSIGN
    init(return) := 0;
    next(return) :=
        case
            (!Userloggedin$0$0$) : 1;
            1 : 0;
        esac;
    next(Userloggedin$0$0$) :=
        case
            !Userloggedin$0$0$ : 1;
            1 : Userloggedin$0$0$;
        esac;
FAIRNESS
    running;

...

MODULE Logout$2$0$(Userloggedin$2$0$)
VAR
    return : boolean;
ASSIGN
    init(return) := 0;
    next(return) :=
        case
            (Userloggedin$2$0$) : 1;
            1 : 0;
        esac;
```

```
    next(Userloggedin$2$0$) :=
        case
            Userloggedin$2$0$ : 0;
            1 : Userloggedin$2$0$;
        esac;
FAIRNESS
    running;

...

MODULE Createpatientprofile$0$1$2$(Userloggedin$0$2$,Patientprofileexists$0$2$1$)
VAR
    return : boolean;
ASSIGN
    init(return) := 0;
    next(return) :=
        case
            (Userloggedin$0$2$ & !Patientprofileexists$0$2$1$) : 1;
            1 : 0;
        esac;
    next(Patientprofileexists$0$2$1$) :=
        case
            Userloggedin$0$2$ & !Patientprofileexists$0$2$1$ : 1;
            1 : Patientprofileexists$0$2$1$;
        esac;
FAIRNESS
    running;

-- (7) Include dummy module.

MODULE dummy_module()
ASSIGN
FAIRNESS
    running;
```

In order to perform verification on the program above, Computational Tree Logic (CTL) properties need to be inserted in the specified location. Suppose that we need to check whether *Sister Mary* can log out of the MuTI application at the *Hospital*. The indices of *Sister Mary* and *Hospital* in the corresponding variable types are `1` and `2` respectively, thus for this property the use case instance of interest is `Logout$1$2$`. In fact rather than the moudule itself, we need to consider the process instantiating it which is called `activated_Logout$1$2$`. The CTL formula that we insert into the NuSMV program as the specification is `EF activated_Logout$1$2$.return`. This will determine whether there exists a path in the computation tree for the program such that the value

of `activated_Logout$1$2$.return` is equal to `1` in one of the states on that path. Put simply, it checks the possibility of `activated_Logout$1$2$.return` being assigned to the value of `1`. After performing the check on the model, NuSMV reports that this property is true.

There is an alternative way in which we can determine whether the property considered above holds. If the negation of the CTL formula that we just used does not hold in the model then *Sister Mary* can in fact log out at the *Hospital*. The negation of this property is `AG !activated_Logout$1$2$.return` and NuSMV produces the following trace to show that this specificaiton is violated in the model.

```
-- specification AG (!activated_Logout$1$2$.return) is false
-- as demonstrated by the following execution sequence
-> State 1.1 <-
    [executing process activated_Login$0$2$]
    Userloggedin$0$0$ = 0
    Userloggedin$0$1$ = 0
    Userloggedin$0$2$ = 0
    Userloggedin$1$0$ = 0
    Userloggedin$1$1$ = 0
    Userloggedin$1$2$ = 0
    Userloggedin$2$0$ = 0
    Userloggedin$2$1$ = 0
    Userloggedin$2$2$ = 0
    Patientprofileexists$0$0$0$ = 0
    Patientprofileexists$0$0$1$ = 0
    Patientprofileexists$0$1$0$ = 0
    Patientprofileexists$0$1$1$ = 0
    Patientprofileexists$0$2$0$ = 0
    Patientprofileexists$0$2$1$ = 0
    Patientprofileexists$1$0$0$ = 0
    Patientprofileexists$1$0$1$ = 0
    Patientprofileexists$1$1$0$ = 0
    Patientprofileexists$1$1$1$ = 0
    Patientprofileexists$1$2$0$ = 0
    Patientprofileexists$1$2$1$ = 0
    Patientprofileexists$2$0$0$ = 0
    Patientprofileexists$2$0$1$ = 0
    Patientprofileexists$2$1$0$ = 0
    Patientprofileexists$2$1$1$ = 0
    Patientprofileexists$2$2$0$ = 0
    Patientprofileexists$2$2$1$ = 0
    activated_Login$0$0$.return = 0
    activated_Login$1$0$.return = 0
    activated_Login$2$0$.return = 0
    activated_Login$0$1$.return = 0
```

```
    activated_Login$1$1$.return = 0
    activated_Login$2$1$.return = 0
    activated_Login$0$2$.return = 0
    activated_Login$1$2$.return = 0
    activated_Login$2$2$.return = 0
    activated_Logout$0$0$.return = 0
    activated_Logout$1$0$.return = 0
    activated_Logout$2$0$.return = 0
    activated_Logout$0$1$.return = 0
    activated_Logout$1$1$.return = 0
    activated_Logout$2$1$.return = 0
    activated_Logout$0$2$.return = 0
    activated_Logout$1$2$.return = 0
    activated_Logout$2$2$.return = 0
    activated_Createpatientprofile$0$0$0$.return = 0
    activated_Createpatientprofile$1$0$0$.return = 0
    activated_Createpatientprofile$2$0$0$.return = 0
    activated_Createpatientprofile$0$0$1$.return = 0
    activated_Createpatientprofile$1$0$1$.return = 0
    activated_Createpatientprofile$2$0$1$.return = 0
    activated_Createpatientprofile$0$0$2$.return = 0
    activated_Createpatientprofile$1$0$2$.return = 0
    activated_Createpatientprofile$2$0$2$.return = 0
    activated_Createpatientprofile$0$1$0$.return = 0
    activated_Createpatientprofile$1$1$0$.return = 0
    activated_Createpatientprofile$2$1$0$.return = 0
    activated_Createpatientprofile$0$1$1$.return = 0
    activated_Createpatientprofile$1$1$1$.return = 0
    activated_Createpatientprofile$2$1$1$.return = 0
    activated_Createpatientprofile$0$1$2$.return = 0
    activated_Createpatientprofile$1$1$2$.return = 0
    activated_Createpatientprofile$2$1$2$.return = 0
-> State 1.2 <-
    [executing process activated_Login$1$2$]
    Userloggedin$0$2$ = 1
    activated_Login$0$2$.return = 1
-> State 1.3 <-
    [executing process activated_Logout$1$2$]
    Userloggedin$1$2$ = 1
    activated_Login$1$2$.return = 1
-> State 1.4 <-
    [executing process activated_Login$0$2$]
    Userloggedin$1$2$ = 0
    activated_Logout$1$2$.return = 1
```

The trace above corresponds to a number of use case instance activations. The transition from `State 1.1` to `State 1.2` is initiated by the activation of *Dr Bosman.Login(Hospital)*

as shown by `[executing process activated_Login$0$2$]`. The fact that this use case instance is activated does not necessarily mean that the activation was successful. It is only successful in the case where the `return` variable for the corresponding module is assigned to `1` in the next state. As you can see, in `State 1.2` the `return` variable for the `Login$0$2$` is indeed assigned to `1`, which is shown by line `activated_Login$0$2$.return = 1`. Hence the activation of *Dr Bosman.Login(Hospital)* was successful.

In this way we interpret the above counter-example trace as follows.

1. Dr Bosman.Log in(Hospital) – successful
2. Sister Mary.Log in(Hospital) – successful
3. Sister Mary.Log out (Hospital) – successful
4. Dr Bosman.Log in (Hospital) – unknown

The last use case instance activation in the trace can be ignored, as the specification property is actually violated by the second-last step already. In this case, as soon as *Sister Mary* successfully logs out of the *Hospital*, our property is shown to be false. Furthermore, it can be seen that generated traces often contain some redundant information. For example, it is not necessary for us to know that *Dr Bosman* logged in at the *Hospital* as shown by the first step in the counter-example. As a consequence, counter-example traces sometimes require close inspection to determine which steps are indeed of essence.

The remaining sections of this chapter present the different analysis options incorporated into the Susan method.

5.3 Overview of Susan Verification

We developed two modes of verification for the Susan technique: generic and model-specific. An overview of how verification is performed with NuSMV and the SusanX tool is given in Figure 13.

The mappings from Susan to NuSMV described in this chapter are used to translate Susan models to NuSMV programs, as shown in Figure 13. Generic verification can be applied to any Susan model and the CTL properties for this verification mode are embedded into SusanX. They are simply parameterised for the current model and passed to the NuSMV model checker as shown in the diagram. SusanX provides a number of specification

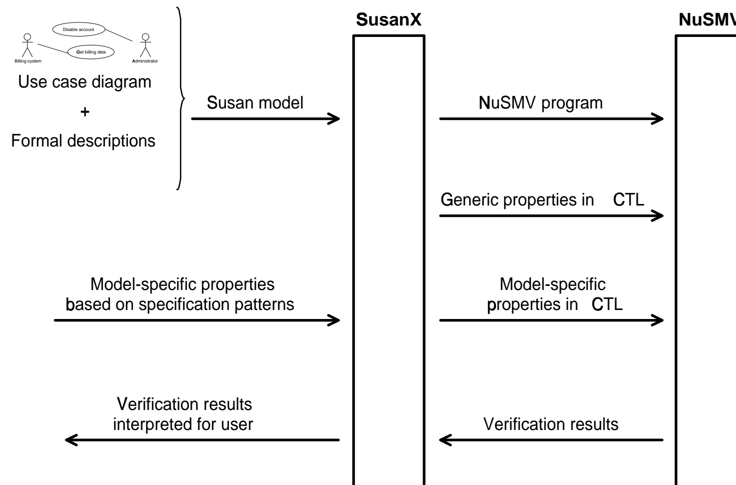


Figure 13: Overview of Verification in Susan

patterns that assist the user in constructing model-specific properties for verification. As can be seen, these are translated to CTL by SusanX. Finally, verification results are interpreted for the user in terms of the original Susan model. The details of the two verification modes are described in the following sections on a conceptual level, while the SusanX tool is presented in the following chapter.

5.4 Generic Verification of Susan Models

Generic verification can be applied to any Susan model irrespective of the type of system being modelled. A whole Susan model is checked by NuSMV against a number of predefined generic CTL properties and the results of this verification provide useful information about every use case and condition in the model. More specifically, these generic properties are used to analyse use cases for *liveness* and conditions for *reversibility* as described below.

5.4.1 Liveness of Use Cases

An informal definition of the liveness property is that "something good will always eventually happen" [Kin94]. We defined three liveness categories for a use case in Susan: *Dead*, *Transient* and *Live*. For each one of these three categories a CTL property testing whether a use case instance falls into it is defined. The liveness category for each use case instance is determined by checking it against the three properties. Hence, results of generic verification

place each use case instance in one of the liveness categories.

Each of the use case liveness categories is explained next and for each one the corresponding CTL formula is given. In the explanation below we consider determining the category for the Login$0$1$_activated process that represents an instance of the *Log in* use case.

Dead: Successful activation of the use case instance is not possible. Usually, one should be alarmed if all instances of a use case fall into the *Dead* category, because a use case that can never be successfully activated serves no purpose in a model. This scenario occurs when one of the use case pre-conditions can not become true during the entire model execution. The *Log in* instance falls into the *Dead* category if the CTL formula below is determined to hold by the model checker.

```
!EF (Login$0$1$_activated.return)
```

The above formula states that it is never possible for the `Login$0$1$_activated.return` variable to be assigned to `1`.

Transient: It is possible to successfully activate the use case instance a finite number of times. A typical example of this would be something that only happens once and is irreversible, for example *Delete patient profile* can only be done once unless the patient profile is recoverable. *Transient* use cases can place a limitation on the system functionality and one should be sure that the use case irreversibility is actually intended. The following CTL formula tests the *Log in* use case instance against this category.

```
EF (Login$0$1$_activated.return) & !AG EF (Login$0$1$_activated.return)
```

The left hand side of the conjunction above states that it must be possible for the *Log in* use case instance to be activated. While the right hand side states that at some state in each path of the computation tree it becomes impossible for the use case instance to be activated again.

Live: It is possible to activate the use case instance an infinitely many times. Most use

case instances in a model usually fall into this category. For example, all instances of *Log in* and *Log out* use cases are *Live* since all users can log in and out of MuTI infinitely often. The CTL formula for the *Live* category is as follows.

```
AG EF (Login$0$1$_activated.return)
```

The formula above states that at any state in the model computation tree, it is possible for `Login$0$1$_activated.return` to become `1` on one of the future computation paths.

5.4.2 Reversibility of Conditions

During generic verification we also test how condition instances change their truth-values throughout system execution. Each condition instance is placed into one of the following reversibility categories. The CTL formulae are given assuming that we are determining the reversibility category for the *Patient profile exists* condition instance represented by the `Patientprofileexists$1$1$0$` condition variable in the NuSMV program. It is important to note that initially the truth-value of this condition instance is false.

Constant: The truth-value of the condition instance never changes, it remains the same as assigned initially. This may be both desirable and undesirable. For instance, our full MuTI model contains a condition *User is doctor* with one parameter of type *User ID*. Also to indicate that *Dr Bosman* is in fact a doctor, we have this initial condition: *User is doctor(Dr Bosman)*. This condition instance must retain its initial truth-value throughout the model execution, hence it rightfully belongs to the *Constant* category. On the other hand, if all the instances of the *Call established* condition were to be reported *Constant* it would mean that it is impossible to establish a call in the model.

The following CTL formula tests whether the *Patient profile exists* instance is Constant.

```
!EF (Patientprofileexists$1$1$0$)
```

Irreversible: In this case the truth-value of the condition instance is changed once and then remains constant. In the simple MuTI example from Section 5.1.1 all the instances of the *Patient profile exists* condition would be reported as *Irreversible*. This is because a profile can be created in the model with the *Create patient profile* use case, but there are no

means to delete it afterwards. The CTL formula for this reversibility category is as follows.

```
EF (Patientprofileexists$1$1$0$) & AG (Patientprofileexists$1$1$0$ →
AG (Patientprofileexists$1$1$0$))
```

Essentially the formula above states that once the `Patientprofileexists$1$1$0$` condition variable becomes true it remains constant for the rest of the model execution.

Finitely-reversible: The condition instance changes its truth-value more than once, but still a finite number of times. In the full MuTI example, we have the following condition declaration to keep track of appointment requests within the system.

Condition
Name: Appointment pending
Parameters: Nurse *of type* User ID, Doctor *of type* User ID, Patient *of type* Patient ID, Request time *of type* Time

When an appointment is requested by a nurse, an instance of the above condition becomes true. Once the doctor accepts or rejects the appointment request, that condition instance becomes false. Our model also captures the fact that the *Request time* is different for each appointment made by the same nurse. Once an instance of this condition becomes false, it can never become true again and hence it falls into the *Finitely-reversible* category.

The CTL formula for this category is as follows.

```
EF (Patientprofileexists$1$1$0$) & !AG EF (Patientprofileexists$1$1$0$)
```

Reversible: The condition changes its truth-value infinitely many times. Most conditions fall into this category. For example, all instances of the *User logged in* condition are *Reversible*. The CTL formula below tests reversibility of the *Patient profile exists* condition instance.

```
EF (Patientprofileexists$1$1$0$) & AG (Patientprofileexists$1$1$0$ →
EF (!Patientprofileexists$1$1$0$))
```

Verification for liveness of use cases and reversibility of conditions can be used to compile

a report that classifies each use case instance and condition instance according to the above-described categories. This report can provide the user with insight into the behaviour of the system described by the model, as well as warn him of potential errors in the model.

5.5 Model-Specific Verification of Susan Models

Verification against generic properties yields useful results, but because the generic properties cannot be used to test model-specific behaviour, this type of verification is limited. The fact that our user only works with the Susan modelling constructs and does not know anything about the NuSMV representation, excludes the possibility of letting the user to insert his own CTL properties into NuSMV programs. We could however allow the user to construct CTL properties using the Susan modelling elements and then interpret these appropriately for NuSMV. However, the CTL notation and semantics can get very cryptic especially when expressing lengthy formulae. For this reason, we adopt another approach that makes model-specific verification of Susan models more accessible for the user. We present the user with *property specification patterns* for the creation of custom properties. These patterns let one express simple properties for behavioural analysis without knowing the details concerning the underlying formalism, which is CTL in our case.

5.5.1 Property Specification Patterns

In software engineering, a pattern is a proven solution to a commonly encountered problem. In recent years, patterns have been widely used in software system design and programming [GHJV]. Property specification patterns are generalised descriptions of commonly-sought behaviours for verification of finite state systems. Each pattern has a name, description of intent, mappings to several specification formalisms such as CTL for example, common uses of the pattern and relationships to other patterns. Hence a user can easily instantiate a specification pattern, thus creating a verification property and effortlessly arrive at a corresponding formula in CTL.

Property specification patterns were first proposed by Dwyer *et al* in [DAC98] and further supported by empirical studies [DAC99]. The SAnToS Laboratory maintain an ongoing project for evolving these patterns, which is documented online [DAC04]. Dwyer *et al* developed a system of specification patterns, which comprises a set of property specification patterns that are organised into a hierarchy showing relationships between different

patterns. This hierarchy is shown below in Figure 14.

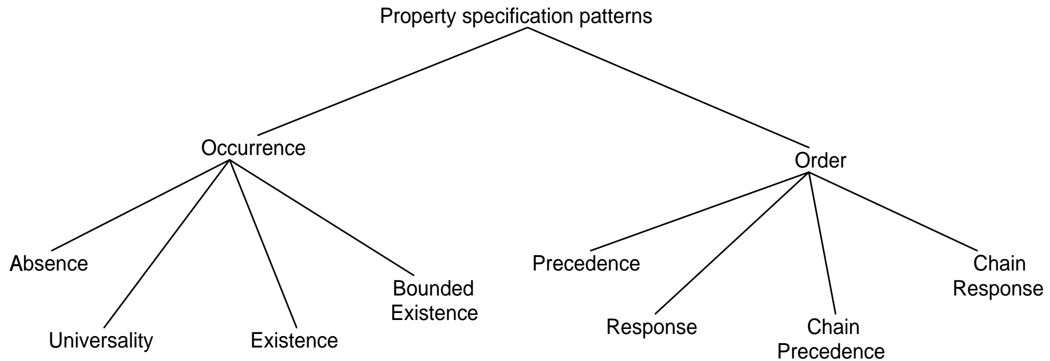


Figure 14: Dwyer's Property Specification Pattern Hierarchy

As can be seen in Figure 14, there are two main types of patterns: *occurrence* and *order*. Occurrence patterns can be used to verify existence or absence of system states where a certain property holds. While order patterns verify a certain ordering of system states or events.

Dwyer additionally defines *scopes* that are used in combination with patterns to specify on which part of the system execution the property must hold. Three examples of these scopes are *Global*, *Before* and *Between*. When the *Global* scope is used, the property must hold for the entire execution of the system model. *Before* means that the property applies to the execution up to a provided state or event. Finally, *Between* means that the property must hold on the execution between two states or events. Even though the concept of scopes allows one to construct more properties, it also unnecessarily complicates the patterns framework. This was shown by the survey results in [DAC99] where the overwhelming majority of sample properties were constructed using the *Global* scope.

For model-specific verification of Susan models, we do not support the use of scopes. All the patterns provided are implicitly instantiated with the *Global* scope and hence all the constructed properties apply to the entire model execution. Furthermore, we tailor the original pattern hierarchy slightly to suit our specific needs for Susan model verification. The augmented Susan pattern hierarchy is presented next.

5.5.2 Property Specification Patterns in Susan

In their surveys, Dwyer *et al* also discovered that the following patterns were used very rarely: *Bounded Existence*, *Response Chain* and *Precedence Chain*. For this reason, we did not include these patterns in our pattern hierarchy for Susan models. Additionally, we refined the *Existence* pattern into four more specific patterns as shown in Figure 15. In Susan, we used mappings to CTL as defined by Dwyer *et al* for all the patterns except the new *Existence* sub-patterns, for which we define our own mappings.

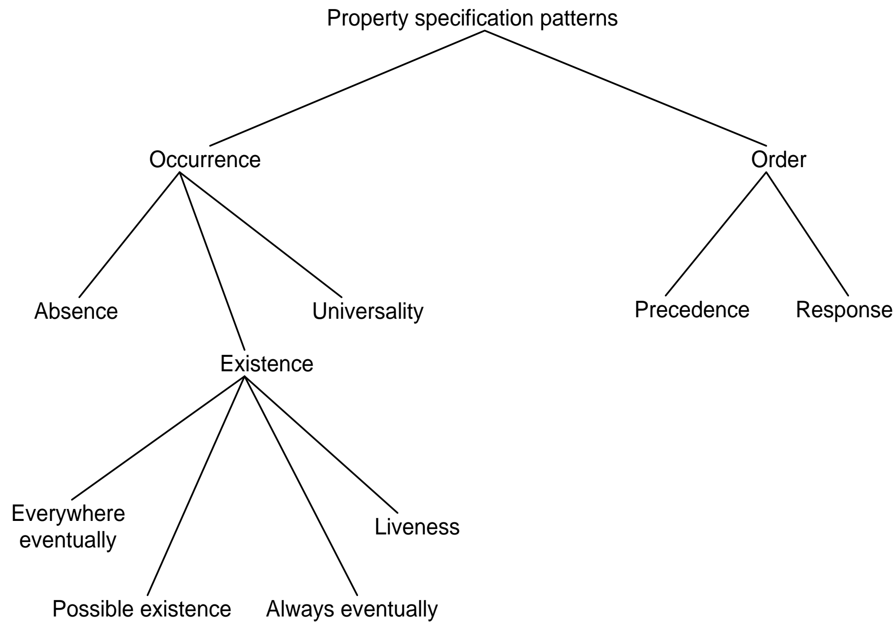


Figure 15: Susan Property Specification Pattern Hierarchy

Instantiation of patterns to construct behavioural properties is performed as follows. Each specification pattern contains one or more *pattern variables* that the user must substitute with valid values from the model being verified. Pattern variables are *predicates* or in other words functions that yield a boolean value. A pattern variable is parameterised and may be true for some arguments and false for others. In Susan, pattern variables can be constructed from: condition instances and the logical operators NOT (`!`), AND (`&`), OR (`|`) and implication ($\rightarrow$). Once the user chooses a pattern and fills in the pattern variables, the corresponding CTL formula can be generated.

Each of the patterns in our hierarchy is described next. We use condition instances from

the MuTI system for illustrative purposes. To make our discussion more understandable and concise, we use the following shorthands.

Con_A : Patient profile exists(Sister Nhlanga, Clinic A, Nicolas Brooks)
Con_B : Medical record exists (Sister Nhlanga, Clinic A, Nicolas Brooks, Doctor Bosman)
Con_C : User online (Dr Bosman, Hospital)
`Var_A` : `Patientprofileexists$2$0$1$`
`Var_B` : `Medicalrecordexists$2$0$1$0$`
`Var_C` : `Userloggedin$0$2$`

Absence (Never): *Safety properties* can be constructed using this pattern. An informal definition of a safety property is that "something bad will never happen" [Kin94]. Using this pattern we can check that a certain MuTI user can not create a medical record for a patient without having a profile for that patient first. To determine whether our model satisfies this, we can construct the following property: *Absence* of (!Con_A & Con_B). The following CTL formula is generated for this property.

```
AG !(!Var_A & Var_B)
```

Universality (Globally): This pattern can be used to express *invariants* for a model. An invariant is a property that must hold throughout the execution of the system. This pattern is closely related to the *Absence* pattern but while the *Absence* pattern is applied to negative properties, the *Universality* pattern applies to positive ones. We can use this pattern to express the same property that we used as an example for the *Absence* pattern above: *Universality* of (!(!Con_A & Con_B)) or *Universality* of (Con_B → Con_A). The corresponding CTL formulae are given below.

`AG (!(!Var_A & Var_B)) )` or `AG (Var_B → Var_A)`

Existence (Eventually): If we are interested in reachability of certain system states, then this pattern can be used to construct properties for model verification. When using *Existence* patterns it is important to remember that our NuSMV programs ensure fairness, which means that all use case instances are activated equally often. The four sub-categories

that we created for this pattern are explained below.

- **Everywhere eventually:** Something will always eventually happen, no matter what execution path is taken. Suppose that we want to make sure that irrespective of how the MuTI system is used by the users, *Dr Bosman* will eventually be online. The required property is *Everywhere eventually* (Con_C) and the CTL formula is as follows.

  `AF (Var_C)`

- **Possible existence:** It is possible for something to happen. In other words, the property may hold on some paths but not all the paths of execution. This pattern is closely related to the *Absence* pattern. Another alternative to check whether a MuTI user can create a medical record for a patient without having a profile for that patient first is by testing for *Possible existence* (!Con_A & Cond_B). The generated CTL formula is given below.

  `EF (!Var_A & Var_B)`

  It is clear that certain properties can be expressed using a number of different patterns. This may seem redundant, however in certain situations it is more natural to use one pattern and another pattern in a different situation.

- **Always eventually:** No matter where in the system execution we are, something will always eventually happen. This pattern is a stronger variation of the *Everywhere eventually* pattern. In our example for the *Everywhere eventually* pattern, the property tests that *Dr Bosman* can go online once for every possible scenario of system use. On the other hand, if we want to ensure that at any point in time *Dr Bosman* can go online, irrespective of what happens we need to use this pattern instead. This stronger property would be *Always eventually* (Con_C) and the corresponding CTL formula is as follows.

  `AG (AF (Var_C))`

- **Liveness:** Sometimes we want to ensure that at any time during the execution of the system, something will eventually become possible. This pattern is a stronger variation of the *Possible existence* pattern. For instance, the property *Liveness* (Con_C) states that at any time *Dr Bosman* will be able to go online at the *Hospital* in one of the

system usage scenarios. That usage scenario may be such that the network is available at the *Hospital* and *Dr Bosman* can log into the system. The generated CTL property is as follows.

```
AG (EF (Var_C))
```

Precedence: This pattern describes a dependency between two system states or events. It can be used to verify that one state or event always occurs before the other one. We can check that a patient profile is always created by a user before a medical record for that patient with Con_A *Precedes* Con_B. Here is the CTL formula.

```
!E [!Var_A U (Var_B & !Var_A)]
```

Response: Cause-effect relationships between system states or events can be expressed using this pattern. It is similar to the *Precedence* pattern but is used to verify that every cause must be followed by an effect rather than for every effect there must be a cause. In the *Precedence* pattern causes may occur without subsequent effects, while in the *Response* pattern effects may occur without causes. If we check the property Con_B *Responds to* Con_A, the verification of the property fails. This is because if a patient profile is created, it does not necessarily mean that a medical record will be created for the patient. The CTL formula is as follows.

```
AG (Var_A → AF (Var_B))
```

This concludes our discussion of the Susan method. In this chapter we described how Susan models can be constructed, translated to NuSMV and analysed using generic and model-specific verification. In the next chapter we present the SusanX tool, which implements the Susan method.

Chapter 6

The SusanX Tool

In this chapter we present the SusanX tool, which we developed to support the proposed Susan method for modelling and analysis of software requirements. An overview of all the features provided by the tool is first given, followed by a portrayal of how these features can be accessed by the user through the SusanX interface. Important architectural design and implementation concerns are also discussed.

6.1 Overview of Features Supported by SusanX

Computer-Aided Software Engineering (CASE) refers to the collection of tools built to assist developers during software engineering activities such as requirements specification, design and implementation [Som01]. These tools have become an integral part of any software development project, as they make the development process more efficient and reliable. Today, any practical software engineering method needs to be suitably supported with a CASE tool. Some of the common features provided by CASE tools include editors for the notation prescribed by the underlying method, analysis modules that can check models against rules defined by the method and report generators that help to produce system documentation [Som01]. SusanX is the CASE tool that we developed to accompany the proposed Susan method, and in this section we describe the features incorporated into this tool.

The main goal of SusanX is to allow the user to construct Susan requirements models and have them analysed as defined by the Susan method. SusanX offers five main features to the user, which we discuss next.

Model construction and navigation. The user is presented with a graphical editor

that allows easy construction and navigation of Susan models. The SusanX editor allows the user to view and work with one Susan model at a time. Use case diagrams for the models can be constructed by dragging-and-dropping modelling elements from the toolbar onto the main diagram canvas. A navigation tree containing all the elements in the opened model allows the user to traverse the model and view properties of each of the elements.

Static model check. Every constructed Susan model needs to pass a static check before it can be analysed dynamically. This check ensures that a model is structurally correct, as this is a prerequisite for a valid translation to NuSMV and obtaining accurate verification results. If errors are discovered in a model, an error report is generated for the user in a format that makes it easy to identify the source of the errors. The user is only allowed to proceed to dynamic verification of a model if the static check is completed successfully with no errors reported.

Batch generic verification. This is the most automated analysis option available in SusanX. The user initiates the verification process without providing any additional input into the SusanX tool and thereafter waits for results to be generated. Batch generic verification determines a liveness category for each use case instance, and a reversibility category for each condition instance in a model. At the end of the analysis, a report showing these categories is generated for the user. The only way in which the user can configure the verification is by restricting it to checking either only use case instances or only condition instances. For a sizeable model, this type of verification can take a long time as many properties need to be checked with NuSMV in order to compute a category for each use case and condition instance. In the case where the user is only interested in the results for selected use cases or conditions, the less expensive guided generic verification option is recommended.

Guided generic verification. Instead of verifying all the instances of use cases and conditions, this analysis option allows the user to check instances of an individual use case or condition against a specific category. Interpretation of generic verification results requires the user to have insight into the Susan model and know what the correct category should be for each of the use cases and conditions. In this analysis option the user is required to specify in which category he thinks the use case or condition instances belong, and hence the option is called "guided". A check against the given category is performed first, and only those instances that do not qualify are then checked against other categories. At the end of the analysis process, a category for each use case or condition instance is shown to

the user. In the case where the user can speculate well about the categories for use cases and conditions, the number of properties to be checked by NuSMV and hence the time to produce verification results is reduced.

Model-specific verification. In addition to generic verification, the SusanX user is also allowed to construct model-specific properties for analysis using specification patterns. When a property is found to be false, the counter-example generated by NuSMV is interpreted for the user as a sequence of use case activations by actors from the model.

6.2 Graphical User Interface of SusanX

In this section we present the graphical user interface of SusanX by going through a series of screenshots thus demonstrating how one would make use of the tool. The graphical editor window of SusanX with its parts labelled is shown in Figure 16.

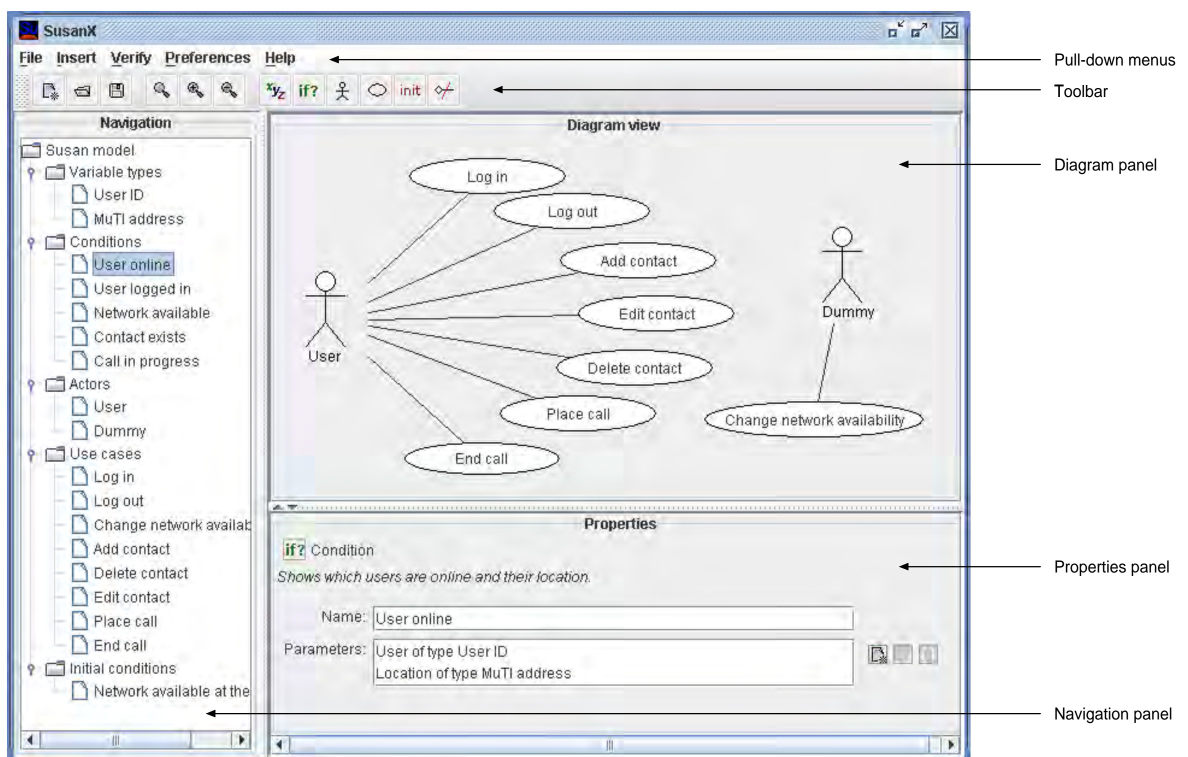


Figure 16: Graphical Editor Window of SusanX

SusanX allows the user to work with one Susan model at a time, and currently only one use case diagram can be associated with a model. In Figure 16 a model from the MuTI

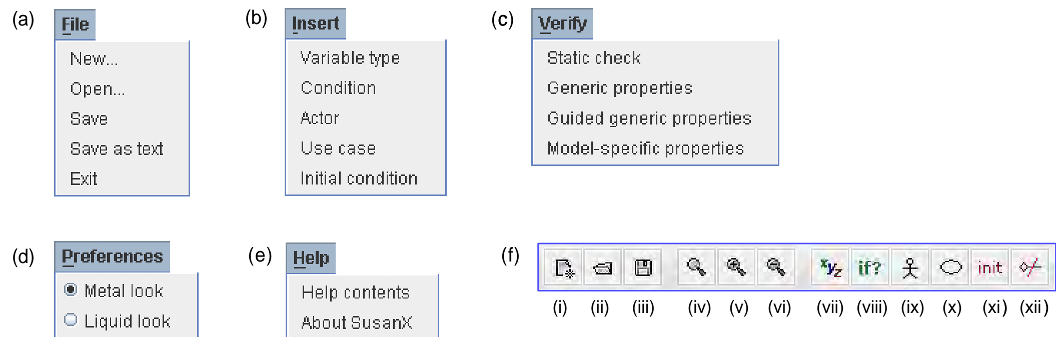


Figure 17: Pull-down Menus and Toolbar in SusanX

example is opened. As can be seen from the screenshot, the graphical editor of SusanX is divided into three main panels where each panel gives a different view of the model. The diagram panel displays the use case diagram for the opened Susan model, which shows all the actors and use cases and their associations in that model. The non-graphical modelling elements can be found in the navigation panel, which contains a navigation tree with all the elements making up the opened model. When the user selects an element in the navigation tree or the use case diagram, the properties for that particular element appear in the properties panel. In Figure 16, the *User online* condition is selected in the navigation panel and the details for this condition are displayed in the properties panel where they can also be edited. Manipulation of modelling elements with the editor is discussed further in Section 6.2.1.

The pull-down menus and toolbar appearing at the top of the editor window shown in Figure 16 are shown in detail in Figure 17.

Menu (a) allows the user to create, open, save models and also exit the application. The regular *Save* option stores the data about the current Susan model in a binary ".sus" file, which can later be opened in SusanX. Additionally, the user can select the *Save as text* option to write the model out to a text file. Such a text file contains all the information about the model, including all its elements and their properties. It can assist the user in creation of reports and similar tasks, but cannot be used to re-open a Susan model with SusanX. Buttons (i)-(iii) on the toolbar in (f) provide quick access to the most commonly used functionality in menu (a).

New elements can be added to a model in SusanX either through menu (b) or toolbar buttons (vii)-(xi). These buttons on the toolbar appear in the same order as the menu

items, in other words button (vii) corresponds to a new variable type, button (viii) to a new condition and so on. When a new element is added to the model, the navigation tree is updated and the properties panel is filled appropriately for this element to allow the user to populate its properties. If a new actor or use case is added then the diagram panel is updated as well. Once actors and use cases are added to the diagram view canvas, associations between them can be created in a drag-and-drop manner. Toolbar button (xii) allows the user to change the association mode in SusanX. By default the association mode is "off" as shown in Figure 17, which means that the user first has to change it before he can create associations by dragging-and-dropping on the diagram canvas. Toolbar buttons (iv)-(vi) let the user zoom in and out of the use case diagram.

All the verification options are available through menu (c). Each one of the menu items brings up a separate verification window. Some preference and help options can be accessed through menus (d) and (e) respectively.

6.2.1 Building a Model in SusanX

The SusanX interface is built to ease the process of creating Susan models for the user. We recommend that the user adds elements and defines their properties in a specific order described next to facilitate efficient model construction.

As a first step of model construction, the user should draw the use case diagram for the model in the diagram panel. Next, variable types should be added and valid values defined for each. Then the user should insert conditions into the model and specify their parameters. The properties panel for the *Call in progress* condition is shown in Figure 18 (a). As can be seen from the panel screenshot, this condition has four parameters. The user can add, edit and remove parameters using the three buttons situated to the right of the parameters list in the properties panel. In the screenshot the first condition parameter is selected in the list. If the user presses the edit button then the window shown in Figure 18 (b) and (c) is displayed. One can see that the drop-down list in this window allows the user to set the type for the parameter only to one of the variable types that have already been added to the model. This prevents the user from making erroneous type assignments, but requires variable types to be inserted into the model in the very beginning.

Once conditions are defined, the user should proceed to define attributes for each of the actors in the use case diagram. This should be followed by the user filling in the use case properties. The properties panel for the *Log in* use case is shown in Figure 18 (d). In

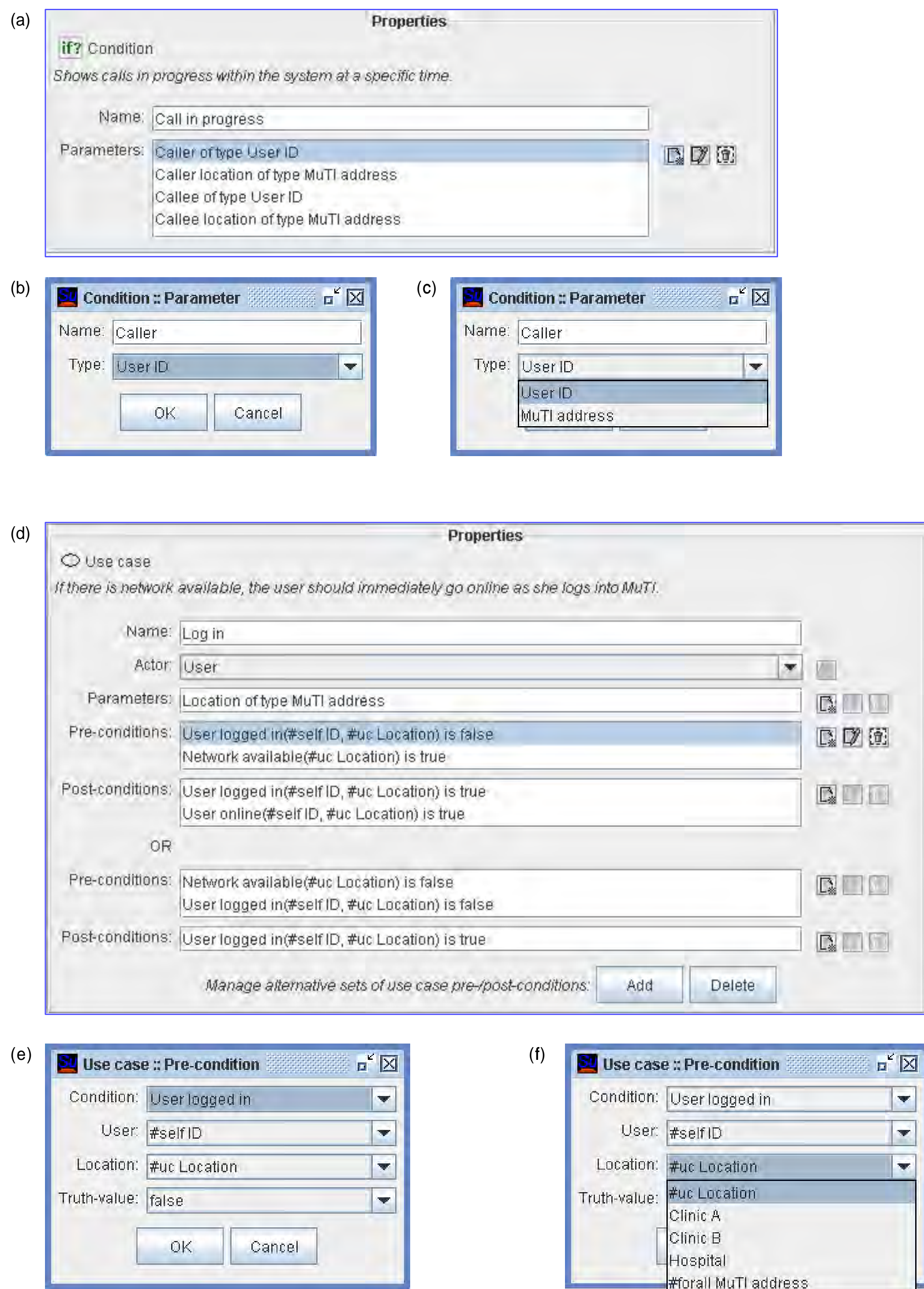


Figure 18: Properties of Condition and Use Case Elements in SusanX

addition to making associations between use cases and actors with dragging-and-dropping on the diagram view canvas, the user can also set the associated actor for each use case in the properties panel by using the drop-down list labeled "Actor:". Use case parameters are created in the same way as condition parameters, which was explained earlier. Screenshots (e) and (f) in Figure 18 show how a pre- or post-condition for a use case can be created and edited. Firstly, the user selects a condition type from the drop-down list labeled "Condition:". After this selection, the window updates to display the parameters corresponding to the chosen condition. Next, the user needs to assign each of these parameters to one of the valid options again available via drop-down lists. For each parameter, SusanX finds the possible assignment options through type-matching. In other words, it gets literal values from the variable type definition and also finds those actor attributes and use case parameters that are of the same type as the parameter. All these and the *forall* option are combined and used to populate the parameter drop-down list as shown in Figure 18 (f). Once each of the parameters is assigned, the user also needs to specify the truth-value for the pre- or post-condition created.

Alternative sets of pre- and post-conditions can be added and deleted with the buttons at the bottom of the properties panel in Figure 18 (d).

The final step in the construction of a model in SusanX is insertion of initial conditions. Properties of initial conditions are defined in a similar way to all other modelling elements.

The recommended procedure for building models in SusanX described above assumes that the user knows in advance all the elements for the model and their properties. Of course, this is not usually the case and the model construction process will require some iterations. Consequently, our recommendations define the general outline for the actual process. Once the model is constructed, the user needs to run a static check on it before he can make use of the verification options offered in SusanX.

6.2.2 Static Checking on a Susan Model

The screenshot in Figure 19 illustrates how structural errors are reported to the user after a static check is performed in SusanX.

The generated error report informs the user of the number of errors that were found in the model and describes each of the errors discovered. The first line of each error message conveys the type of the error and is followed by the details that allow the user to locate the source of the error easily. For example, error 3 in Figure 19 says that the *Patient* parameter

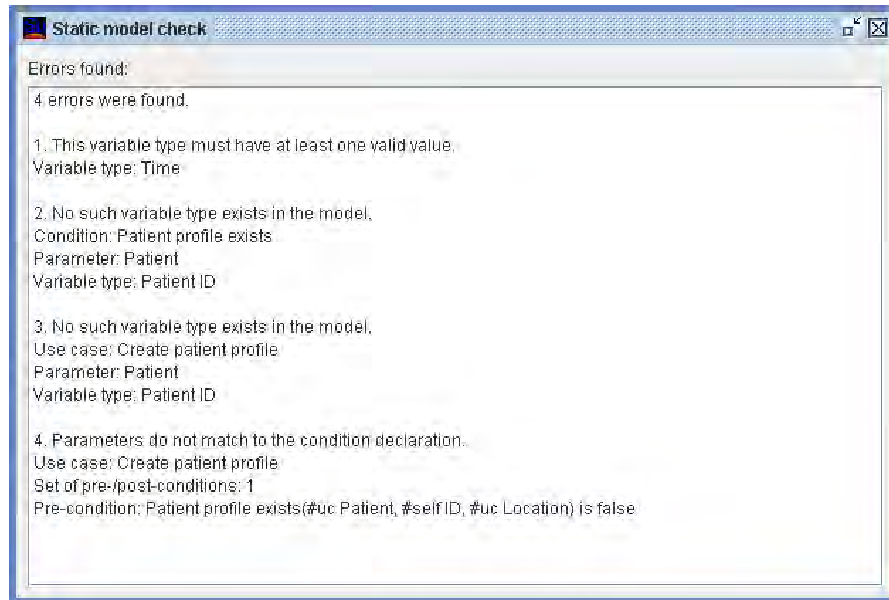


Figure 19: Static Model Check Window in SusanX

of the *Create patient profile* use case is assigned to an invalid variable type called *Patient ID*. Such an error typically arises when after the definition of use case properties, one of the used variable types is removed from the model - in this case, the *Patient ID* variable type was removed. In order to fix this error, the user must either re-assign the *Patient* parameter to another type that is valid or he must create a variable type called *Patient ID* once again.

There are ten different error types that can be determined by the SusanX static checker:

1. *Empty variable type:* A variable type with no values is defined within the model, which is not permitted.

2. *Invalid variable type:* A condition parameter, use case parameter or an actor attribute is assigned to a variable type that does not exist within the model.

3. *Invalid condition:* A pre- or post-condition of a use case is assigned to a condition type that is not defined within the model.

4. *Condition parameter mismatch:* Parameters of a use case pre- or post-condition do not match those of the corresponding condition type declaration.

5. *Invalid actor attribute:* A use case pre- or post-condition parameter is assigned to a

non-existent actor attribute.

6. *Invalid use case parameter:* A use case pre- or post-condition parameter is assigned to a non-existent use case parameter.

7. *Invalid value:* A use case pre- or post-condition parameter is assigned to a non-existent literal.

8. *No use cases or conditions:* The model does not contain any conditions or use cases, which does not make it suitable for verification.

9. *Invalid actor for use case:* An association between a use case and a non-existent actor is found within the model.

10. *No pre-conditions in the set:* A use case with multiple pre- and post-conditions sets is found, but one of the sets does not contain any pre-conditions.

The user has to adjust the Susan model until no errors are reported by the static checker. Only in this case the translation to NuSMV can be correctly executed and verification can be performed.

6.2.3 Running Batch Generic Verification

Batch generic verification checks each use case and condition instance in a Susan model against liveness and reversibility categories respectively. When the user chooses this verification option, no additional input is required from the user in order to begin the verification. Generated results are displayed as shown in Figure 20.

As can be seen from the screenshot in Figure 20, computed categories are shown next to use case and condition names and not their individual instances. This compressed view presents the user with an overview of the results that is easily comprehensible. Full results for a use case or condition can be viewed by choosing it in the drop-down list at the bottom of the window and pressing the "expand" button. This displays all the instances of the selected condition or use case with a category assigned to each one. Commonly, all instances of a condition or use case fall into the same category and the expansion is not necessary. When this is not the case and instances span more than one category, all these categories are displayed in a list next to the condition or use case name in the overview report. In Figure 20 some of the *Delete contact* use case instances are *Live* while others are

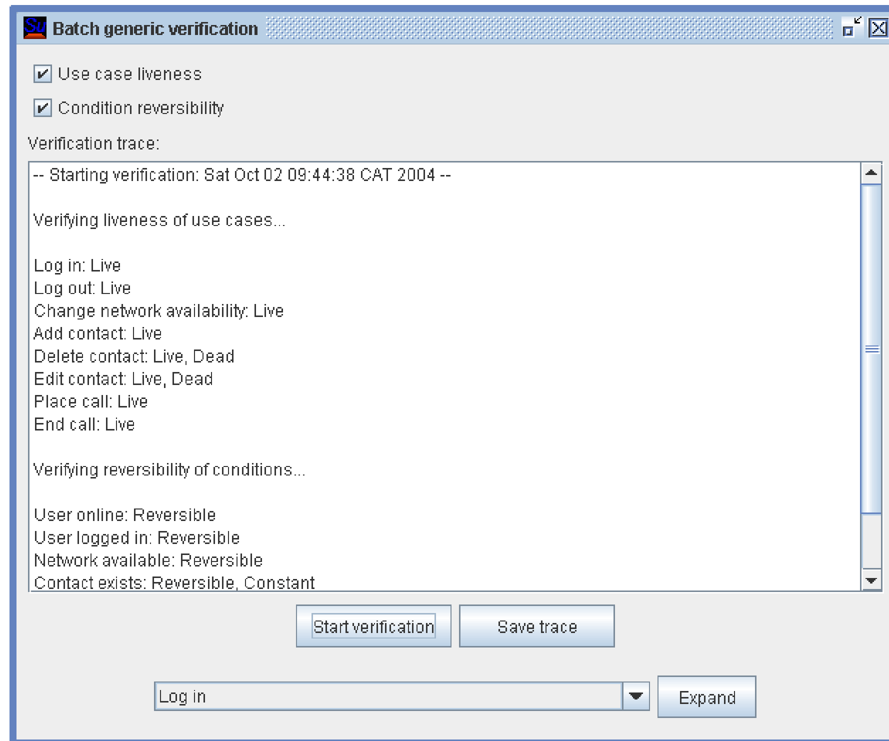


Figure 20: Batch Generic Verification Window in SusanX

Dead. The results for this use case can be expanded to see precisely which instances fall into which liveness category.

6.2.4 Using Guided Generic Verification

As batch generic verification can take time to complete for a model of a significant size, the guided verification option can be very useful in obtaining the same results more quickly. The window that allows the user to initiate this type of verification and where the results are displayed is shown in Figure 21.

One verification run determines categories for all the instances of one particular condition or use case. In addition to selecting that condition or use case, the user is also required to speculate as to which category the instances of the condition or use case are likely to belong. If the speculation is correct for the majority of instances then the time taken by verification is reduced. Details about the implementation of this type of verification are given in Section 6.3.2.

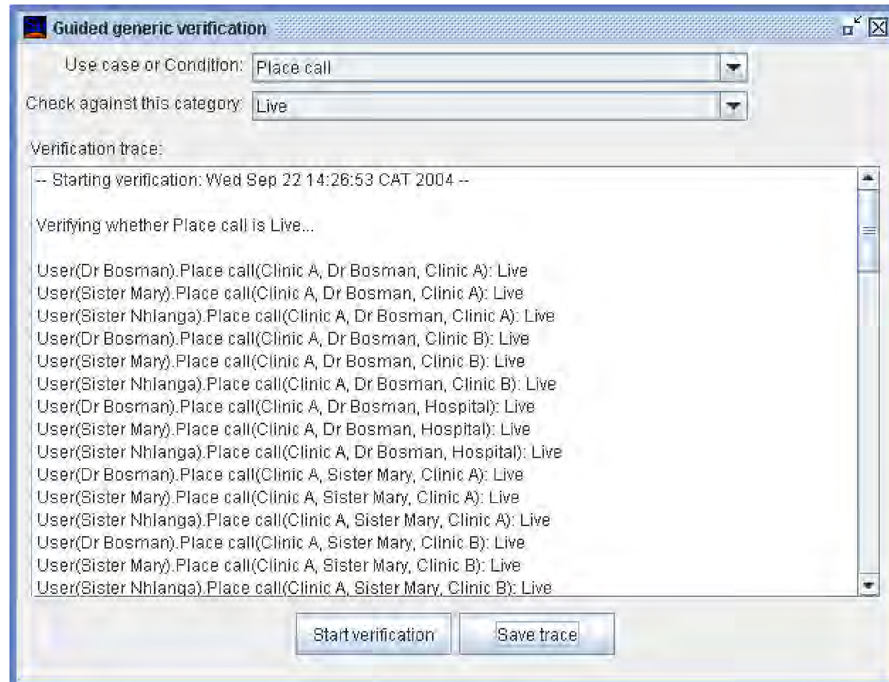


Figure 21: Guided Generic Verification Window in SusanX

The generated results displayed to the user show a category for each instance of the chosen condition or use case. In Figure 21 instances of the *Place call* use case are shown. For each instance, literal values for actor attributes and use case parameters are shown in brackets next to the actor and use case names respectively. Where an instance does not fall into the category selected by the user, the correct category would be computed by SusanX and shown in the results.

6.2.5 Verifying Model-specific Properties

Model-specific verification in SusanX allows the user to construct his own properties to be checked against the model. This verification option provides the most power to the user, but at the same time requires the most user input in addition to the model constructed. Figure 22 shows two examples of model-specific properties constructed and verified in SusanX.

Firstly the user needs to select one of the patterns from the pattern list labeled in Figure 22 (a). The name of each pattern in the drop-down list is followed by a short explanation where “A” and “B” represent pattern variables. These explanations make

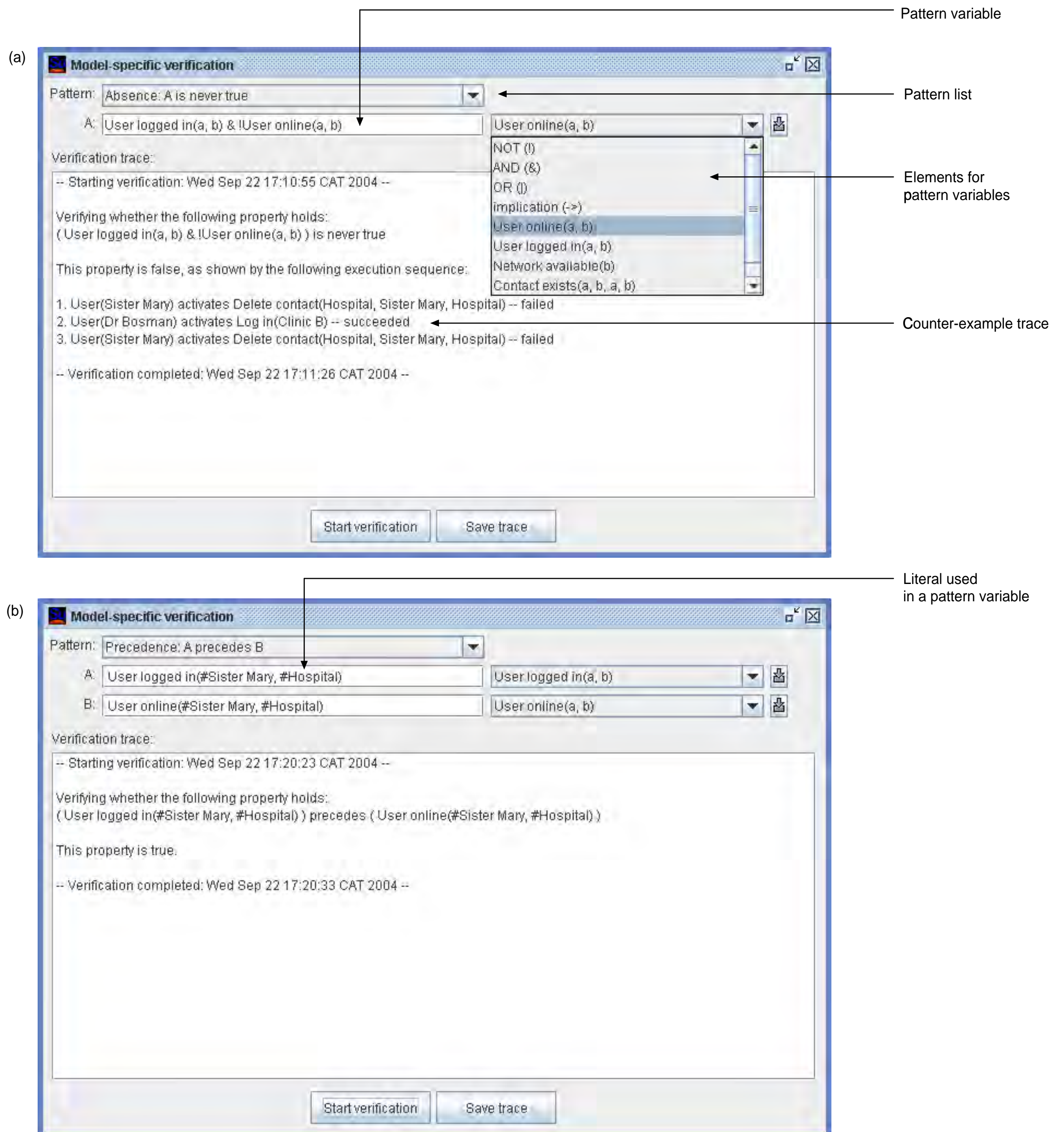


Figure 22: Model-specific Verification Window in SusanX

it easier for the user to construct properties using the available patterns correctly. In Figure 22 (a) “A is never true” is the explanation for the *Absence* pattern, and in (b) “A precedes B” explains the pattern of *Precedence*.

After choosing a pattern, the user needs to construct the pattern variable or variables. The user can type these in or insert valid pattern variable elements using the drop-down list and the “insert” button located to the right of the pattern variable text area. In the drop-down list, parameters of different conditions that belong to the same variable type are labeled with the same letter. In the example shown in screenshot (a) SusanX used “a” to label all condition parameters of type *User ID* and “b” for those of type *MuTI address*. These letters must be changed manually by the user in the pattern variable text area if no matching between parameters is required. Our example in (a) illustrates a property that checks that a particular MuTI user can never be logged in and offline at the same time. In this scenario, matching of parameters is required as we want the *User logged in* and *User online* conditions to refer to the same user and location. The example in (b) checks that before *Sister Mary* can go online at the *Hospital*, she needs to log into the MuTI system. As can be seen from the screenshot, literals are used in pattern variables to express this property. Each literal value used in a pattern variable must be prefixed by a hash sign (#).

In example (a), verification determined the property to be false and the counter-example trace is displayed to the user. Each step in the trace is a use case activation showing also its success or failure. Literals for actor attributes and use case parameters appear in brackets next to actor and use case names respectively. The counter-example in this case consists of one successful use case activation, where *Dr Bosman* logs into MuTI at *Clinic B*. The property verified in example (b) was determined to be true, so no counter-example trace is necessary.

The next section contains important details about the design and implementation of the SusanX tool.

6.3 Implementation Details

The entire implementation of SusanX was done in Java (j2sdk1.5.0), with Swing being used for the Graphical User Interface (GUI) programming. An open source Java Graph visualisation library called JGraph[1] (http://www.jgraph.com) assisted us in the creation of

[1] An open source version of JGraph that was used in this project is available under the LGPL. No alterations were made to the JGraph libraries in this project.

the diagram panel in the SusanX editor.

Version 2.1.2 of the NuSMV[2] model checker was used as the verification engine in SusanX. We chose WindowsXP for the testing of the SusanX tool, as it is the operating system most likely to be used in a generic software development environment. The Windows version of NuSMV 2.1.2 that we incorporated into SusanX works with Cygwin, a Linux emulator for Windows that provides substantial Linux API functionality (http://www.cygwin.com/). SusanX initiates a Cygwin process from inside the Java code, which subsequently passes commands to the NuSMV model checker. As a result, the current SusanX tool can only be used on a Windows operating system. Nevertheless, the implementation can easily be extended to work with other versions of NuSMV, making SusanX platform-independent.

An improved NuSMV version 2.2.0 was released on 10 June 2004, at which time this project was nearing its completion. A number of tests were done to determine whether this later release of the model checker could improve verification performance of SusanX. No substantial enhancement was noted with our particular use of NuSMV.

In the remainder of this section we describe the architecture and structure of the SusanX implementation. Special emphasis is placed on the implementation of the different analysis options in SusanX, where the interaction between SusanX and NuSMV is explained in detail.

6.3.1 Architectural Overview of SusanX

The SusanX implementation consists of several distinct parts or *components*, each of which provides a well-defined function of the tool. A Unified Modelling Language (UML) definition of a component is "a physical and replaceable part of a system that conforms to and provides the realisation of a set of interfaces" [BRJ99]. Systems that are developed according to a component-based design are generally more robust and flexible. Such systems can be evolved without difficulty, especially in cases where incorporation of new technologies into the implementation is required while the user interface needs to remain the same. All the components used to assemble SusanX and their relationships are shown with a UML component diagram in Figure 23.

Figure 23 shows three different types of components: *executables*, *libraries* and *files*. Executable and library components are self-explanatory. File components represent files

[2]All versions of NuSMV are provided under the LGPL v2.1, which is an open source license that allows free academic and commercial usage of NuSMV.

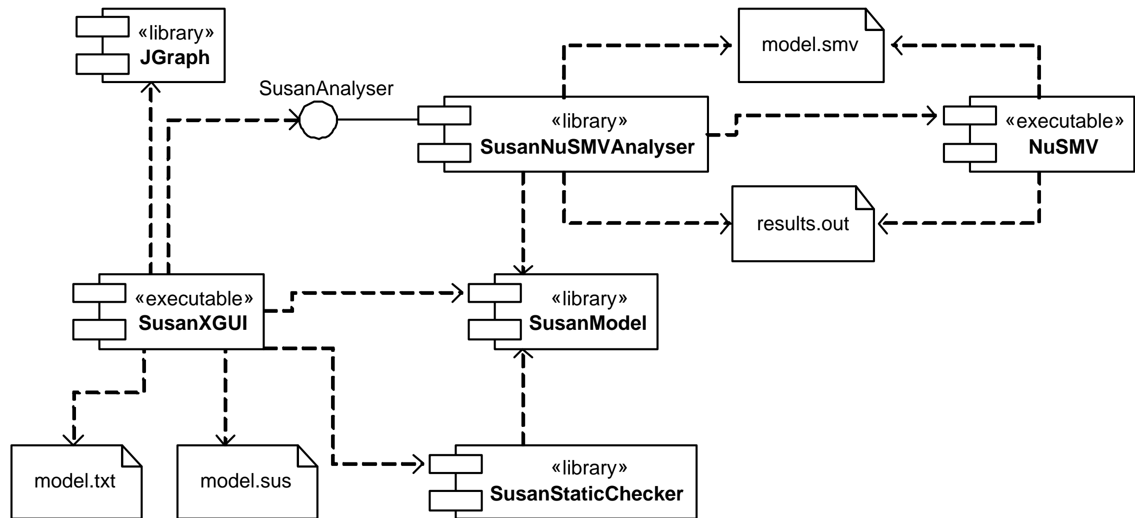


Figure 23: Component View of SusanX

either generated or used by the SusanX tool. Even though these files are not literally a part of the SusanX system, they are indispensable to the functionality that it provides.

The *SusanXGUI* component contains the implementation of the graphical editor interface provided by SusanX. All the Susan model related data displayed to the user in the editor is stored internally using data structures from the *SusanModel* library. The classes contained in this library and their relationships conform to the Susan metamodel described in Chapter 5. The *JGraph* library is used for displaying use case diagrams in the diagram panel of the editor and allowing the user to manipulate these diagrams. Once a Susan model is created, it can be saved with the layout of its use case diagram and opened again at a later stage. When a model is saved, the associated data structures and object representation of the diagram layout are serialised to a "model.sus" file. Moreover, a Susan model can be saved as a "model.txt" file readable by a human user, but not suitable for re-opening a model.

The interface provided by *SusanXGUI* allows the user to access and parameterise the analysis options, but the analysis functionality itself is not implemented in this component. *SusanStaticChecker* runs static checks on a Susan model and reports on the identified errors, which are then interpreted for the user by *SusanXGUI*. Dynamic analysis is implemented in the *SusanNuSMVAnalyser* component, which uses the *NuSMV* executable

as the verification engine. Provided with a Susan model by *SusanXGUI*, *SusanNuSMV-Analyser* generates a NuSMV representation of the model and saves it in a data file called "model.smv". For each of the analysis options, *SusanNuSMVAnalyser* also inserts appropriate verification properties into this file and passes it to *NuSMV* to be checked. Once verification is completed, *NuSMV* writes out the results to a "results.out" file, which is subsequently parsed by *SusanNuSMVAnalyser*. These results are also interpreted in terms of Susan model elements by *SusanNuSMVAnalayser*, after which they are finally shown to the user by *SusanXGUI*.

As can be seen from Figure 23, the *SusanNuSMVAnalyser* realises an interface called *SusanAnalyser*. It means that this component can be replaced by another, provided that the new component also realises the defined interface. For instance, further experimentation with the Susan method using SPIN [Hol97] as a verification engine could be easily done using SusanX with a new dynamic analysis component.

Our entire implementation of the SusanX tool consists of four packages, shown in Figure 24 with a UML package diagram. Each of these packages corresponds to a component described earlier.

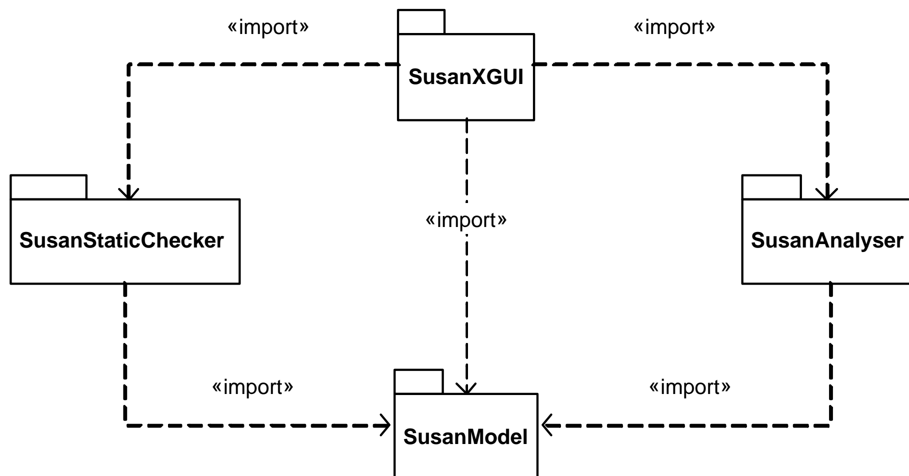


Figure 24: Package View of SusanX

An incremental development process was followed for SusanX, where each increment integrated a new component into the tool. The development was structured "bottom-up", with the low-level components built before the graphical interface. *SusanModel* and *SusanNuSMVAnalyser* were implemented in the first and second increments respectively.

This particular approach was taken as it was important to establish in the beginning whether the NuSMV model checker was suitable for our verification purposes. The GUI of the SusanX tool was developed in the third increment, after implementation of the NuSMV analysis functionality was completed and sufficiently tested for correctness. As the last addition to the tool, the *SusanStaticChecker* component was added in increment four.

Further details on the contents of the *SusanModel*, *SusanXGUI* and *SusanStaticChecker* packages are not essential for the purpose of this dissertation. On the other hand, the design and implementation aspects of the *SusanAnalyser* package are worthy of note and are discussed next.

6.3.2 SusanAnalyser Package

The *SusanAnalyser* package consists of 15 classes, of which the most important 6 are shown in the class diagram in Figure 25. *UseCase* and *Condition* classes belong to the *SusanModel* package, as indicated by their class names in the diagram.

The "user" of the *SusanAnalyser* package can invoke all the analysis functionality by calling the methods of the *VerificationControl* class. In SusanX, the *SusanXGUIControl* class from the *SusanXGUI* package creates an instance of the *VerificationControl* class and makes calls to the provided methods. In turn, *VerificationControl* uses the *CodeGenerator* class to create the "model.smv" file for the analysed Susan model and the *VerificationResultsParser* class to parse the "results.out" file produced by NuSMV.

UseCaseWithCategory and *ConditionWithCategory* are specialisations of the *UseCase* and *Condition* classes respectively. These classes contain an indication of the use case liveness or condition reversibility category to which the underlying element belongs. After any type of generic verification a *LinkedList* of these objects is returned to the *SusanXGUIControl*, consider for example the method *verifyUCLivenessBatch() : LinkedList* in the *VerificationControl* class.

UseCaseActivated also inherits from the *UseCase* class, and is used to represent a use case activation rather than just a use case element in a model. In addition to the use case details, it contains an indication of whether the use case activation was successful or not. A *LinkedList* of *UseCaseActivated* objects is created to represent a counter-example during model-specific verification and returned as in the method *processModelSpecificResults() : LinkedList* in the *VerificationResultsParser* class.

Instead of explaining the individual attributes and methods that appear in Figure 25, we

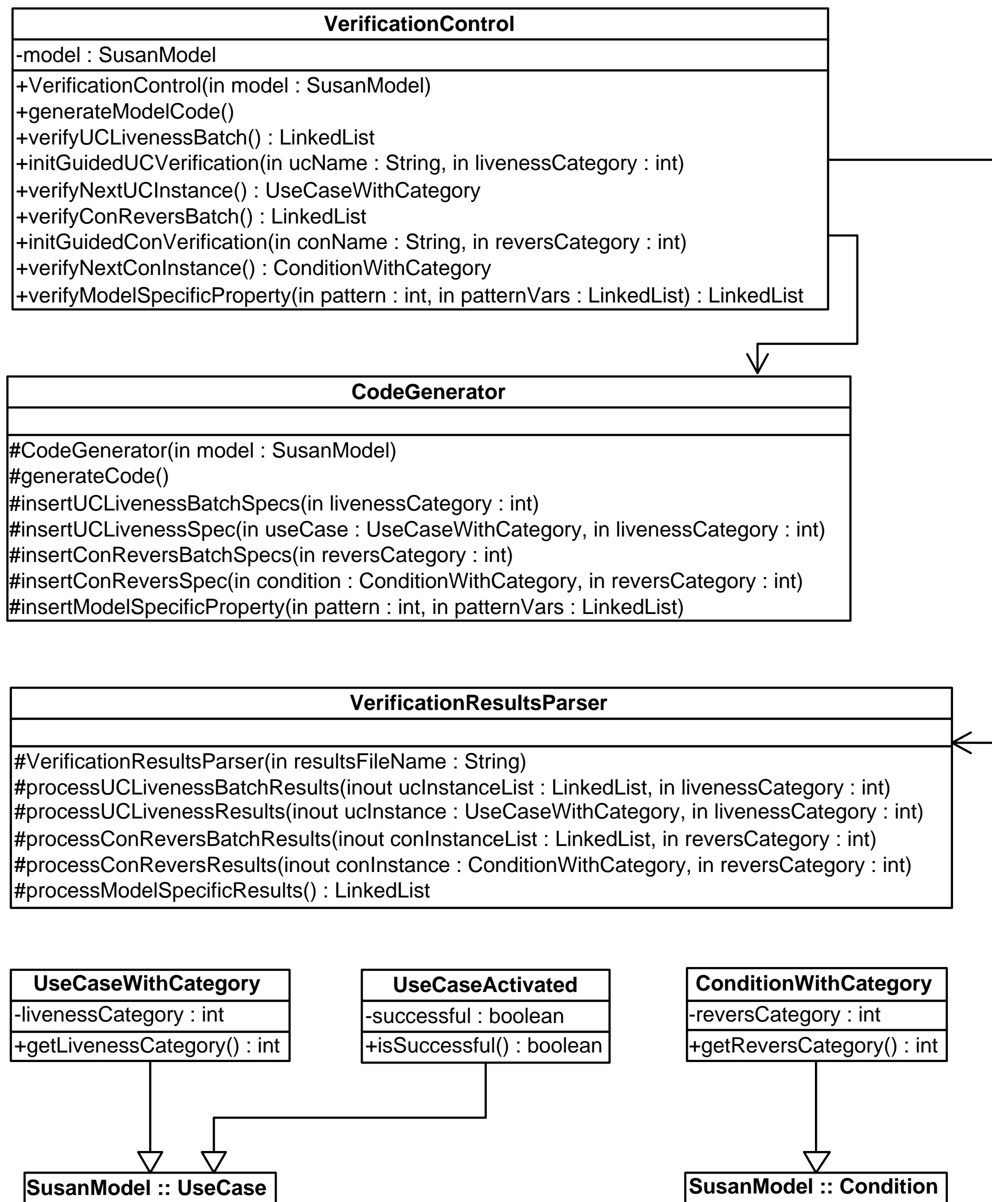


Figure 25: Classes in SusanAnalyser Package

illustrate the typical usage scenario of the classes in the *SusanAnalyser* package. Figure 26 uses a UML sequence diagram to show the sequence of actions that take place during batch generic verification of use case liveness.

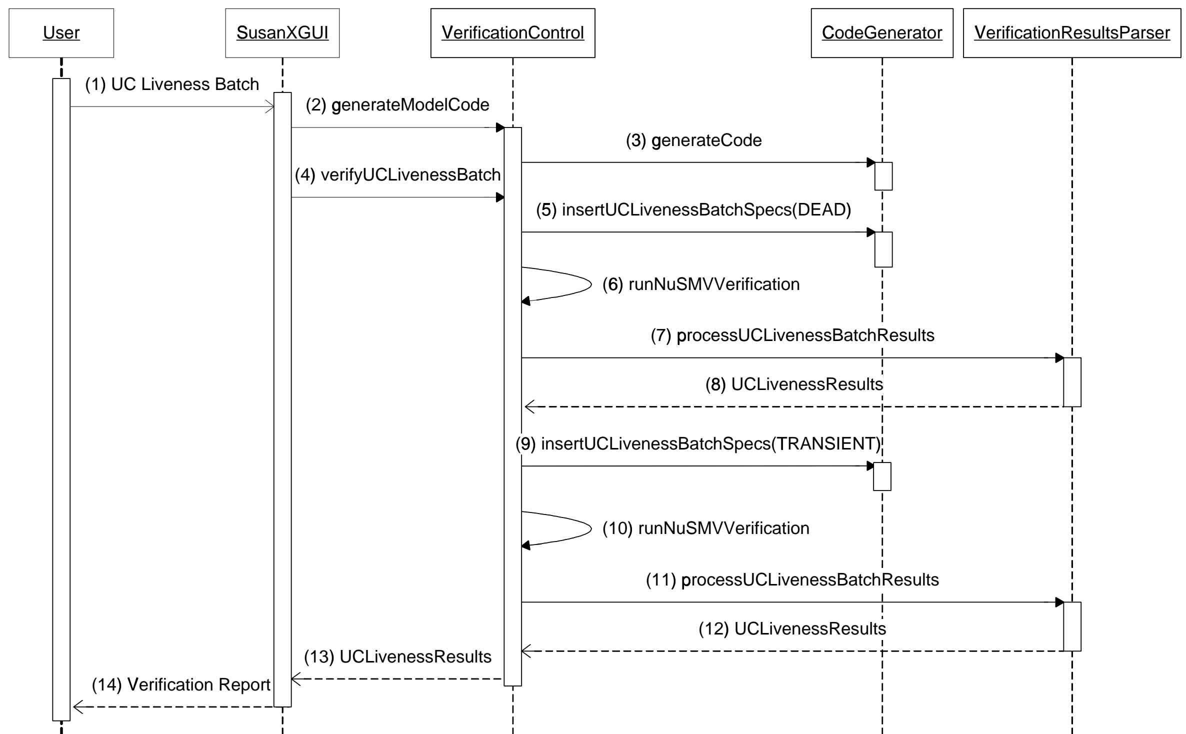


Figure 26: Sequence of Actions During Batch Use Case Liveness Verification

In Figure 26 the user initiates the verification option in (1), after which NuSMV code is generated for the current Susan model in (2) and (3). Next the *verifyUCLiveness-Batch* method in the *VerificationControl* class is called in (4). Inside this method, firstly in (5) *CodeGenerator* is used to insert Computational Tree Logic (CTL) properties into the "model.smv" file that check each use case instance against the *Dead* category. Then verification is run with the NuSMV model checker in (6), which produces the "results.out" file.

In (7) *VerificationResultsParser* is used to parse the verification results and assign the *Dead* category to the use case instances for which the CTL properties were reported to hold in the model. Using the results in (8), *VerificationControl* finds those use case instances that still need to be assigned to a liveness category and runs verification to check whether they are *Transient* in (9), (10) and (11). The remaining unassigned use case instances in (12) get assigned to the *Live* category, after which a list of *UseCaseWithCategory* objects is passed back to *SusanXGUI* in (13). This list is used to generate a verification results report shown to the SusanX user in (14), the final step in this scenario. Batch generic verification of condition reversibility is performed in an analogous manner in SusanX.

During guided generic verification, NuSMV is used to verify only one property at a time. For a use case, *SusanXGUI* first instructs *VerificationControl* to prepare the guided verification for the chosen use case and liveness property with *initGuidedUCVerification* and then calls *verifyNextUCInstance* repeatedly until all the instances of the use cases have been checked. A similar process is followed during guided generic verification of a condition.

For model-specific verification, the pattern information is passed to the *VerificationControl* class by *SusanXGUI*. That information is used to generate the necessary CTL properties in the *insertModelSpecificProperty* method of *VerificationResultsParser*. If the property is determined to be false during verification, a counter-example represented by a list of *UseCaseActivated* objects is passed back to *SusanXGUI*.

In this chapter we presented the SusanX tool and explained its design and implementation details. In the next chapter we describe the case study that we performed in order to assess the applicability of Susan and the SusanX tool in practice.

Chapter 7

Case Study: A Cash Management System

In this chapter we describe the case study we carried out in order to assess the suitability of the Susan method for addressing real problems in a software development project. Firstly, an overview of the case study system is provided and the goals of the case study are identified. Subsequently, our approach to the case study and the process followed during the investigation are presented. Each stage of this process is explained in detail, including the results obtained. The chapter is concluded by a synopsis and an evaluation of all the major results of the case study.

7.1 Case Study Overview

In order to evaluate the suitability of our proposed method for modelling and analysing requirements of real-world software development projects, we conducted a case study of a Cash Management System (CMS). This case study was made possible through cooperation with SoftCo[1], a South African IT company. SoftCo were contracted to develop the CMS for an international business group and at the time of the case study a part of this project was still in progress. More specifically, the project was divided into two phases. Phase 1 was completed in July 2003 and the developed system was successfully deployed. Development for Phase 2 of the project was put on hold until June 2004 when requirements gathering for this next part of the system began.

SoftCo is a well-established developer and distributor of software business solutions

[1]The real name of the company is Software Futures. However, it is not essential to our discussion in this chapter and hence we use the fictional name, SoftCo.

in South Africa. It has a proven track record of successful projects for medium-sized businesses, large corporations and government organisations. The software engineering approach adopted by SoftCo is based on the best practices, including those advocated by the Rational Unified Process (RUP) [Kru01] and Model Driven Architecture (MDA) [mda02]. Specification of requirements is recognised as an integral part of any development project by this company and is done using traditional use case modelling. This choice of SoftCo allowed us to observe how effective traditional use case modelling is in practice and explore how Susan can be used to bring about additional benefit. Furthermore by analysing a "business system" such as the CMS, we were testing our approach in the sphere of use case modelling rather than formal methods. This coincides with the objective of our work, which is to improve the current state of use case modelling and not to extend the use of this approach to specifying requirements for systems unsuited to this representation, such as for example safety-critical systems.

The main goal of the CMS is to support management of receipts, as well as coordinate the flow of information between various other computer systems employed by the client company. An overview of the system architecture is presented in Figure 27.

Employees of the client company should be able to make use of the CMS services through a web-based interface. This is represented by the two workstations connected to the CMS server in Figure 27. Examples of the services that the system should provide are receipt search, receipt printing and report generation. The CMS database should store all the important receipt information, as well as other data such as user details. The CMS should act as the front-end to the company's accounting system that keeps record of all the monies received from various debtors. The receipt transactions posted to the accounting system should also be reflected in the operations system databases. As shown in the diagram, the scope of the CMS architecture includes the databases of the accounting and operational systems since the CMS provides these with receipt data. However, how the receipts data is handled by the accounting and operational systems is out of the CMS scope.

The development of the system was divided up as follows:

Phase 1: General Receipts and Bulk Receipts

- **Increment 1:** Administrative System
- **Increment 2:** Manual Receipts
- **Increment 3:** Bulk Receipts

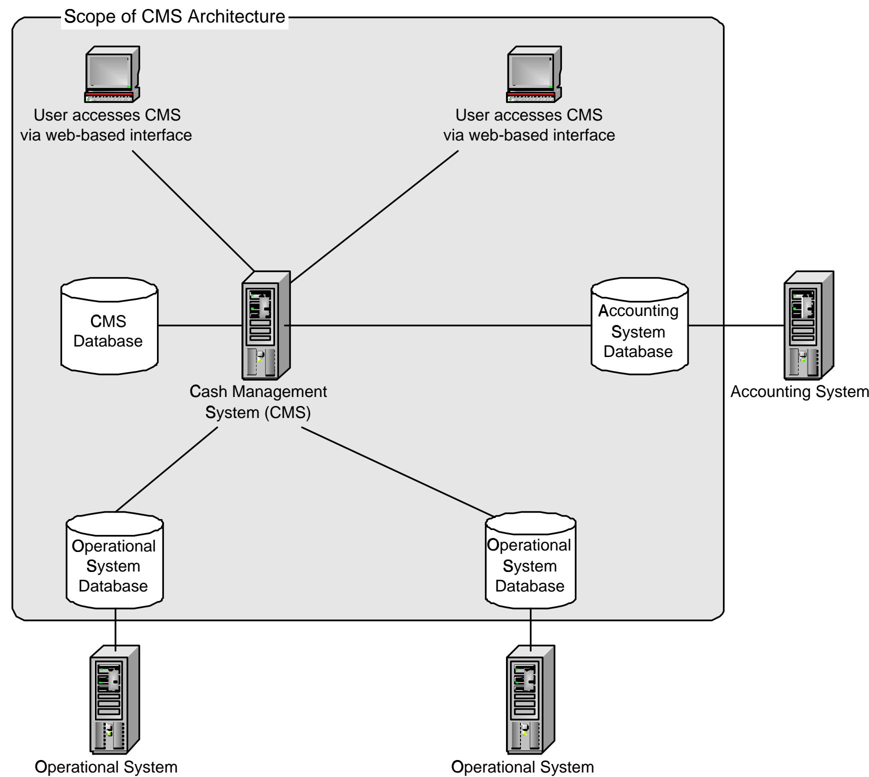


Figure 27: CMS Architectural Overview

Phase 2: Requisition System

For the case study we worked with Increments 1 and 2 from Phase 1, which we deemed sufficient for our purpose. We acquired the requirements models and documents for the CMS project drawn up by the SoftCo team, and based our investigation on these. These models and documents composed the final requirements specification for Phase 1 of the CMS. However, they were captured informally and were not analysed by any rigorous means, hence we aimed to show that Susan can be used to improve the state of the CMS requirements specification. More specifically, our objective was to assess the following.

1. *Suitability of Susan notation.* Firstly, we wanted to determine how appropriate the Susan notation is for expressing requirements for a software system such as the CMS.

2. *Benefits of formalising use case models.* Additionally, we were to establish whether formalising software requirements with Susan is advantageous for this type of system.

For example, whether it assists in discovering errors or makes requirements more intelligible.

3. *Effectiveness of SusanX analysis options.* Furthermore, it was important to evaluate how effective the analysis options provided by the SusanX tool are in making requirements more correct, complete and consistent.

4. *Usability and performance of SusanX.* Lastly, we wanted to assess the usability and performance of SusanX when modelling and analysing realistic requirements models.

7.2 Modelling and Analysing CMS Requirements with Susan

The process that we followed during the case study consisted of four stages, where at the end of the last stage, we produced an improved requirements model of the CMS requirements for Increments 1 and 2. The details of this process are depicted as a flowchart in Figure 28.

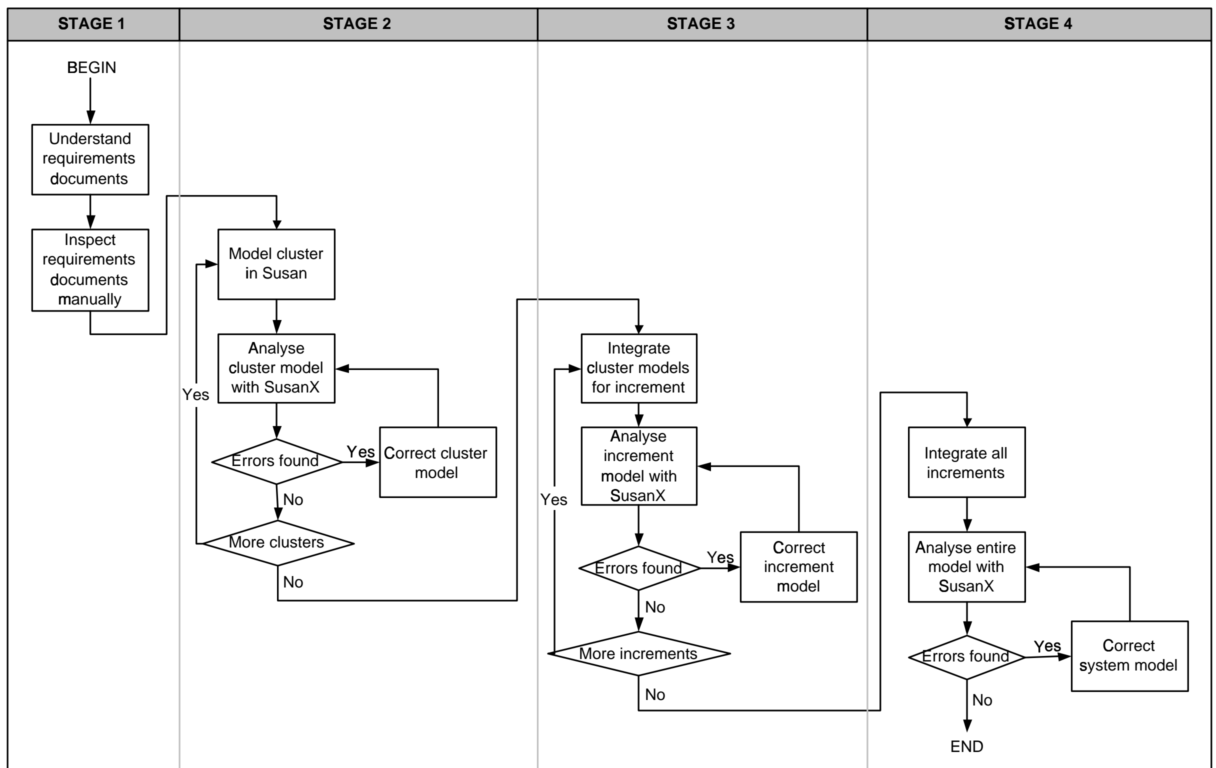


Figure 28: The Four Stages of the Case Study Process

As can be seen from the diagram in Figure 28, during Stage 1 we studied the CMS requirements documentation with the goal of understanding the system and also assessing the state of the requirements specification created by the SoftCo team. Additionally, we revealed several inconsistencies and shortcomings in the requirements models and documents at this point by inspecting them informally. In the requirements models provided, closely related use cases were grouped into functional “clusters” for each increment. In Stage 2 we began our modelling and analysis of requirements by transforming use case models for each cluster into Susan models, and subsequently analysing them separately from each other with SusanX. As shown in Figure 28 analysis and modelling activities often required a number of iterations. When errors were discovered during analysis, models were corrected and passed through the same analyses until no errors were reported. At the end of Stage 2 we were satisfied with the models for each cluster. In Stage 3, we combined cluster Susan models for each of the two increments and analysed these. During Stage 4, the two increment models were brought together to produce the entire model for the part of the system chosen for the case study. After this Susan model was analysed, we were confident that we had a more correct, complete and consistent requirements definition for the selected part of the CMS.

The remainder of this chapter describes our findings during each of the four stages of the case study. Certain parts of the CMS requirements model are included in the discussion for illustrative purposes. The complete Susan model obtained at the end of the modelling and analysis process can be found in Appendix A. The main results of the case study are summarised and evaluated at the end of this chapter.

7.3 Stage 1: Examining the Provided Documentation

The requirements definition documentation for the CMS consisted of a Software Requirements Specification (SRS) that provided an overview of the requirements for the entire system, and a Software Design Specification (SDS) for each increment containing requirements on a more detailed level. These two types of documents were structured in an analogous manner and contained the following sections.

1. **System overview:** This section contained a brief overview of the problem, which was the same for the SRS and all the SDS documents.

2. **Purpose of the document.** The purpose of the document type was described here.

3. **Document scope:** The scope of each document was described in terms of the required system features that it presented.

4. **Definitions and acronyms:** Terms that were often used in the documents, as well as certain use case modelling concepts were defined here.

5. **Assumptions and dependencies:** This section contained the assumptions made with regards to the requirements described in that particular document.

6. **Screen flow diagrams:** These diagrams gave an overview of the web pages that made up the web-interface and how they were linked to each other.

7. **Package diagram:** Each SDS included a package diagram showing the packages that were used to organise use cases for each of the increments.

8. **Use case diagrams:** A separate use case diagram for each of the packages was included in the documents.

9. **Use case descriptions:** A description for each use case appearing in the use case diagrams was provided. For each one, the following were included:

   - *Context of use* or a short description of the use case.
   - *Screen layout* of the web-interface for the use case, where appropriate.
   - *Actors* associated with the use case.
   - *Trigger* events that initiated the use case.
   - *Pre- and post-conditions* for the use case.
   - *Main and alternative flows* described textually as a sequence of steps and sometimes supplemented by activity diagrams.
   - *Notes and issues* providing additional information about the use case.

10. **Supplementary specification:** Only the SRS contained this section that described the non-functional requirements for the system.

Manual inspection of the captured requirements revealed a fair number of inconsistencies in the SRS and SDS documents. This showed that even in a mature software development company such as SoftCo, analysis at the early stages of development is not given a great

deal of consideration. Our most significant criticisms of the state of the CSM requirements definition documents were as follows.

There were no sound dependencies between the requirements documents. There were a number of inconsistencies between the requirements captured in the studied documents. As used in this particular project, the SRS was supposed to provide an overview of the requirements for the entire system or at least for Phase 1 of development. In reality however, the connection of this document to the SDSs was very questionable. For example, requirements for Increment 1 were not reflected at all in the SRS. Additionally, contradictory references between SDSs were identified. For instance, the SDS for Increment 2 made untrue assumptions about the system functionality described in the SDS for Increment 1. As a result, some use cases for Increment 2 were dependant on nonexistent or at least unexpressed requirements.

The documents were difficult to navigate. The requirements documents were not easy to follow for a number of reasons. Firstly, numerous diagrams in the documents were not labelled and appeared without any description. Secondly, cross-referencing between use case diagrams and use case descriptions was difficult. In fact, several cases were found where a use case appeared in a diagram but had no description or vice versa. Lastly, a few inconsistencies such as different ordering of sections and varied use case modelling notations were found between the documents. This can be attributed to the fact that each of the three documents that we examined (SRS, SDS for Increment 1 and SDS for Increment 2) were created by a different developer.

Incorrect use case modelling notation was used. Several use case diagrams contained invalid associations between modelling elements. For example, actors were connected with an extend relationship that should only be used to join use cases. Moreover, the inconsistency in notation used in the different documents showed that there was no general consensus on a particular extended or augmented use case modelling notation.

There were no actor descriptions. Use case diagrams were not supplemented with actor descriptions, which made it difficult to understand the precise roles that the actors represented. Selected actors were explained in the “Definitions and acronyms” section, but among the general definitions these explanations appeared out of place. Additional actor descriptions are not prescribed by the UML use case modelling notation, but it would enhance the comprehensibility of the requirements documents.

Use case pre- and post-conditions were incomplete. The definitions of pre- and post-conditions for use cases appeared to be incomplete and not well thought-through. Certain use cases had more than one main flow, but only one set of pre- and post-conditions were defined.

7.4 Stage 2: Modelling and Verifying Use Case Clusters

As mentioned before, use cases in the requirements documents provided were already grouped into functional clusters. For each one of these clusters we created a Susan model within the SusanX tool and analysed it for generic and model-specific properties. A brief overview of each of the clusters is given below.

Increment 1: Administrative System

- *Company admin.* The client business group includes several companies, and capturing of receipts for all of these needs to be done through the CMS. Each company can be associated with a number of business types, and each business type can have several pay codes. The administrator maintains a record of all the companies, business types and pay codes within the system.
- *Role admin.* A list of user roles is kept in the system, where each role is associated with a set of functions or access rights. The administrator is responsible for maintaining the roles within the system.
- *User admin.* A system user may have access to more than one company's receipt information. For each company that the user can access, a different role can be assigned to him. The maintenance of valid system users, and their assignment to companies and roles is once again the responsibility of the administrator.

Increment 2: Manual Receipts

- *Login.* A user is required to log into the system by providing his login name and password, as well as indicating with which company he wishes to work. Users are required to change their passwords regularly for security reasons.
- *Manual receipts.* Once a user is logged in, he gains access to manual receipt services provided by the system. He can capture details of a new receipt and either print it

straight away or save it for later use. Once a receipt is printed, it is posted to the accounting system and also sent to the operational systems. Printed and saved receipts are all stored within the CMS and can be searched by the user. An existing receipt can be opened and edited by the user. Receipts that have not been posted to the accounting system can also be deleted. In order to record a reversed transaction, the user can void an already posted receipt. Access to some of these services is restricted to certain user roles only.

The re-modelling of a cluster in the Susan notation required careful examination of the use case diagram for the cluster and the associated use case descriptions. Entities manipulated during the use case flows had to be pinpointed, as well as the attributes that uniquely identify these entities. These entity identifiers determine the variables that are used in a Susan model, and hence understanding these before creating a model is essential. For example, in the *Manual receipts* cluster receipts were manipulated and each receipt was uniquely identified by its receipt number and the ID of the issuing company.

For each cluster we first revised the associated use case diagram, correcting the notation and enhancing the structure where necessary. Figure 29 shows the original and revised use case diagrams for the *Manual Receipts* cluster. After the revision of a use case diagram, we flattened the diagram to make it suitable for input into SusanX. The flattened version of the revised *Manual Receipts* use case diagram is given in Figure 30.

As can be seen from Figures 29 (b) and 30, we replaced the three actors in the revised diagram by the *User* actor in the flattened diagram. Currently the Susan notation does not provide for actor inheritance and hence we included only one generic actor called *User*. The three roles to which the original actors corresponded (*Receipts enquiry clerk*, *Receipts capture clerk* and *Receipts supervisor*) and their influence on the services accessed by various users is expressed inside the textual elements of the Susan model. Nevertheless, the visual representation of these was lost due to the flattening of the use case diagram.

The flattened use case diagrams were constructed in SusanX, as the first step to creating complete Susan models in the tool. Next, the necessary variable types were inferred from the entity analysis performed earlier. For example, *Receipt number* and *Company ID* were two of the variable types in the *Manual receipts* cluster model. For each variable type a number of test values were assigned. Once variable types were in place, we added actor attributes and use case parameters to the model.

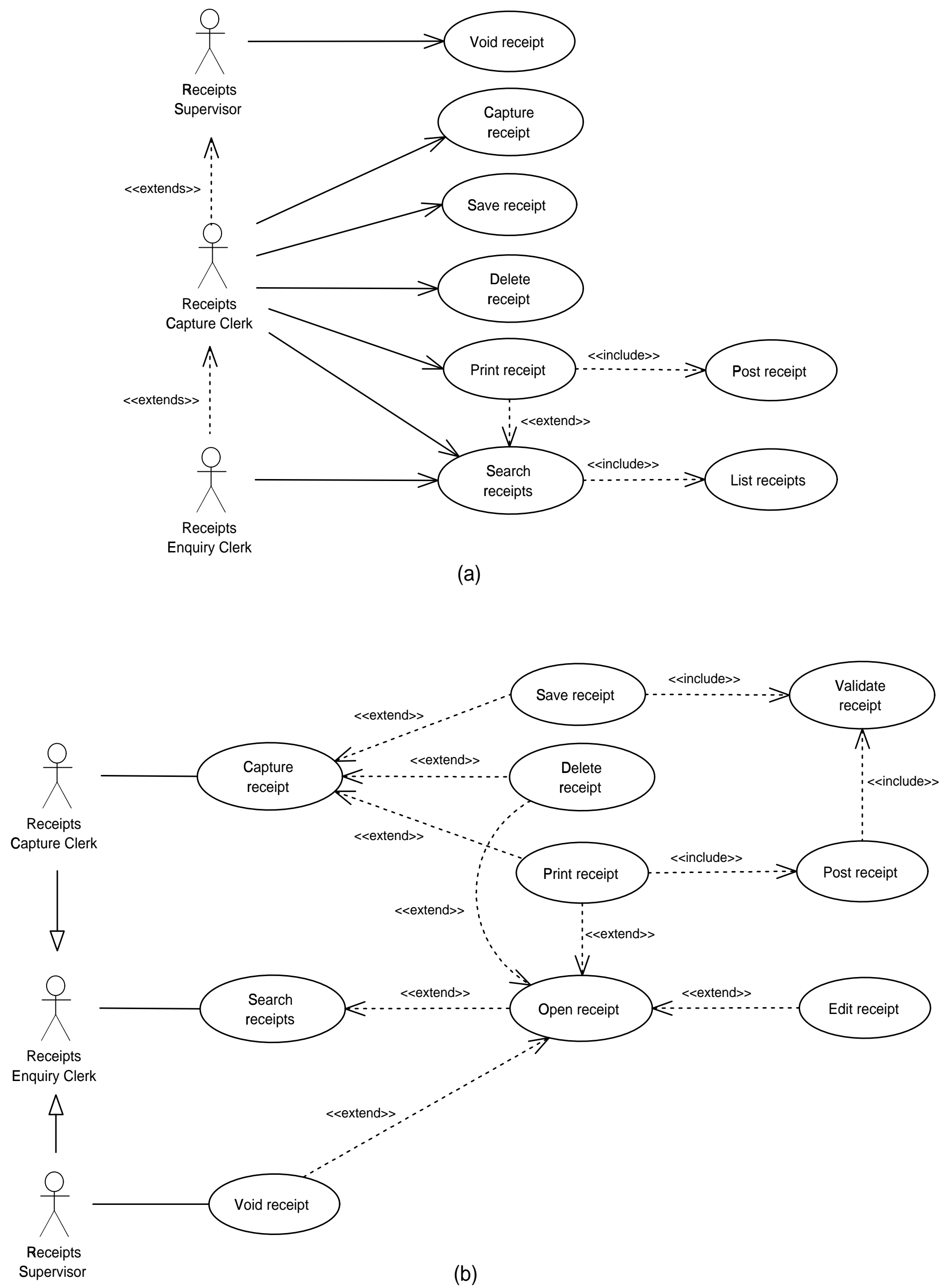


Figure 29: (a) Original and (b) Revised Use Case Diagrams for *Manual Receipts* Cluster

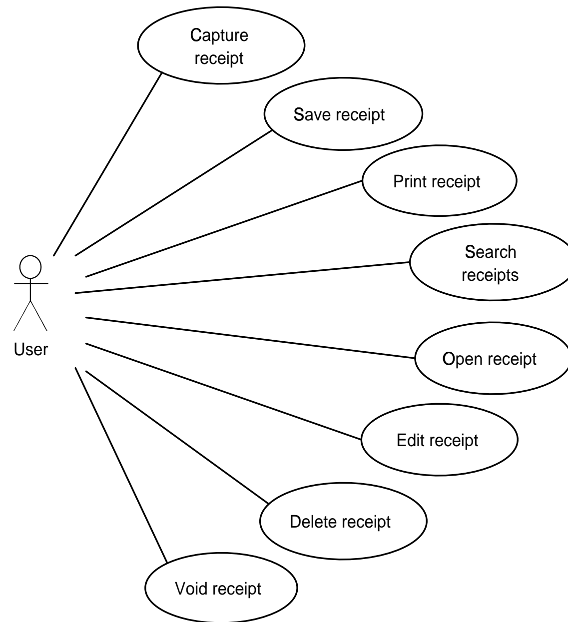


Figure 30: Use Case Diagram for *Manual Receipts* Cluster Flattened for Susan Model

The use case pre- and post-condition definitions in the original documents required considerable revisions before they could be used as input into SusanX. After revising these, we could complete the Susan models with condition declarations, use case pre- and post-conditions, as well as initial conditions for the system.

Analysing the CMS cluster models in SusanX presented a problem due to many dependencies between the clusters. In this context, we were concerned with dependencies where satisfaction of a pre-condition for a use case in one cluster depended on activation of a use case in another cluster. In cluster models where such dependencies existed, analysis produced results that were not valid for the entire system. For example, in the *Login* cluster model one of the pre-conditions for the *Log in* use case was *User exists*. This use case was reported as *Dead* by the SusanX generic verification, because the *User exists* pre-condition only became true after the *Add user* use case in the *User admin* cluster model was activated.

The cluster dependency problem was overcome in the following way. We first obtained the generic verification results for the *Role admin* and *Company admin* clusters, which were not dependent on any other clusters in the system. The *User admin* cluster model depended on both of these clusters through the *Role exists* and *Company exists* conditions. Inside the *Use admin* cluster model we used a *Dummy* actor and two use cases to mimic the changes

made to the *Role exists* and *Company exists* conditions. The reversibility categories of these two conditions were preserved by the mimicking, and hence the analysis results for the *User admin* cluster model could be generalised for the entire system. The dependencies related to the *Login* and *Manual receipts* clusters were handled in the same way.

Generic verification provided us with better insight into the cluster requirements models and also revealed a number of flaws in them. The insight was gained when we studied the results generated by verification, which required us to compare the computed category for each use case and condition with our expectations. A good understanding of the definitions for the use case liveness and condition reversibility categories was necessary to interpret the results produced by the generic verification and determine whether they were in order. For instance, contrast the following three cases where a condition was placed in the *Irreversible* category.

1. *Role flagged as deleted:* For auditing purposes, when a role is deleted it is not physically removed from the CMS database, but rather flagged as deleted. If a role is flagged deleted, it remains that way and there is no way to undo it. Hence, the *Irreversible* category was appropriate for this condition.

2. *Receipt opened:* If a previously saved receipt is opened by a user, it should be closed subsequently at some point in time. Therefore, this condition should actually have been in the *Reversible* category and the model needed to be corrected.

3. *Receipt saved in operational system:* This result implied that once a receipt was sent to the operational system, it remained there indefinitely. It might be the case that such a situation would not be desirable, however the removal of receipt records from operational systems is out of the CMS scope.

In cases where we established that the computed category for use case or condition was incorrect, the source of the error still had to be found. One of the flaws in the *Login* cluster model that we uncovered with generic verification was that the system did not provide a way for a user to log out. We identified this error because the *Log in* use case was categorised *Transient* and the *User logged in* condition *Irreversible*. This error was easily fixed by adding a *Log out* use case to the model. Another instance of an invalid verification result was found in the *Manual receipts* cluster, where the *Void receipt* use case was placed into the *Live* category. This meant that a particular receipt could be voided more than

once. Since every time a receipt is voided the corresponding transaction is reversed in the accounting and operational systems, this situation would ultimately result in incorrect transaction records. Taking into consideration that these transactions could involve very large amounts of money, such a flaw in the requirements model could have had devastating consequences. The model was corrected by adding a pre-condition to the *Void receipt* use case that ensured that the receipt in question has not been voided before.

Once we were satisfied with the results of the generic verification, we proceeded to analysis for model-specific properties. We found that the majority of the cluster models contained few tightly related use cases, which made it difficult to construct interesting and useful model-specific properties for the analysis. The *Manual receipts* cluster was the only model that we verified in this manner. For example, we confirmed that once a receipt is posted it cannot be deleted from the CMS. Additionally, we determined that a receipt could be edited after it has been posted, which should not have been the case. However, the validity of these properties could have been quite easily inferred by looking at the pre- and post-conditions defined for the *Delete receipt* and *Edit receipt* use cases instead of running model-specific verification. Nevertheless, even if in this particular case the effort of verifying properties of this nature by formal means was not justified, it very well may be for another type of system.

At the end of Stage 2 of our modelling and analysis process, the requirements model for the CMS was already considerably improved. The process of constructing Susan models for the system alone was very effective in making the requirements definition more precise and complete. Additionally, verification assisted us in correcting several logical flaws in the models. We concluded that for this particular system, generic verification was more helpful in analysing use case clusters than model-specific verification. Overall, the major drawback of the applied approach was presented by the flattening of the given use case diagrams for Susan models. Visual representation and some of the semantics of use case relationships and actor generalisations were lost due to the flattening process.

We felt that the interface of the SusanX tool provided very good support for building syntactically correct models quickly. This can be mainly attributed to the support SusanX provides to the user when prompting for input, such as for example restricting parameter assignment to valid values for the associated variable type with a drop-down list. A useful addition to the tool would be automatic detection of pre-conditions referring to a condition, the value of which does not change within the model. Dependencies among clusters could

have been identified more easily if such a feature was available. Furthermore, SusanX could automate the process of mimicking the reversibility of such conditions in the model by simply asking the user to provide the desired reversibility categories and taking care of the rest.

7.5 Stages 3 and 4: Integrating Cluster Models for Each Increment and Integrating Increment Models

This stage of the modelling and analysis process firstly involved bringing together the individual clusters for the two increments. Using the insight we obtained in the previous stage, we decided that the *Login* cluster model fitted in better in Increment 1 rather than Increment 2. Therefore, the *Company admin*, *Role admin*, *User admin* and *Login* cluster models were combined to form one model for Increment 1. On the other hand, the model for Increment 2 consisted of the unchanged *Manual receipts* cluster model.

Generic verification of the combined model for Increment 1 did not produce any new results, since the dependencies between our clusters were appropriately modelled in the previous stage. On the contrary, several valuable model-specific properties could now be constructed for the model analysis. For example, we discovered that non-existent users could have access to receipt services for a valid company. The generated counter-examples showed that there were two ways in which such a situation could arise. Firstly, when a user was deleted by the administrator, the company assignments involving that user were not removed from the system. Secondly, when details of a user were edited, this was not reflected in the relevant company assignments. Correcting the latter of these scenarios presented a problem, because there are no means of modelling the propagation of changed user details to the company assignments in Susan. In order to correct the former undesired situation, we appended the *Delete user* use case with post-conditions to remove all company associations for that user. However, the *Edit user* use case additionally required the creation of new associations for only those companies that were linked to the old user login. The Susan notation currently does not allow one to express this in a post-condition. Instead, we alleviated the problem by making the definition of the post-condition for the *Edit user* use case more abstract as demonstrated next.

Correcting the Use Case *Edit user*

BEFORE:	**AFTER:**
Actor:	*Actor:*
Administrator	Administrator
Parameters:	*Parameters:*
Old user *of type* User login,	User *of type* User login
New user *of type* User login	*Pre-conditions:*
Pre-conditions:	Admin logged in () *is* true,
Admin logged in () *is* true,	Users listed () *is* true,
Users listed () *is* true,	User exists (#*uc* User) *is* true
User exists (#*uc* Old user) *is* true,	*Post-conditions:*
User exists (#*uc* New user) *is* false	User details changed (#*uc* User) *is* true,
Post-conditions:	Company assignments changed (#*uc* User) *is* true
User exists (#*uc* Old user) *is* false,	
User exists (#*uc* New user) *is* true	

As can be seen above, instead of explicitly modelling the change of user logins, we now indicate the change more abstractly with *User details changed* and *Company assignments changed* post-conditions.

Another example of an error in the Increment 1 model discovered through model-specific verification, was that the administrator was able to delete a user while that user was logged into the system. We fixed this flaw by adding pre-conditions to the *Delete user* use case that checked that the user was not logged in under any company at the time the use case was activated.

Once we were satisfied with the state of the Susan model for Increment 1, we proceeded to Stage 4 and integrated all the created models to produce one Susan model for both Increments 1 and 2. This model was once again put through a number of analysis and correction cycles, where model-specific verification of SusanX was used for analysis. Our entire modelling and analysis process was complete when we had an acceptable Susan model for the selected part of the CMS.

As SusanX is currently a prototype tool, it provides limited support for model management such as grouping models into packages. In order to create the model for the whole of Increment 1, we had to manually construct a new model containing all the elements from the cluster models. SusanX could be enhanced with a feature that allowed the user to view multiple models at the same time, and analyse the models either individually or in groups.

7.6 Performance Measures for Analysis with SusanX

Our experiments with the case study were performed on a lightly loaded Intel Pentium M, 1.7 GHz processor with 512 MB RAM running Microsoft WindowsXP. During the course of the case study we captured performance data for all the analyses carried out with SusanX; these results are summarised in Table 3.

Model name	**TYP**	**CON**	**UC**	**Batch generic**	**Guided generic**	**Model-specific**
Company admin	3	10	12	37s	-	-
Role admin	1	5	6	5s	-	-
User admin (1)	3	8	6	16s	-	-
User admin (2)	3	8	8	31s	-	-
Increment 1 (1)	5	19	24	975s	-	31s (0.06)
Increment 1 (2)	5	20	27	711s	-	10s (0.03)
Login (1)	2	3	2	4s	-	-
Login (2)	2	3	4	5s	-	-
Manual receipts	4	11	10	381s	-	51s (0.37)
Entire system	6	29	35	∞	1122s (17.27)	18s (0)

Table 3: Times for CMS Model Verification with SusanX

In Table 3 above, the first three columns after the model name provide an approximate measure of the model sizes. TYP stands for the number of variable types declared in the model, CON - the number of conditions and UC - the number of use cases. These measures are followed by the captured times for the different types of analyses offered in SusanX, where seconds are used as the units of measure. In the cases where guided generic or model-specific verification was used, more than one time measurement was obtained. For these verification options, the Mean value is given in the table together with the calculated Standard Error of the Mean (SEM)[2], which is given in brackets next to the Mean. The SEM measure reflects the amount of variation between the individual time results in the verification data set and the accuracy of the Mean representation for the entire set. For Increment 1 (1), we can be 95% confident that the "true" Mean value for model-specific verification lies between 30.88s $(31 - 0.06 * 2)$ and 31.12s $(31 + 0.06 * 2)$. Therefore there was little variation between the time measurements obtained in that particular set. On the other hand, guided generic verification times for the Entire system varied greatly as shown by the high SEM value (17.27).

[2] $SEM = S/\sqrt{n}$ where n is sample size and S is Standard Deviation.

Batch generic verification for the Entire system did not complete as the NuSMV analysis exhausted the available memory, and this is shown by the infinity (∞) entry in the corresponding table cell. Generic verification categories were obtained through the guided option for this model instead.

On the whole, the performance of SusanX verification during the case study was good with all of the analyses completing under 20 minutes (or 1200s), except for the batch generic verification of the Entire system model. In that one case, we counteracted the problem with using guided generic verification instead on individual conditions and use cases in the model.

Performance or time taken by verification is the main concern in the application of model checkers in practice. A great deal of effort is currently going into solving the performance issues of model checking and considerable improvements have been made already, as discussed before. This leads us to believe that with a more efficient model checker than NuSMV, analysis of Susan models can be speeded up even more.

7.7 Summary and Evaluation of Results

On the whole our experience with the case study was extremely encouraging as it demonstrated that our proposed method for modelling and analysing software requirements is applicable in an industrial project. This is not to say that we were not confronted with any problems during the investigation, since there were in fact several difficulties that we encountered. Nonetheless, we managed to resolve all of these in one way or another and the solutions that we used could serve as valuable examples to potential users of the Susan method. Furthermore, during the case study we came across new ideas for improving Susan and the SusanX tool.

The concept of a use case cluster was introduced in this chapter to denote a group of use cases with tightly related functionality. Since the requirements models already appeared in such clusters in the CMS requirements documents, it seemed natural for us to adopt an incremental approach. We carried out iterations of modelling and analysis activities, gradually combining cluster Susan models until we had the final model for Increments 1 and 2. By modelling cluster dependencies appropriately, we obtained most of our results for generic verification by running analyses on the small cluster models in Stage 2. Most of these results were valid for the entire model, and hence we were relieved from repeating the expensive generic verification procedure for the larger models in Stages 3 and 4.

We conclude this chapter by summarising the results obtained from our experience of modelling and analysing the CMS, in terms of the objectives that we posed for the case study in Section 7.1.

1. *Suitability of Susan notation.* The Susan notation was appropriate for representing textual aspects of the CMS requirements usually included in use case models. Additionally, certain non-functional requirements that expressed constraints on system behaviour could be incorporated into the Susan models at the time of analysis as verification properties. As expected, the visual perception of use case diagrams deteriorated by the flattening process required by the SusanX tool.

2. *Benefits of formalising use case models.* Formalising the given requirements models with Susan improved the requirements specification for the CMS considerably. Firstly, merely the process of creating Susan models was very constructive in assisting us with achieving a full grasp on the requirements for the system. During this process several inconsistencies and missing details were identified and remedied. Secondly, the Susan models created expressed the CMS requirements in a much more precise and comprehensible manner than the original models.

3. *Effectiveness of SusanX analysis options.* Both generic and model-specific analysis options in SusanX proved to be beneficial. It was discovered that for small models consisting of tightly related use cases, model-specific verification was not always effective. However, generic verification was still valuable even for small models.

4. *Usability and performance of SusanX.* The current features offered in SusanX were found to be very accessible. Judging by our own experience, we suspect that any user familiar with the Susan method should find the SusanX tool easy to use. A number of additional features such as multiple model view mentioned in Section 7.5, could make SusanX even more effective. We could obtain all the required results from the analysis options offered by SusanX in a reasonable amount of time, from which we conclude that it is feasible to use the tool for real-world software systems that are comparable to the CMS in size.

On the completion of the case study we presented our results and shared our experience with SoftCo, who showed much interest in the Susan technique and especially the accompanying tool. The team that was involved in the CMS project validated all our criticisms

with respect to the condition of the requirements specification documents. We also learnt from the team that most of the inconsistencies and errors that we identified during our case study were discovered later in the development process, but not reflected in the original requirements definition. The client company was satisfied with the delivered system and little support has been required since its deployment. Nevertheless, if the CMS requirements were modelled and analysed as described in this chapter, this propagation of errors into other development phases could have been avoided.

Chapter 8

Conclusions and Future Work

In this dissertation we addressed the problem posed by the lack of an adequate method for requirements modelling and analysis. In order to solve this problem we proposed a technique called Susan, which extends and alleviates the weaknesses of use case modelling. Susan constitutes a novel way of formalising use case models, as well as using rigorous means for performing analysis of these models.

Through the Cash Management System (CMS) case study, we showed that using a notation that is precise and yet relatively simple to use is very advantageous for modelling requirements. Modelling the CMS requirements in Susan produced specifications that were more detailed, structured, correct, consistent and complete.

The conceptual framework for rigorous analysis in Susan is based on model checking, specification patterns and a our own scheme of generic verification properties for Susan models. Our case study demonstrated that analysis methods incorporated into Susan can assist in discovering errors in requirements models, ensure that a model satisfies certain constraints and also provide the user with better insight into system's requirements. Thus our proposed method allows developers to eliminate requirements discrepancies early in the development cycle.

In conclusion, we note that the Susan method meets all the desirable criteria identified in Chapter 1, page 7. Its modelling notation is suitable for representing software requirements, it is unambiguous and at the same time understandable. The method incorporates a rigorous analysis technique and is supported by the SusanX tool.

8.1 Future Work

A number of further developments of Susan and SusanX that would be interesting and beneficial are discussed below.

1. *Extending the Susan metamodel.* Currently Susan does not support several constructs from standard use case modelling, including actor generalisation and relationships among use cases. Extending the Susan metamodel to incorporate these constructs would make the Susan method more expressive and possibly allow for more valuable analysis of requirements models. This would also eliminate the need for the flattening of use case diagrams, thus preserving their visual characteristics. Additionally, during our case study we informally introduced the notion of modelling requirements in use case clusters. This concept would be useful in handling large models and should be formalised and explicitly included into Susan.

2. *Undertaking further case studies.* Additional case studies on applying Susan and SusanX in practice, thus providing more thorough evaluations, would be very beneficial. Requirements for different types of systems could be modelled and analysed to determine the circumstances under which our method is most effective. We worked with existing use case models in the CMS case study, but going through the modelling process from the beginning using Susan would also be worth investigating.

3. *Adding new features to SusanX.* In its current state, SusanX is a prototype tool supporting the Susan method. It could be significantly extended with various features as already mentioned in Chapter 7. Examples of these are multiple model view and use case cluster support.

4. *Integrating Susan with other development phases.* At the moment there is no prescribed way for using the information captured in Susan models further in the development process. In general, few requirements modelling methods allow for transfer of models or model elements to later phases. One way of doing this with Susan would be the generation of test cases from the information captured in Susan models.

Appendix A

Susan Model for CMS

The complete Susan model for the Cash Management System (CMS) discussed in Chapter 7 is given in this appendix. Note that this is the final version of the model, in other words all the errors discovered by SusanX analyses have already been corrected. The use case diagram for this model can be seen from the screenshot of SusanX in Figure 31. The textual definitions of all the model elements are given below.

Variable type 1.
Name: Company ID
Values: OMLAC, OMEB

Variable type 2.
Name: Business type description
Values: Life, Unit Trust, EB, Life Legacy

Variable type 3.
Name: Pay code
Values: SA, MA

Variable type 4.
Name: Role description
Values: Receipts capture clerk, Receipts enquiry clerk, Receipts supervisor

Variable type 5.
Name: User login
Values: jbloggs, mjane, agatonye, enjegi

Variable type 6.
Name: Receipt number
Values: REC324273452, REC324425452

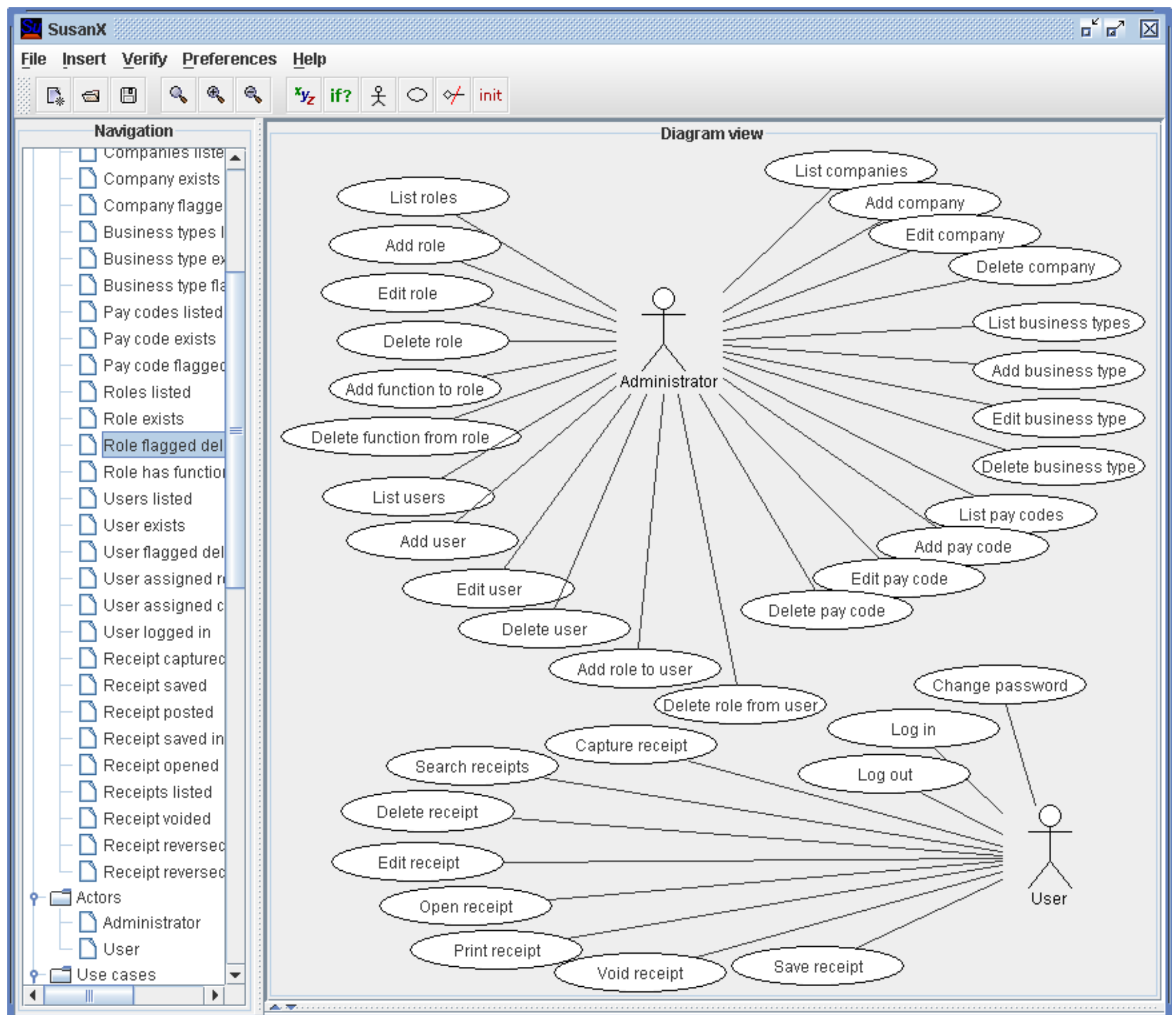


Figure 31: CMS Model in SusanX

Condition 1.
Name: Admin logged in

Condition 2.
Name: Companies listed

Condition 3.
Name: Company exists
Parameters: Company *of type* Company ID

Condition 4.
Name: Company flagged deleted
Parameters: Company *of type* Company ID

Condition 5.
Name: Business types listed
Parameters: Company *of type* Company ID

Condition 6.
Name: Business type exists
Parameters: Company *of type* Company ID, Business type *of type* Business type description

Condition 7.
Name: Business type flagged deleted
Parameters: Company *of type* Company ID, Business type *of type* Business type description

Condition 8.
Name: Pay codes listed
Parameters: Company *of type* Company ID, Business type *of type* Business type description

Condition 9.
Name: Pay code exists
Parameters: Company *of type* Company ID, Business type *of type* Business type description, Pay code *of type* Pay code

Condition 10.
Name: Pay code flagged deleted
Parameters: Company *of type* Company ID, Business type *of type* Business type description, Pay code *of type* Pay code

Condition 11.
Name: Roles listed

Condition 12. Name: Role exists
Parameters: Role *of type* Role description

Condition 13.
Name: Role flagged deleted

Parameters: Role *of type* Role description

Condition 14.
Name: Role has functions
Parameters: Role *of type* Role description

Condition 15.
Name: Users listed

Condition 16.
Name: User exists
Parameters: User *of type* User login

Condition 17.
Name: User flagged deleted
Parameters: User *of type* User login

Condition 18.
Name: User assigned role
Parameters: User *of type* User login, Role *of type* Role description, Company *of type* Company ID

Condition 19.
Name: User assigned company
Parameters: User *of type* User login, Company *of type* Company ID

Condition 20.
Name: User logged in
Parameters: User *of type* User login, Company *of type* Company ID

Condition 21.
Name: Receipt captured
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Condition 22.
Name: Receipt saved
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Condition 23.
Name: Receipt posted
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Condition 24.
Name: Receipt saved in back-end system
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Condition 25.
Name: Receipt opened
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Condition 26.
Name: Receipts listed
Parameters: Company *of type* Company ID

Condition 27.
Name: Receipt voided
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Condition 28.
Name: Receipt reversed
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Condition 29.
Name: Receipt reversed in back-end system
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number

Actor 1.
Name: Administrator

Actor 2.
Name: User
Attributes: Login *of type* User login

Use case 1.
Name: List companies
Actor: Administrator
Pre-conditions: Admin logged in () *is* true
Post-conditions: Companies listed () *is* true

Use case 2.
Name: Add company
Actor: Administrator
Parameters: Company *of type* Company ID
Pre-conditions: Admin logged in () *is* true, Companies listed () *is* true,
Company exists (#uc Company) *is* false
Post-conditions: Company exists (#*uc* Company) *is* true

Use case 3. Name: Edit company
Actor: Administrator
Parameters: Company *of type* Company ID
Pre-conditions: Admin logged in () *is* true, Companies listed () *is* true,
Company exists (#*uc* Company) *is* true
Post-conditions: none

Use case 4.
Name: Delete company
Actor: Administrator

Parameters: Company *of type* Company ID
Pre-conditions: Admin logged in () *is* true, Companies listed () *is* true,
Company exists (#*uc* Company) *is* true
Post-conditions: Company exists (#*uc* Company) *is* false,
Company flagged deleted (#*uc* Company) *is* true,
User assigned role (#*forall* User login, #*forall* Role description, #*uc* Company) *is* false,
User assigned company (#*forall* User login, #*uc* Company) *is* false,
Business type exists (#*uc* Company, #*forall* Business type description) *is* false,
Pay code exists (#*uc* Company, #*forall* Business type description, #*forall* Pay code) *is* false

Use case 5.
Name: List business types
Actor: Administrator
Parameters: Company *of type* Company ID
Pre-conditions: Admin logged in () *is* true, Companies listed () *is* true,
Company exists (#*uc* Company) *is* true
Post-conditions: Business types listed (#*uc* Company) *is* true

Use case 6.
Name: Add business type
Actor: Administrator
Parameters: Company *of type* Company ID, Business type *of type* Business type description
Pre-conditions: Admin logged in () *is* true, Business types listed (#*uc* Company) *is* true,
Business type exists (#*uc* Company, #*uc* Business type) *is* false,
Company exists (#*uc* Company) *is* true
Post-conditions: Business type exists (#*uc* Company, #*uc* Business type) *is* true

Use case 7.
Name: Edit business type
Actor: Administrator
Parameters: Business type *of type* Business type description, Company *of type* Company ID
Pre-conditions: Admin logged in () *is* true, Business types listed (#*uc* Company) *is* true,
Business type exists (#*uc* Company, #*uc* Business type) *is* true
Post-conditions: none

Use case 8.
Name: Delete business type
Actor: Administrator
Parameters: Company *of type* Company ID, Business type *of type* Business type description
Pre-conditions: Admin logged in () *is* true, Business types listed (#*uc* Company) *is* true,
Business type exists (#*uc* Company, #*uc* Business type) *is* true
Post-conditions: Business type exists (#*uc* Company, #*uc* Business type) *is* false,
Business type flagged deleted (#*uc* Company, #*uc* Business type) *is* true,
Pay code exists (#*uc* Company, #*uc* Business type, #*forall* Pay code) *is* false

Use case 9.
Name: List pay codes
Actor: Administrator

Parameters: Company *of type* Company ID, Business type *of type* Business type description
Pre-conditions: Admin logged in () *is* true, Business types listed (*#uc* Company) *is* true,
Business type exists (*#uc* Company, *#uc* Business type) *is* true
Post-conditions: Pay codes listed (*#uc* Company, *#uc* Business type) *is* true

Use case 10.
Name: Add pay code
Actor: Administrator
Parameters: Company *of type* Company ID, Business type *of type* Business type description,
Pay code *of type* Pay code
Pre-conditions: Admin logged in () *is* true,
Pay codes listed (*#uc* Company, *#uc* Business type) *is* true,
Pay code exists (*#uc* Company, *#uc* Business type, *#uc* Pay code) *is* false,
Business type exists (*#uc* Company, *#uc* Business type) *is* true
Post-conditions: Pay code exists (*#uc* Company, *#uc* Business type, *#uc* Pay code) *is* true

Use case 11.
Name: Edit pay code
Actor: Administrator
Parameters: Company *of type* Company ID, Business type *of type* Business type description,
Pay code *of type* Pay code
Pre-conditions: Admin logged in () *is* true,
Pay codes listed (*#uc* Company, *#uc* Business type) *is* true,
Pay code exists (*#uc* Company, *#uc* Business type, *#uc* Pay code) *is* true
Post-conditions: none

Use case 12.
Name: Delete pay code
Actor: Administrator
Parameters: Company *of type* Company ID, Business type *of type* Business type description,
Pay code *of type* Pay code
Pre-conditions: Admin logged in () *is* true,
Pay codes listed (*#uc* Company, *#uc* Business type) *is* true,
Pay code exists (*#uc* Company, *#uc* Business type, *#uc* Pay code) *is* true
Post-conditions: Pay code exists (*#uc* Company, *#uc* Business type, *#uc* Pay code) *is* false,
Pay code flagged deleted (*#uc* Company, *#uc* Business type, *#uc* Pay code) *is* true

Use case 13.
Name: List roles
Actor: Administrator
Pre-conditions: Admin logged in () *is* true
Post-conditions: Roles listed () *is* true

Use case 14.
Name: Add role
Actor: Administrator
Parameters: Role *of type* Role description
Pre-conditions: Admin logged in () *is* true, Roles listed () *is* true, Role exists (*#uc* Role) *is* false

Post-conditions: Role exists (*#uc* Role) *is* true

Use case 15.
Name: Edit role
Actor: Administrator
Parameters: Role *of type* Role description
Pre-conditions: Admin logged in () *is* true, Roles listed () *is* true, Role exists (*#uc* Role) *is* true

Use case 16.
Name: Delete role
Actor: Administrator
Parameters: Role *of type* Role description
Pre-conditions: Admin logged in () *is* true, Roles listed () *is* true, Role exists (*#uc* Role) *is* true
Post-conditions: Role exists (*#uc* Role) *is* false, Role flagged deleted (*#uc* Role) *is* true,
User assigned role (*#forall* User login, *#uc* Role, *#forall* Company ID) *is* false

Use case 17.
Name: Add function to role
Actor: Administrator
Parameters: Role *of type* Role description
Pre-conditions: Admin logged in () *is* true, Roles listed () *is* true, Role exists (*#uc* Role) *is* true
Post-conditions: Role has functions (*#uc* Role) *is* true

Use case 18.
Name: Delete function from role
Actor: Administrator
Parameters: Role *of type* Role description
Flow 1, Pre-conditions: Admin logged in () *is* true, Roles listed () *is* true,
Role exists (*#uc* Role) *is* true, Role has functions (*#uc* Role) *is* true
Flow 1, Post-conditions: none
Flow 2, Pre-conditions: Admin logged in () *is* true, Roles listed () *is* true,
Role exists (*#uc* Role) *is* true, Role has functions (*#uc* Role) *is* true
Flow 2, Post-conditions: Role has functions (*#uc* Role) *is* false

Use case 19.
Name: List users
Actor: Administrator
Pre-conditions: Admin logged in () *is* true
Post-conditions: Users listed () *is* true

Use case 20.
Name: Add user
Actor: Administrator
Parameters: User *of type* User login
Pre-conditions: Admin logged in () *is* true, Users listed () *is* true, User exists (*#uc* User) *is* false
Post-conditions: User exists (*#uc* User) *is* true

Use case 21.
Name: Edit user
Actor: Administrator
Parameters: User *of type* User login
Pre-conditions: Admin logged in () *is* true, Users listed () *is* true, User exists (#*uc* User) *is* true

Use case 22.
Name: Delete user
Actor: Administrator
Parameters: User *of type* User login
Pre-conditions: Admin logged in () *is* true, Users listed () *is* true, User exists (#*uc* User) *is* true,
User logged in (#*uc* User, #*forall* Company ID) *is* false
Post-conditions: User exists (#*uc* User) *is* false, User flagged deleted (#*uc* User) *is* true,
User assigned role (#*uc* User, #*forall* Role description, #*forall* Company ID) *is* false,
User assigned company (#*uc* User, #*forall* Company ID) *is* false

Use case 23.
Name: Add role to user
Actor: Administrator
Parameters: User *of type* User login, Role *of type* Role description, Company *of type* Company ID
Pre-conditions: Admin logged in () *is* true, Users listed () *is* true, User exists (#*uc* User) *is* true,
Role exists (#*uc* Role) *is* true, Company exists (#*uc* Company) *is* true
Post-conditions: User assigned role (#*uc* User, #*uc* Role, #*uc* Company) *is* true,
User assigned company (#*uc* User, #*uc* Company) *is* true

Use case 24.
Name: Delete role from user
Actor: Administrator
Parameters: User *of type* User login, Role *of type* Role description, Company *of type* Company ID
Flow 1, Pre-conditions: Admin logged in () *is* true,
User assigned role (#*uc* User, #*uc* Role, #*uc* Company) *is* true, Users listed () *is* true,
User logged in (#*uc* User, #*uc* Company) *is* false
Flow 1, Post-conditions: User assigned role (#*uc* User, #*uc* Role, #*uc* Company) *is* false
Flow 2, Pre-conditions: Admin logged in () *is* true, Users listed () *is* true,
User assigned role (#*uc* User, #*uc* Role, #*uc* Company) *is* true
Flow 2, Post-conditions: User assigned role (#*uc* User, #*uc* Role, #*uc* Company) *is* false,
User assigned company (#*uc* User, #*uc* Company) *is* false

Use case 25.
Name: Log in
Actor: User
Parameters: Company *of type* Company ID
Pre-conditions: User logged in (#*self* Login, #*forall* Company ID) *is* false,
User exists (#*self* Login) *is* true, User assigned company (#*self* Login, #*uc* Company) *is* true
Post-conditions: User logged in (#*self* Login, #*uc* Company) *is* true

Use case 26.
Name: Change password

Actor: User
Parameters: Company *of type* Company ID
Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true

Use case 27.
Name: Log out
Actor: User
Parameters: Company *of type* Company ID
Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true
Post-conditions: User logged in (*#self* Login, *#uc* Company) *is* false

Use case 28.
Name: Capture receipt
Actor: User
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number
Flow 1, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts capture clerk *of type* Role description, *#uc* Company) *is* true
Flow 1, Post-conditions: Receipt captured (*#uc* Company, *#uc* Receipt) *is* true,
Receipts listed (*#uc* Company) *is* false
Flow 2, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts supervisor *of type* Role description, *#uc* Company) *is* true
Flow 2, Post-conditions: Receipt captured (*#uc* Company, *#uc* Receipt) *is* true,
Receipts listed (*#uc* Company) *is* false

Use case 29.
Name: Save receipt
Actor: User
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number
Flow 1, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts capture clerk *of type* Role description, *#uc* Company) is true,
Receipt captured (*#uc* Company, *#uc* Receipt) *is* true
Flow 1, Post-conditions: Receipt saved (*#uc* Company, *#uc* Receipt) *is* true
Flow 2, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts supervisor *of type* Role description, *#uc* Company) *is* true,
Receipt captured (*#uc* Company, *#uc* Receipt) *is* true
Flow 2, Post-conditions: Receipt saved (*#uc* Company, *#uc* Receipt) *is* true

Use case 30.
Name: Print receipt
Actor: User
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number
Flow 1, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts capture clerk *of type* Role description, *#uc* Company) *is* true,

Receipt captured (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* false
Flow 1, Post-conditions: Receipt posted (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved in back-end system (#*uc* Company, #*uc* Receipt) *is* true
Flow 2, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts capture clerk *of type* Role description, #*uc* Company) *is* true,
Receipt captured (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* true
Flow 2, Post-conditions: none
Flow 3, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#self Login, Receipts capture clerk of type Role description, #uc Company) is true,
Receipt opened (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* false
Flow 3, Post-conditions: Receipt posted (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved in back-end system (#*uc* Company, #*uc* Receipt) *is* true
Flow 4, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts capture clerk *of type* Role description, #*uc* Company) *is* true,
Receipt opened (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* true
Flow 4, Post-conditions: none
Flow 5, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts supervisor *of type* Role description, #*uc* Company) *is* true,
Receipt captured (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* false
Flow 5, Post-conditions: Receipt posted (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved in back-end system (#*uc* Company, #*uc* Receipt) *is* true
Flow 6, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts supervisor *of type* Role description, #*uc* Company) *is* true,
Receipt captured (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* true
Flow 6, Post-conditions: none
Flow 7, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts supervisor *of type* Role description, #*uc* Company) *is* true,
Receipt opened (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* false
Flow 7, Post-conditions: Receipt posted (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved (#*uc* Company, #*uc* Receipt) *is* true,
Receipt saved in back-end system (#*uc* Company, #*uc* Receipt) *is* true
Flow 8, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,

User assigned role (*#self* Login, Receipts supervisor *of type* Role description, *#uc* Company) *is* true,
Receipt opened (*#uc* Company, *#uc* Receipt) *is* true,
Receipt posted (*#uc* Company, *#uc* Receipt) *is* true
Flow 8, Post-conditions: none

Use case 31.
Name: Search receipts
Actor: User
Parameters: Company *of type* Company ID
Flow 1, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts enquiry clerk *of type* Role description, *#uc* Company) *is* true
Flow 1, Post-conditions: Receipts listed (*#uc* Company) *is* true,
Receipt captured (*#uc* Company, *#forall* Receipt number) *is* false,
Receipt opened (*#uc* Company, *#forall* Receipt number) *is* false
Flow 2, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts capture clerk *of type* Role description, *#uc* Company) *is* true
Flow 2, Post-conditions: Receipts listed (*#uc* Company) *is* true,
Receipt captured (*#uc* Company, *#forall* Receipt number) *is* false,
Receipt opened (*#uc* Company, *#forall* Receipt number) *is* false
Flow 3, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts supervisor *of type* Role description, *#uc* Company) *is* true
Flow 3, Post-conditions: Receipts listed (*#uc* Company) *is* true,
Receipt captured (*#uc* Company, *#forall* Receipt number) *is* false,
Receipt opened (*#uc* Company, *#forall* Receipt number) *is* false

Use case 32.
Name: Open receipt
Actor: User
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number
Flow 1, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts enquiry clerk *of type* Role description, *#uc* Company) *is* true,
Receipts listed (*#uc* Company) *is* true,
Receipt saved (*#uc* Company, *#uc* Receipt) *is* true
Flow 1, Post-conditions: Receipt opened (*#uc* Company, *#uc* Receipt) *is* true,
Receipts listed (*#uc* Company) *is* false
Flow 2, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts capture clerk *of type* Role description, *#uc* Company) *is* true,
Receipts listed (*#uc* Company) *is* true,
Receipt saved (*#uc* Company, *#uc* Receipt) *is* true
Flow 2, Post-conditions: Receipt opened (*#uc* Company, *#uc* Receipt) *is* true,
Receipts listed (*#uc* Company) *is* false
Flow 3, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,

User assigned role (*#self* Login, Receipts supervisor *of type* Role description, *#uc* Company) *is* true,
Receipts listed (*#uc* Company) *is* true,
Receipt saved (*#uc* Company, *#uc* Receipt) *is* true
Flow 3, Post-conditions: Receipt opened (*#uc* Company, *#uc* Receipt) *is* true,
Receipts listed (*#uc* Company) *is* false

Use case 33.
Name: Edit receipt
Actor: User
Parameters: Company *of type* Company ID Receipt *of type* Receipt number
Flow 1, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts capture clerk *of type* Role description, *#uc* Company) *is* true,
Receipts listed (*#uc* Company) *is* true,
Receipt posted (*#uc* Company, *#uc* Receipt) *is* false
Flow 1, Post-conditions: none
Flow 2, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts supervisor *of type* Role description, *#uc* Company) *is* true,
Receipt opened (*#uc* Company, *#uc* Receipt) *is* true,
Receipt posted (*#uc* Company, *#uc* Receipt) *is* false
Flow 2, Post-conditions: none

Use case 34.
Name: Void receipt
Actor: User
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number
Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts supervisor *of type* Role description, *#uc* Company) *is* true,
Receipt opened (*#uc* Company, *#uc* Receipt) *is* true,
Receipt posted (*#uc* Company, *#uc* Receipt) *is* true,
Receipt voided (*#uc* Company, *#uc* Receipt) *is* false
Post-conditions: Receipt voided (*#uc* Company, *#uc* Receipt) *is* true,
Receipt reversed (*#uc* Company, *#uc* Receipt) *is* true,
Receipt reversed in back-end system (*#uc* Company, *#uc* Receipt) *is* true

Use case 35.
Name: Delete receipt
Actor: User
Parameters: Company *of type* Company ID, Receipt *of type* Receipt number
Flow 1, Pre-conditions: User logged in (*#self* Login, *#uc* Company) *is* true,
User assigned role (*#self* Login, Receipts capture clerk *of type* Role description, *#uc* Company) *is* true,
Receipt opened (*#uc* Company, *#uc* Receipt) *is* true,
Receipt posted (*#uc* Company, *#uc* Receipt) *is* false
Flow 1, Post-conditions: Receipt saved (*#uc* Company, *#uc* Receipt) *is* false,

Receipt opened (#*uc* Company, #*uc* Receipt) *is* false
Flow 2, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts supervisor *of type* Role description, #*uc* Company) *is* true,
Receipt opened (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* false
Flow 2, Post-conditions: Receipt saved (#*uc* Company, #*uc* Receipt) *is* false,
Receipt opened (#*uc* Company, #*uc* Receipt) *is* false
Flow 3, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts capture clerk *of type* Role description, #*uc* Company) *is* true,
Receipt captured (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* false
Flow 3, Post-conditions: Receipt saved (#*uc* Company, #*uc* Receipt) *is* false,
Receipt captured (#*uc* Company, #*uc* Receipt) *is* false
Flow 4, Pre-conditions: User logged in (#*self* Login, #*uc* Company) *is* true,
User assigned role (#*self* Login, Receipts supervisor *of type* Role description, #*uc* Company) *is* true,
Receipt captured (#*uc* Company, #*uc* Receipt) *is* true,
Receipt posted (#*uc* Company, #*uc* Receipt) *is* false
Flow 4, Post-conditions: Receipt saved (#*uc* Company, #*uc* Receipt) *is* false,
Receipt captured (#*uc* Company, #*uc* Receipt) *is* false

Initial condition 1.
Name: Administrator logged in
Condition: Admin logged in ()

Bibliography

[AB95] Michael Andersson and Johan Bergstrand. Formalizing Use Cases with Message Sequence Charts. Master's thesis, Department of Communication Systems at Lund Institute of Technology, Sweden, May 1995.

[ASJ01] Bente Anda, Dag Sjoberg, and Magne Jorgensen. Quality and Understandability of Use Case Models. In *Proceedings of ECOOP 2001 - Object-Oriented Programming: 15th European Conference*, LNCS, page 402, Budapest, Hungary, June 2001.

[BBDEL96] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. RuleBase: An Industry-Oriented Formal Verification Tool. In *Proceedings of the 33rd Annual Conference on Design Automation*, pages 655–660, Las Vegas, Nevada, United States, 1996. ACM Press.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT Procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[BCG88] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59(1-2):115–131, 1988.

[BM84] R.S. Boyer and J.S. Moore. Proof-Checking, Theorem-Proving, and Program Verification. *In Contemporary Mathematics, Automated Theorem Proving: After 25 Years, American Mathematical Society*, pages 119–132, 1984.

[BPG+01] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. A Knowledge Level Software Engineering Methodology for Agent

Oriented Programming. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 648–655, Montreal, Quebec, Canada, 2001. ACM Press.

[BPPP99] Ralph-Johan Back, Luigia Petre, and Ivan Porres-Paltor. Analyzing UML Use Cases as Contracts. In *UML'99 - Second International Conference on the Unified Modeling Language: Beyond the Standard*, pages 518 – 533. Springer-Verlag, October 1999.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language*. Addison-Wesley, 1999.

[BS03] Kurt Bittner and Ian Spence. *Use Case Modeling*. Addison-Wesley, June 2003.

[BW98] R-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[CAB+98] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th Conference on Computer-Aided Verification (CAV'99)*, number 1633 in LNCS, pages 495–499, Trento, Italy, July 1999. Springer.

[CCMS02] Laura A. Campbell, Betty H. C. Cheng, William E. McUmber, and R. E. K. Stirewalt. Automatically Detecting and Visualising Errors in UML Diagrams. *Requirements Engineering*, 7(4):264 – 287, December 2002.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *In Logics of Programs: Workshop*, volume 131 of LNCS, Yorktown Heights, New York, May 1981. Springer.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sisla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic. In *ACM Transactions on Programming Languages and Systems*, volume 8, pages 244–263, 1986.

[Coc00] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[CTB04] Marshini Chetty, William Tucker, and Edwin Blake. Developing Locally Relevant Software Applications for Rural Areas: A South African Example. In *Proceedings of SAICSIT 2004*, LNCS, Cape Town, South Africa, October 2004.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[DAC04] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Online Repository for Information about Specificaiton Patterns for Finite-state Verification. Available online: http://patterns.projects.cis.ksu.edu/, Last accessed: November 2004.

[DBDP94] E. Dubois, P. Bois, F. Dubru, and M. Petit. Agent-Oriented Requirements Engineering - A Case Study Using the Albert Language. In Proceedings of the 4th International Working Conference on Dynamic Modelling and Information Systems, 1994.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[DDMvL97] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an Environment for Goal-Driven Requirements Engineering. In *Proceedings of the 19th International Conference on Software Engineering*, pages 612–613, Boston, Massachusetts, United States, 1997. ACM Press.

[DKK02] Ian Davies, Pieter Kritzinger, and William Knottenbelt. Symbolic Methods for the State Space Exploration of GSPN Models. In *Proceedings of TOOLS 2002*, LNCS, London, United Kingdom, April 2002.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[Ebe97] Armin Paul-Gerhard Eberlein. *Requirements Acquisition and Specification for Telecommunication Services*. PhD thesis, University of Wales, November 1997.

[EP02] Cindy Eisner and Doron Peled. Comparing Symbolic and Explicit Model Checking of a Software System. In *In Proceedings of SPIN Workshop on Model Checking of Software*, volume 2318 of LNCS, pages 230–239. Springer, 2002.

[Fit96] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Texts in Computer Science. Springer, 2nd edition, 1996.

[Fra03] D.S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[GFST91] S. Greenspan, M. Feblowitz, C. Shekaran, and J. Tremlett. Addressing Requirements Issues Within a Conceptual Modeling Environment. In *Proceedings of the 6th International Workshop on Software Specification and Design*, October 1991.

[GHJV] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*.

[GMB82] S. Greenspan, J. Mylopoulos, and A. Borgida. Capturing More World Knowledge in the Requirements Specification. In *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, 1982.

[GMP02] Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams. In *Proceedings*

of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems, pages 35–36, Bologna, Italy, 2002. ACM Press.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HH87] H.Banieqbal and H.Barringer. Temporal Logic with Fixed Points. *Temporal Logic in Specification*, 398 of LNCS:62–74, November 1987.

[HHT02] Jan Hendrik Hausmann, Reiko Heckel, and Gabi Taentzer. Detection of Conflicting Functional Requirements in a Use Case-Driven Approach: A Static Analysis Technique based on Graph Transformation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 105–115, Orlando, Florida, 2002. ACM Press.

[Hol97] Gerard J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[IEE84] IEEE Guide to Software Requirements Specifications. ANSI/IEEE Standard 830-1984. Institute of Electrical and Electronics Engineers, 1984.

[Ins90] The Institute of Electrical and Electronics Engineers. *IEEE Std 610.12 Standard Glossary of Software Engineering Terminology*, 1990.

[ITU96] ITU. Recommendation Z.120. Message Sequence Charts (MSC'96). ITU Telecommunication Standardisation Sector, Geneva, 1996.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1st edition, June 1992.

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[Jon90] C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[JS90] C. Jones and R. C. Shaw. *Case Studies in Systematic Software Development*. Prentice-Hall, 1990.

[Kin94] E. Kindler. Safety and Liveness Properties: A Survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

[KNP02] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002.

[KNP04] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, September 2004.

[Kri63] S. A. Kripke. Semantic Analysis of Modal Logic. In *I: Normal propositional calculi. Zeitsc. Math. Logik Grund. Math.*, pages 67–96, 1963.

[Kru01] P. Kruchten. *The Rational Unified Process an Introduction*. Addison-Wesley, 2nd edition, June 2001.

[KS98] G. Kotonya and I. Sommerville. *Requirements Engineering, Processes and Techniques*. John Wiley and Sons Ltd., 1998.

[KTK02] Jana Koehler, Giuliano Tirenni, and Santhosh Kumaran. From Business Process Model to Consistent implementation: A Case for Formal Verification Methods. In *Proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC'02)*, September 2002.

[LV01] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In *Proccedings of SPIN2001*, Toronto, May 2001.

[MBJ90] J. Mylopoulos, A. Borgida, and M. Jarke. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, October 1990.

[MC92] K.C. Kang M.G. Christel. Issues in Requirements Elicitation. Technical Report CMU/SEI-92-TR-12, ESC-TR-92-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, September 1992.

[MC01] William E. McUmber and Betty H. C. Cheng. A General Framework for Formalizing UML with Formal Languages. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 433–442, Toronto, Ontario, Canada, 2001. IEEE Computer Society.

[McM93] Ken McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[mda02] The Model-Driven Architecture, Guide Version 1.0.1, omg/2003-06-01. OMG Document, April 2002.

[mof03] Meta Object Facility (MOF) Specification, Guide Version 1.4, formal/02-04-03. OMG Document, June 2003.

[MP04] M. Mach and F. Plil. Addressing State Explosion in Behavior Protocol Verification. In *Proceedings of SNPD'04*, Beijing, China, June 2004.

[Mul79] G.P. Mullery. CORE - A Method for Controlled Requirement Specification. In *Proceedings of the 4th International Conference on Software Engineering*, pages 126–135, Munich, Germany, 1979.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46, Limerick, Ireland, 2000. ACM Press.

[ocl03] UML 2.0 OCL Final Adopted Specification, ptc/03-10-14. OMG Document, 2003.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. 19th Annual Symposium on Foundations on Computer Science, 1977.

[Pri57] A. Prior. *Time and Modality*. Oxford University Press, London, 1957.

[Pri67] A. Prior. *Past, Present and Future*. Oxford University Press, London, 1967.

[QS81] J. P. Quielle and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th International Symposium on Programming. LNCS 137*, pages 337–350, New York, 1981. Springer.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an Extensible and Highly-Modular Software Model Checking Framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, Helsinki, Finland, 2003. ACM Press.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, London, 2nd edition, 1992.

[sta03] What Are Your Requirements? The Standish Group International, 2003.

[SVW87] A. P. Sistla, M. Y. Vardi, and P. Wolper. The Complementation Problem for Buchi Automata with Application to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.

[uml03a] UML 2.0 Infrastructure Final Adopted Specifcation, ptc/03-09-15. OMG Document, 2003.

[uml03b] UML 2.0 Superstructure Final Adopted Specification, ptc/03-08-02. OMG Document, 2003.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[vLDDD91] A. van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy. The KAOS Project: Knowledge Acquisition in Automated Specification of Software. In *Proceedings AAAI Spring Symposium Series*, pages 59–62. Stanford University, March 1991.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1):1–37, November 1994.