uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Divya Karunakaran Nair

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

A Logic Based Approach to Use Case based Requirement Verification and Domain Model Improvement

TITRE DE LA THÈSE / TITLE OF THESIS

Stephanie Somé

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

A. Felty

Y. Labiche

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

A Logic based Approach to Use Case based Requirement Verification and Domain Model Improvement

By

Divya K Nair

A thesis submitted to
The Faculty of Graduate Studies and Research
In Partial Fulfillment of requirements of the degree
Master of Computer Science (M.C.S)

School of Information Technology, SITE
Department of Computer Science
University of Ottawa
Ottawa, Ontario K1N 6N5
Canada

April 2007

Your file Votre référence
ISBN: 978-0-494-32471-4
Our file Notre référence
ISBN: 978-0-494-32471-4

The undersigned hereby recommend to

The Faculty of Graduate Studies and Research

The acceptance of the thesis

A Logic based Approach to Use Case based Requirement Verification and Domain Model Improvement

Submitted by

Divya K Nair

In partial fulfillment of the requirements for the degree of

Master of Computer Science (M.C.S)

Professor Stephane S. Some (Thesis Supervisor)

University of Ottawa

April 2007

Table of Contents

ABSTRACT

In software systems, a significant number of software errors and disasters can be traced to late detection of requirements errors. Hence, it is crucial to concentrate on early design phases to verify the primary design against user requirements. In this thesis, two model based verification methods are proposed for performing early design verification and validation. The first approach is a predicate based verification method termed 'Predicate based Sequential Verification (PSV)' which checks the domain model against semi-formal Natural Language (NL) based Use Cases. The PSV module reports requirement violations and model inconsistencies and suggests improvements to the domain model. A proof based strategy termed 'Scenario Sequential Verification (SSV) Strategy' is also discussed as an alternative method for proving domain model requirements which relies on Program Transformation method using Hoare logic. The second verification approach called 'Semi-Automated Validation (SAV)' verifies the formal design model (state chart) against Semi-formal NL based Use Case requirements.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude towards all those who helped me throughout my thesis. First and foremost, I would like to thank my husband Sivan and my parents for their generous time and help. I would like to thank my thesis supervisor, Professor Stephane S Some for his guidance and valuable comments. He has helped me in selecting courses relevant for my thesis and has generously funded for my thesis. He has also been a great help in guiding me with concepts and in locating material for my work. I am also grateful to Professor Amy Felty and Professor Gregor.V.Bochman, who provided constructive feedbacks and suggestions on my thesis concepts. I wish to thank University of Ottawa for presenting me with such a wonderful research opportunity in fulfilling my career. Last but not the least; I thank the Great Almighty for giving me confidence and strength in completing my thesis work.

Thank You All!

Divya K Nair

April 2007

Ottawa, Canada

List of Figures

List of Tables

List of Abbreviations

ATP	:	Automated Theorem Proving
AUS	:	Abstract Usage Scenario
BDD	:	Binary Decision Diagram
CASE	:	Case Oriented Software Engineering
CSP	:	Communicating Sequential Processes
CTL	:	Computational Temporal Logic
DCG	:	Definite Clause Grammar
DDL	:	Domain Definition Language
DL	:	Dynamic Logic
EMC	:	Explicit Model Checking
FODA	:	Feature Oriented Domain Analysis
FOL	:	First Order Logic
LTL	:	Linear Temporal Logic
MBV	:	Model Based Verification
NL	:	Natural Language
OCL	:	Object Constraint Language
PDDL	:	Planning Domain Definition Language
PAS	:	Patient Admittance System
PMS	:	Patient Monitoring System
PSV	:	Predicate based Sequential Verification
RE	:	Requirements Engineering
RVS	:	Rule based Verification of Scenarios
SAT	:	Propositional Satisfiability Checker
SAV	:	Semi-Automated Validation
SCL	:	State chart based Logical Statements
SMC	:	Symbolic Model Checking
SMV	:	State Machine Verification
SSV	:	Scenario based Sequential Verification
SUM	:	Synthesized Usage Model

UCed	:	Use Case based Requirements Engineering
UCDA	:	Use Case Driven Analysis
UCL	:	Use Case based Logical Statements
UCS	:	Use Case Specification
UCM	:	Use Case Maps
UCD	:	Use Case Diagram
UML	:	Unified Modeling Language
UORE	:	Usage Oriented Requirements Engineering
V&V	:	Verification and Validation
XMI	:	Extended Meta-Data Intechange

Chapter 1. Introduction

This thesis is focused on performing design verifications in early stages of software development. Two logic based model verification approaches are proposed to verify the primary design models (domain model and formal design model) against semi-formal Natural Language based requirements. The type of semi-formal requirements considered for the proposed approaches are functional user requirements expressed in the form of semi-formal Natural Language based Use Cases.

1.1 The Problem

Requirements are functional and non functional specifications, operating under specific conditions required to be fabricated in a system under development.

Developing a 'correct' software product obeying the exact user specifications has become increasingly challenging in the last few years. As the number of requirements grows, the degree of complexity involved in understanding the requirements and the techniques used for pursuing them broadens. Some of the main reasons that contribute to the increasing complexity of requirements are:

- It is highly impractical for a single person or a few to imagine all sets of possible situations and scenarios pertaining to a system.
- There is usually an enormous gap experienced between the original informal view of the system and the formal design aspects for implementing the system.

If the entire life cycle of software development is analyzed, it can be seen that the requirements are referred to at all stages. This is usually caused by the lack of clear understanding of requirements before commencing the design or due to non-conformance of the design model to requirements. Therefore, the requirements need to be systematically analyzed for the accurate understanding of problem. This proper understanding of requirements eventually aids in developing a comparatively concrete and sound domain model, which guarantees a good consistent design. For the successful launch of a software product, it is crucial to make sure that the requirements are clearly studied and thoroughly verified against the design.

It is necessary to bring an insight into requirements engineering since it analyses all the

requirement related activities. Requirements Engineering (RE) is the major branch in software engineering that decides the accurate functional behavior of the software product. RE deals with discovering, analyzing, documenting and maintaining a set of requirements for a system [35], [77].

Karl Wiegers defines Requirements Engineering as a non-technical activity of understanding what exactly we plan to build prior to building it [49]. But, this is more of an abstract definition and does not specify the processes involved in it. Ian Sommerville describes Requirements Engineering as "*a specification of what is to be implemented*" [41]. According to Zave [98], Requirements Engineering is the branch of software engineering concerned with the real world goals for, functions of, and constraints on software systems. It is also concerned with the relationships of these factors to precise specifications of software behavior and to their evolution over time and across software families.
This is one of the best definitions for Requirements Engineering since it provides basis for verifying the properties and constraints for software systems. It also throws light on the changing trend in specifications of software behavior.

The RE process usually covers a broad spectrum of activities mainly comprising of requirements elicitation, software requirements analysis and specifications, requirements documentation, requirements validation and verification and requirements change management. These RE activities commence even before the earlier phases of software development like preliminary planning and the survey phases and continue with design, implementation, testing and maintenance phases.

Requirements elicitation is focused on obtaining a clear understanding of the problem domain and extracting the '*correct*' requirements in an efficient fashion. The result is a set of customer requirement specifications understandable by the users of the system. Requirement analysis, on the other hand, is oriented towards studying the requirements closely and coming up with a good solution to implement the identified requirements. The requirements management activity keeps track of the changes in requirements and

monitors different techniques to realize these requirements. The above mentioned RE activities work hand-in-hand to derive a good requirement specification. The following Figure 1 briefs the RE activities. The informal Requirements Specification (denoted by dotted-lined connection to RE activities) is the input to the RE activities which includes the extraction of informal requirements from the specification in a structured format, analysis of the extracted requirements, verification of the requirements and finally the management of requirements.

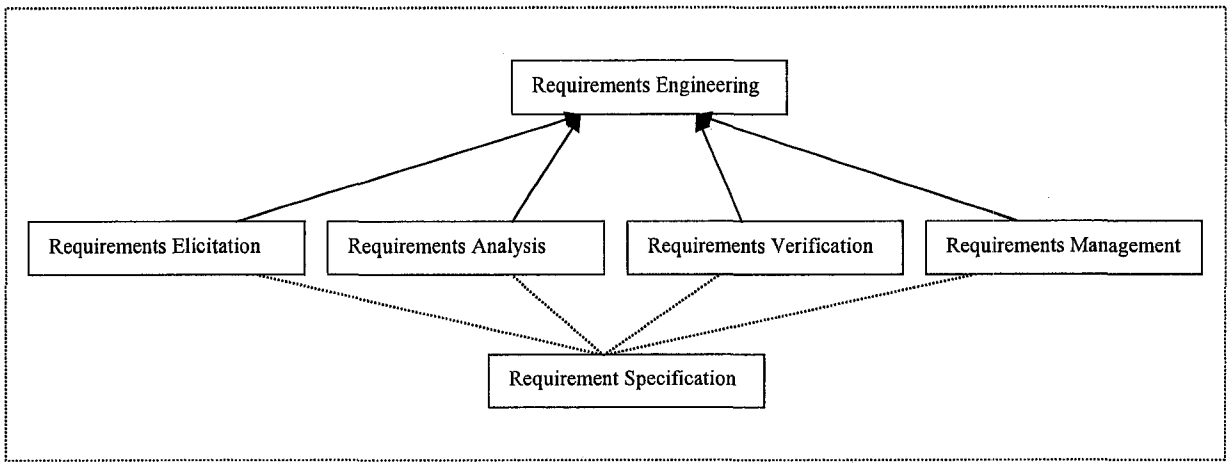


Figure 1: Requirements Engineering process

Once a concrete set of requirements is created, it is crucial to make sure that the stated requirements are devoid of any inconsistencies. This can be achieved by assigning a good structure to represent these requirements such as the Use Case representation [44]. Once the requirements have obtained a uniform structure, the conceptual or the domain model [48] can be built from these Use Cases. Some inconsistencies or scenario violations can still be caused by any false assumptions or bad strategies used for constructing the Use Case and domain model. A domain model is the candidate for initial design in most of the systems. For any software development project, the first step in design, after structuring the informal requirements, is the creation of a domain model from the informal requirements. A domain model provides an abstraction of the system under development. A domain model serves as a dictionary to the domain (problem area) and defines the functions, objects, operations and relationships between objects participating in the

domain. The domain model is a conceptual view of the system which links the informal requirements to the formal design through various levels of abstractions [64]. The creation of the domain model happens at the end of the requirement analysis phase and at the start of the design phase where the informal requirements get linked to the structural design. In most development processes, the domain model refers to a list of class diagrams where all these diagrams are considered as approximations of the domain at different degrees of abstraction. The quality of a domain model is decided by the degree of understandability regarding the entities, vocabulary and relations between entities, conveyed to the stakeholders.

In addition to information from the Use Case requirements, a certain degree of data collection, analysis and evaluations is needed for creating a domain model. The source of this additional information for building the domain model comes from various perspectives and interpretations from users associated with the system under development. Therefore, the information can be fuzzy and incomplete and results in errors and inconsistencies in the domain model construction. Also, the Use Cases have an informal Natural Language form which can contribute to ambiguity and misinterpretations regarding requirements by the designers and developers. Since, the domain model is the first structural model constructed for the whole system, there is a high degree of abstraction imminent in the model. In spite of all these factors, for a realistic domain model with a certain degree of high level details, the presence of errors and inconsistencies cannot be denied. For creating the domain model, the developer adds some essential details for obtaining technical feasibility based on his own assumptions and guesses.

All these above mentioned elements call for validating the domain model against the user requirements. In the case of safety and security critical systems, the verification of domain model against the basic functional requirements cannot be avoided under any circumstances. Moreover, even for a non safety critical system, efficiency and reliability need to be ensured. In most of the systems, the requirement level tasks confirm the verification of informal Use Case and the domain level tasks confirms whether the domain model satisfies the requirements represented by Use Case. So, the requirement level and

domain level verifications behold the development of a successful end product.

The recent trend from various studies in software development shows that very little importance is given to requirement verification activity and primary model checking and that requirement errors affect the safety of systems more than errors in any other development stages [65], [74]. Most of the projects especially the industrial projects, give focus to verification and testing either when they are through with the formal design phase or during the implementation. This happens due to the result of the high cost factor involved in validating and verifying the requirement specifications [15]. According to a study conducted by Linda Rosenberg of NASA [53], the graph for the cost component in fixing errors for various phases is shown in Figure 2.

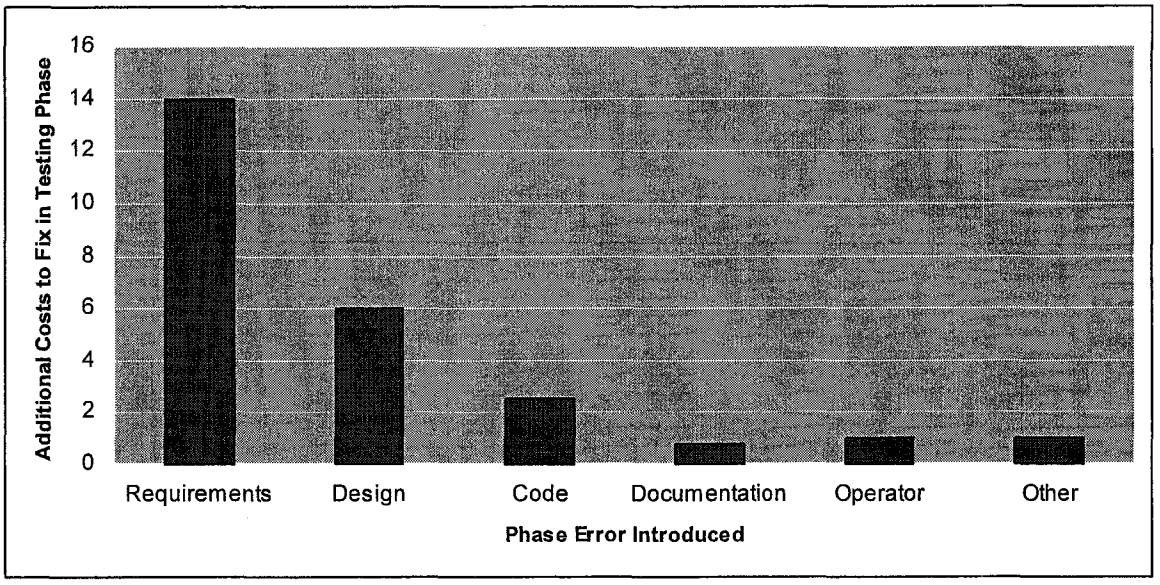


Figure 2: Degree of Errors in different Development phases

Requirement Verification and Validation (V&V) has even more importance for safety critical systems since a large degree of software errors occurs at requirements analysis and early design stages [14], [1], [7], [21]. Because the involvement of requirements lasts till the end of the software development cycle, it is absolutely critical to verify and validate the requirements as early as possible for an efficient implementation. Verification is essential for checking the properties, constraints and operations in requirements. It is natural that

there will be inconsistencies arising when the requirements are subjected to perform under various scenarios. These inconsistencies remain undetermined until the design proceeds in the advanced development stages. If this situation holds true, then eradicating these errors can occur only at the expense of time and money. Hence, it is vital to make certain that the constructed domain model is not straying from the original user requirements and the Use Case scenarios do not bring forth any violations. This whole perspective recalls the necessity for early requirement and design phase verifications. This kind of verification is inevitable for software systems because the requirement violations can eventually trigger some serious disasters.

Verification and validation processes are supposed to take place in all phases of software development. But the verification process should be given significance while dealing with the requirements in the requirement analysis and early design phases because these phases decide and control the advanced development stages [15]. In the present realm of requirements verification and validation, there exist a lot of techniques for verifying the Use Case requirements and the primary design models derived from the requirements. The two most important techniques are the application of formal methods like model checking and theorem proving [87]. Apart from these techniques, there are models, which suggest some important rules to be observed while developing a software product. For instance, the 'Software Verification & Validation' model [66] aids in providing some rules to confirm that the software product development goes in the right direction and that the quality of a software product is assured. But the problem with these techniques and models is that they either exceed the cost limit incurred for the development of the system [15] or highly abstract to be implemented [52]. Also, most of these methods lack either early verification of the preliminary design domain model against the user requirements or if in case there exists such a system for early verification, the system is known to consume much time and effort. So, in light of these obstacles, two model based approaches are proposed in this thesis to verify the primary design models against the semi-formal user requirements.

1.2 Overview of the Proposed Solution

The main objective of this research is to perform formal software requirement and design verifications at early design stages. The difficulties in the existing formal approaches have

motivated this research to come up with two model based approaches for performing primary design verification and validation. The significance of the proposed approaches lies in performing early design verification and detecting the requirement errors and inconsistencies present in the primary design models. The first verification method verifies the domain model for requirement errors and inconsistencies and suggests methods to further improve the domain model. The second verification method verifies the formal design model (state chart) against user requirements. Since Use Cases [26] are regarded to be one of the effective ways to model requirements, the Use Case representation is used to express informal requirements in both model based approaches. Use Cases are easily traceable to the later development stages namely implementation and testing. A Use Case describes a set of sequential interactions between the actors and the system. A Use Case scenario represents a sequential flow of events involving actors entitled to perform a specific function. In order to automate the proposed verification methods, a semi-formal Natural Language Use Case representation based on contextual grammar is adopted for expressing informal requirements.

The first model verification method is termed *Predicate based Sequential Verification (PSV)* that checks whether the domain model satisfies the Use Case requirements. The domain model representation used in *PSV* method is an extended version of the UML domain model. The domain operations are represented using preconditions and postconditions based on the idea of 'contracts' [11]. The *PSV* verification method verifies whether the domain operations correctly imply the Use Case requirements. This is performed by verifying the postconditions of domain operations against the *expected* postconditions of Use Case scenarios extracted from the semi-formal Natural Language (NL) Use Cases. The result of *PSV* verification reveals requirement violations and inconsistencies present in the domain model. A *PSV* report is generated to provide suggestions on improving the domain model in case of requirement errors or model inconsistencies. An alternative proof based strategy termed *Scenario Sequential Verification Strategy (SSV)* which relies on Program Transformation [33] method is proposed as a future enhancement for proving domain model requirements. The *PSV* method ensures an improved domain model, on which the future design stages can

completely rely on.

The second verification method termed *Semi-Automated Validation* is focused on verifying formal design model against Use Case requirements. The formal design model used for *Semi-Automated Validation* is a modified documented version of UML state chart diagram [37] because of its expressive nature. Each state in the state chart diagram is represented by a characteristic set of predicates and transitions [95]. The output of *Semi-Automated Validation* discloses any requirement errors present in the state chart diagram.

1.3 Thesis Contributions

The main contributions of the proposed approaches in this thesis can be summarized as:

- A translation method for converting semi-formal NL Use Case requirements to logical predicates, thus reducing the gap between informal and formal user requirements
- Early Domain model Verification against the semi-formal NL Use Case scenarios using an automated approach and reporting requirement errors or model inconsistencies present in the domain model and suggesting improvements to the domain model
- Detecting the invariants for the Use Case scenarios and verifying that the domain model meets the invariants
- Verification of formal design model (state chart diagram) against Use Case requirements by a semi-automated logic based approach
- A Hoare logic based approach to verify domain model against semi-formal NL Use Case scenarios; this is proposed as an alternative method for domain model verification as a future enhancement

1.4 Outline of thesis chapters

Chapter 2 presents an analysis of various previous and ongoing research works and projects relating to Requirements Engineering and requirements verification and validation approaches. The merits and demerits of the various approaches are compared and studied and an analysis on all the approaches is finally presented. There are plenty of existing

works, which focus on modeling Use Cases, and specifying and verifying requirements. These works are mainly oriented towards the Requirements Engineering activities especially requirements verification for software systems. But the main drawback of such approaches is that they are technically less practical due to exceeding complexity and processing duration.

Chapters 3 throws light on motivations and background for the proposed approaches. It also describes the basic definitions, concepts and conventions pertaining to the proposed approaches.

Chapter 4 is aimed at describing the *Predicate based sequential verification (PSV)* approach. The approach verifies that the postconditions from the semi formal Use Case scenario are satisfied by the operational postconditions of the domain model. The input to this sequential requirements verification module is the *expected* pre and postconditions of Use Case scenarios, the Use Case scenarios, and the domain model. The output is a *PSV* verification report on the domain model. The Chapter also deals with the proof based approach, *Scenario Sequential Verification Strategy (SSV)* which is an alternative method for domain model verification.

Chapter 5 talks about the *Semi-Automated Validation* approach for verifying formal design model, the state chart diagram. The approach is used for detecting requirement errors in the state diagram. This verification assures that requirements modeling proceeds in the right direction in the context of iterative development. The approach consists of extracting logical statements from Use Case and verifying these logical statements with corresponding logical statements obtained from the state chart. Some requirement violations symptomatic of wrong design assumptions were detected using the *Semi-Automated Validation* method.

Chapter 6 proposes a case study to demonstrate the PSV process and Semi-Automated Validation methods.

Chapter 7 concludes the approaches and discusses benefits of the *PSV* and *Semi-Automated Validation* approaches, future enhancements to these approaches and some of the open issues pertaining to the approaches.

Chapter 2. Related Works

There has been considerable amount of work done and currently in progress on requirements and design verifications. The basis of these verification techniques is to explore methods to connect informal Natural Language requirements to formal design format to proceed smoothly into implementation. The key factors responsible for bridging the gap between the informal and formal requirements are:

- Using Natural Language based definition of requirements to derive a complete informal requirement specification.
- Defining an appropriate structure for expressing the informal requirement specifications in different scenarios.
- Practical strategies for converting the structured informal requirements to logical expressions in a formal specification language format.
- An unambiguous definitive specification language.
- A formal design which can represent the whole system exactly.
- Precise and Pragmatic Verification and Validation (V&V) techniques which confirms and proves the formal design to satisfy users' requirements accurately.

In this chapter, the past and ongoing research relating to the above mentioned factors are analyzed and studied and existing works on requirements and design verifications methods are discussed. Existing research works on logic based conversion of informal requirements based on theorem proving are also presented.

2.1. Giese & Heldal Approach

In this Giese-Heldal approach [57], a way of bridging informal and formal specifications is discussed; a method which tries to relate informal requirements to more formal specifications. The informal representation takes the form of Use Cases, while the formal representation uses the state chart design. The approach conveyed that formal specification can improve the informal understanding of a system by exposing gaps and ambiguities present in the informal specification.

To represent the informal textual requirements, most of the approaches use Use Case templates and these templates usually include pre and postconditions. The relationship between informal textual pre and postconditions of Use Cases and the formal OCL pre and post conditions of operations in the class diagram are analyzed in this approach.

There are two ways suggested in the approach to include contextual knowledge in the formalization process:

1. Improvising the representation in Use Cases
2. 'Acknowledging' contextual knowledge exactly.

There are often two kinds of constraints involved with formalization:

1. Pertaining to users requirements
2. Pertaining to the internal structure of the system

The authors of this approach are interested in the first kind ("Pertaining to users requirements") as it is purely independent on design. The authors rely on Use Case diagrams, class diagrams and state chart diagrams to explain the approach.

2.1.1 Deriving operational postconditions from Use Cases

The approach assumes that a Use Case (UC) comes with a precondition and a post condition, Post-uc, of the Use Case. The precondition is rarely mentioned in the approach because of the idea that a postcondition is supposed to be ensured only if the precondition is satisfied before the Use Case.

The approach is illustrated by an example:

In the context of an ATM application, *'Having the customer entered the correct PIN'* as a precondition:

It is not specified what happens if the customer enters a wrong PIN."

This means that the implication of precondition is actually implicit since without satisfying the precondition, the postcondition cannot occur. To verify whether the post conditions are met, the post conditions of the operations from the state chart are compared against the postcondition of the Use Case. The combination of the operations will be different depending on the user's interest in scenarios. The approach makes sure that irrespective of the order of operations taken by the user, the corresponding postcondition

of the last operation in the sequence should be fulfilled. To make matching of post conditions possible, the post conditions are required to have a representation easier for verification. This representation adopted by the author is a form of OCL representation.

2.1.2 State Chart Verification

The problem is stated as:

"Given a state chart, let the set, $\sum$ represent all paths ,л, from initial state to final state, then for each path, л with events, op_1 ($args_1$), ..., op_k ($args_k$), let final(л) := op_k be the last operation called, then the postcondition of this last operation should imply postcondition of the Use Case , that is :

$$Post_{final(л)} \rightarrow Post_{UC}$$

If Cond (л) represents the conjunctions of all the guards or conditions encountered on the path, л, then for all the paths, л $\in \sum$, the following should be implied:

$$Cond\ (л) \rightarrow (Post_{final(л)} \rightarrow Post_{UC})$$

(Or)

$$(Cond\ (л) \wedge Post_{final(л)}) \rightarrow Post_{UC}$$

Analysis:

The **Giese & Heldal** approach is aimed at bridging the gap between informal and formal requirements by a validation approach. The informal requirements were expressed in Natural Language text and they are related to the formal operations written in OCL. For the formal design, the authors tried to come up with a state chart derived from the class diagram, hence it was a state chart based validation approach to link formal and informal requirements. The relation is expressed by checking the paths of the state chart against the various flow of control in Use Cases. The approach was well explained by an ATM example. The informal requirements were expressed purely in Natural Language text without adding any formal details or using templates, hence the approach is easily understandable from the perspective of the user and the constraints can be negotiated by the users. The approach illustrates that the functional behavior of the informal requirements is captured by the formal design. The concept of preconditions and postconditions aids in getting a better picture of the Use Case descriptions.

There are many paths, in fact, infinite paths connected to a state chart diagram. But the

approach concentrates mainly on the paths that happen to exhibit 'happy' scenarios and 'most frequently' used scenarios. In the case of a security based application like ATM, the occurrence of exceptional or abnormal situations cannot be neglected. Although it is not possible to track all the paths, at least a subset of these paths has to be taken into consideration. Also, there can be a number of state charts pertaining to a problem domain, the approach does not consider a method for integrating these entire state charts and so the functional consistency of the whole system cannot be ensured. The approach lacks essential technical detail of how the transformation to OCL operations is performed as well as the technical feasibility for implementing the formalization strategy. All the operations in the model are not represented by OCL specifications, this is because the approach emphasizes only on the constraints directly relating to customer requirements and neglects the internal structure of the system where all the operations are considered. The expression '@pre' is denoted in the approach to hold the previous status concerning operations, but this status is hardly defined and there is no way to check the correctness of the expression. The flow of controls has to be manually derived by the developers and this solely depends on the interests of the designer. During the checking of paths in state charts, certain OCL expressions appear in addition to the conditions expressed in the informal requirements; the reason for this aspect is not investigated and the impact of these additions are not analyzed.

2.2 The SUM Approach

The SUM [13] research introduces a model driven approach for capturing the functional requirements and performs a lightweight verification on the model. The objective of the approach is to come up with a model known as Synthesized Usage Model (SUM) based on the idea of the Use Case Driven Analysis (UCDA), an extension to Usage-Oriented Requirements Engineering (UORE) [13]. Semantic issues are discussed and notations based on Natural Language are provided. The results provide support for how to successfully apply Requirements Engineering with Use Cases as an important basis for software development.

According to [13], the process of producing a formal specification from the informal

requirements description is too difficult and results in unexpected outcomes if not handled with utmost care. The in depth knowledge of the expectations of the end users about the system responses and an unambiguous informal requirement descriptions governs this conversion process. The authors of this approach tried to tackle these issues by a structured and semantic approach using UORE. UORE is an extension from UCDA. The approach introduces a new phase called 'synthesis phase' to UCDA. This phase is aimed at combining various individual Use Cases to a single model called 'Synthesized Usage Model'(SUM). A graphical representation of Use Case is used with abstraction mechanisms defined for user and system actions.

2.2.1 UCDA

The key elements of UCDA are actors and Use Cases and the Use Cases are described using Natural Language. There is a Use Case Model (UCM) which represents the Use Cases and actors. But the interactions and the relationships among the Use Cases are not defined properly in the UCM; otherwise the UCM is 'a loose collection of Use Cases'. Then an Analysis Model is derived from UCM which describes the whole system's structural behavior. The UCM lacks the ability to automatically generate any test cases and hence fails as a candidate to verification. Also, in UCM, the semantics are poorly defined and Use Cases are simultaneously represented as classes with inheritance relations and as series of events. Since different actors and different combinations of actions are involved in all Use Cases, there will be much room for inconsistencies among the Use Cases unless they are verified by some mechanism.

2.2.2 UORE

The defects of UCM compelled the author to launch a new phase in UORE process called the 'synthesis phase'. The synthesis phase is aimed at collecting all the Use Cases and formalizes them to derive a Synthesized Usage Model (SUM).

The SUM is known to have the following elements:

- Categories of system users and their objectives
- Domain objects, their attributes and operations
- Stimuli and responses of user-system communications

- User and system actions, their possible combinations and usage contexts
- Scenarios of system usage, their flow of events and trigger contradictions

UORE has two important phases

1. Analysis phase
2. Synthesis phase

2.2.2.1 Analysis phase

The result of the analysis phase is a Use Case model containing the descriptions of actors and Use Cases. It consists of two major stages namely:

- Identifying Use Cases and actors
- Unification of terminology

Though the analysis phase of UORE is similar to the traditional Object Oriented Software Engineering approach (OOSE [43]), there are a number of differences mainly:

- Structure description of Use Cases
- Changed semantics of actors and Use Cases
- Identification of Use Case contexts
- Strict application of single-actor view

2.2.2.2 Synthesis phase

It consists of three important activities:

- Formalization of Use Cases
- Integration of Use Cases
- Verification

Formalization of Use Cases:

A Use Case Specification (UCS) is obtained for each Use Case from the analysis phase. The result is a formal graphic representation of the all the produced UCS.

Three activities are involved with the formalization phase:

1. Identification of abstract interface objects
2. Identification of atomic operations
3. Creation of UCS for all Use Cases individually

UCS models that possess the temporal relations involving stimuli/responses/states,

representing Use Cases individually are the output.

2.2.2.3 Integration

The output of the integration stage is a Synthesized Usage Model which comprises a group of usage views one for each actor.

Integration consists of the following activities:

- Identification of user and system actions
- Creation of abstract usage scenarios
- Integration of abstract usage scenarios

Every UCS is converted to an Abstract Usage Scenario (AUS) represented by bubbles for user actions, boxes for system actions, arrows for transitions. The construction of the AUS ensures that the synthesis phase is possible even with a large number of Use Cases. A usage view is produced by tracing and combining similar parts of AUS.

2.2.2.4 Verification

The verification stage ensures that the SUM captures all the functional requirements and works well in different scenarios. There are two stages in verification:

- Verification of UCS
- Verification of SUM

Verification of UCS is performed by matching the UCS with the exact Use Case in UCM. Here, it is checked that the informal textual requirements is fulfilled and the entities in the UCS are correctly defined. The next step confirms that all the UCSs are fully contained in SUM. In the process of verification, there is the possibility of identifying new actors and operations which constitutes for the existence of new usage scenarios.

The approach claims that such cases of additional scenarios, if discovered, can be designed by traversing the graphs of usage views. The approach also throws light on deriving test cases automatically for implying the correctness of SUM implementation. It is claimed that SUM emerges to be a system reference model for various empirical case studies.

Analysis:

The **SUM approach** presents a conceptual traditional approach to model Use Cases and comes up with representations with various degrees of abstraction. Most of the suggestions

proposed in the approach are now applied to the development of formal model like message sequence charts. Since modeling Use Cases lays the foundation for further design stages, the approaches have a greater impact to the software research groups. It introduces Usage Oriented Requirements Engineering as an extension to Use Case Driven Analysis. The concept succeeded in launching a requirements model termed 'Synthesized Usage Model' accommodating most of the functional behavior and imparting a reusable perspective to the system. The approach makes it clear that no existing modeling techniques are complete to imitate the informal requirements as exact. Also, some important constructs are discussed which is now used by many projects and a more practical approach is explained for formalizing Use Cases. Since the Use Cases are tried to be made formal while preserving all the functional details, it can provide a solid basis for performing further verifications easily in the advanced stages. The approach is more concrete and reliable because it is built on an accepted strategy, UCDA. The model produced by SUM sweeps the requirements completely and hardly holds any design aspects and therefore the level of abstraction is maintained. Also, the approach introduces a methodology for deriving automatic test cases from the model so that a low level boundary testing is made feasible at the early requirement stage. Altogether, the model perfectly serves as a reference model for validation and verification.

2.3 Rule-based Verification of Scenarios with Preconditions and Postconditions

This approach [89] proposes a strategy for verifying the correctness of scenarios. A new language for describing scenarios is defined by dramatizing simple action traces. A correctness-verification method is introduced which identifies errors using some rules based on the language in the scenarios and secondly, a retrieval method from rule DB to retrieve the scenarios using preconditions and postconditions.

According to the approach, the occurrence of errors in scenarios is due to the following factors:

- Vague representations - type 1
- Lack of necessary events - type 2
- Extra events - type 3
- Wrong sequence among events - type 4

A new scenario language is developed to describe scenarios which add the simple sequences of actions to 'typed frames' on the basis of a simple case grammar of actions [67], [68]. The language is more of a controlled one based on rules and the scenarios can be easily converted to internal representations. It is claimed that, during the conversion, errors pertaining to illegal use of noun types and omitted constraints can be easily determined. According to the nature of the language, it is expected that the 'type 1' errors can be eliminated. So, the approach focuses on 2, 3 and 4 error types occurring in scenarios. It uses rules to catch errors and at the same time uses a rule DB to shrink the rigorous usage of rules. This is carried out by an automatic selection method for rules which can be applied to scenarios described using preconditions and postconditions.

For creating the basis for scenario language, it is assumed that each action or event has only a single verb associated with it and the verb has its own case structure. These verbs and structures rely on specific problem domains, at the same time; the part played by these cases (roles) is independent of the domain. The concept of Requirement Frames adopted by [69] is used to describe the verbs and their case structures. Every event can be converted to the internal representation by using the frame and concrete words are depicted by pronouns.

Four types of time sequences are assigned for events namely:

- Sequence
- Selection
- Iteration
- Parallelism

A method is proposed by the authors to verify the scenarios depending on rules. Many different rules are applied to a single scenario.

A rule consists of:

1. Description of event
2. Frequency of occurrence of the event

During verification of scenario, the events in a rule are compared against the respective events in scenario and the occurrences of the sequence of these events are recoded. If the events of the rule and scenario have the same internal representation, these events

correspond to each other termed as the 'corresponding relation'. For verification to succeed, the ratio between the corresponding relation of the rule's and the scenario's events should be 1:1. So, a single rule can respond to the course of actions in scenario leading to abstract representation in scenario. In this case, the events of the scenario can be broken down into a number of concrete representations. Hence, the author considers the corresponding relation to be modified to 1: many.

The two types of abstract events in rules are:

- Omitted indispensable events
- Having expressions such as "Something","Someone","Same thing", "Same one" etc present in the event

In the first case, such events will be converted to concrete events by substituting concrete nouns for omitted cases, when the respective events in the scenario are found.

In the second type, "same thing" fits the same noun with "something" that appears in the same rule and hence they are substituted by concrete events corresponding to "something/same thing".

There are mainly three types of rules:

1. Scenario-specific rules
2. Domain-specific rules
3. Domain-independent and generic rules

Scenario-specific rules are true only for certain scenarios, domain-specific rules are true for all scenarios in the same problem domain and domain-independent and generic rules are true for a set of problem domains. There exist a lot of rules for a domain and therefore the right rules should be selected to the right scenario for successful verification. This is achieved by a rule DB using preconditions and postconditions.

Verification using rules is comprised of two phases:

1. Selecting rules from DB
2. Analysis of rules

The events in a scenario are identified with the corresponding events in the rule and the time sequence of the scenario events is checked by algorithmic methods.

By this verification approach, the author claims that scenario and rules can be transformed into the internal representation so that scenarios can be verified with rules and the

correctness of one particular observed scenario can be evaluated.

Analysis:

The **Rule based verification** approach aimed at performing requirements verification of scenarios by a rule based approach. A new scenario language is used for describing the course of actions expressed in scenarios. A correctness verification method was described based on rules to verify the scenarios and a 'rule DB' is launched to retrieve and select the desired rules for scenarios based on preconditions and postconditions. The concept of Requirements Frames is adopted to obtain the structure for the scenario based language. The usage of requirement frames aids in verifying the requirements specifications from a developer's perspective. The requirement frames concept imparts a relational database aspect to requirements specifications. The approach was able to resolve missing requirements and detect wrong sequence of actions in the scenarios. The rule is supposed to have description of events and the frequency of the occurrence for the events. The verification succeeds only if the events in the scenarios match with the corresponding events in the rules. Since there are number of rules, a rule DB is maintained for the rules, the author suggests an automatic rule selection methodology from the rule DB. The application of rules to verify scenarios restricts the usage to particular domains, in case, if the rules are able to hold events of higher abstraction, then the rules can be extended to verify scenarios in other domains too. Also, the maintenance of the rule DB proves to provide a concern to the economic feasibility while concerning the industrial systems and at the same time, in the case of safety critical systems, it is required to maintain a large set of accurate rules. There is very little explanation on the search strategy for searching the rules that correspond to the scenario and also the technical aspect on matching events is hardly mentioned. Though the verification strategy succeeds in detecting missing requirements and wrong action sequences, the occurrence of additional events which are not pre-defined, are not handled properly.

2.4. The Glinz Scenario Approach

This approach [58] describes the importance of scenarios for improving the quality of requirements and an integration approach is discussed for integrating a set of scenarios.

The approach suggests the necessity of alteration in the basic realm of requirements arena and a 'proper adaptation' methodology for capturing the evolving nature of requirements. So, the author of the approach comes up with a 'requirements quality model' which concentrates on adequacy to be the prime quality followed by consistency, verifiability and modifiability.

According to the author, this new quality paradigm calls for requirements engineering techniques that should obey the following:

- describe requirements such that customers can easily understand and validate them
- allow the systematic construction of partial user specifications
- support the early detection and resolution of ambiguities

A summary of the advantages for using scenarios in representing requirements is discussed which includes the following:

- Scenarios specify the user's viewpoint
- Scenarios represent partial specifications
- Scenarios present ease of understanding
- Allow short feed back cycles between users and developers
- Scenarios provide basis for system test

Though scenario presents the above mentioned advantages, the author finds that most of the scenarios do not embrace consistency. He exemplifies this fact by stating that "treating each scenario as a separate entity can cause inconsistency problems". Also, some requirements are better if not modeled by scenarios like the state-dependant behavior which can otherwise be easily specified using state automata and object models. Based on these factors, a new systematic approach is discussed in the approach for integrating a set of scenarios while preserving consistency. The template used by the author for representing the scenarios is the Cockburn's template [25]. Cockburn's template can identify the course of actions without confusion and can separate the normal course of actions form the exceptional scenarios. The Glinz representation differs from the Cockburn's template because the author tried to integrate some elements of the state chart with the Natural Language text so as to impart structural completeness to the specification.

Structuring needs to exhibit relations like "scenario A must be followed by scenario B" or "at this point, either scenario A or scenario B can be executed". It is assumed that each scenario has a starting point and one ending point and so does the state chart.

According to the approach, the library system has three abstract scenarios:

- user uses the library
- librarian works in the library
- exit gate which suggests that no book leaves the library that has not been checked out

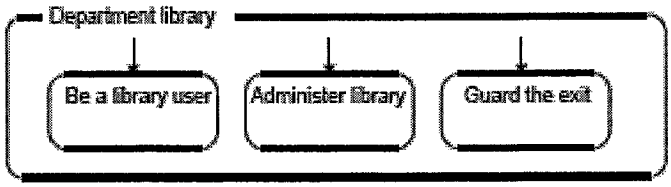


Figure 3: Use Cases for Library system

These are represented as:

If the scenarios for returning books, borrowing books and reserving a book are considered, it can be easily studied that all these scenarios require the process of 'reading' and 'user identification'. Since authenticating user is a common action for all the three scenarios the first step 1 and its alternative can be avoided. This is pictorially represented in Figure 4 as:

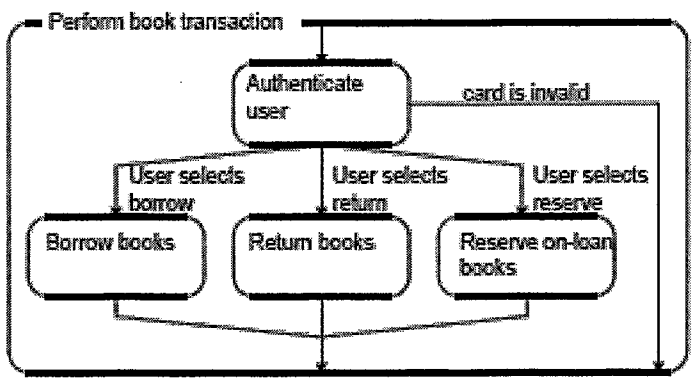


Figure 4: Scenarios for 'Perform book transaction' Use Case

So, the structure that describes all the scenarios can be pictured as:

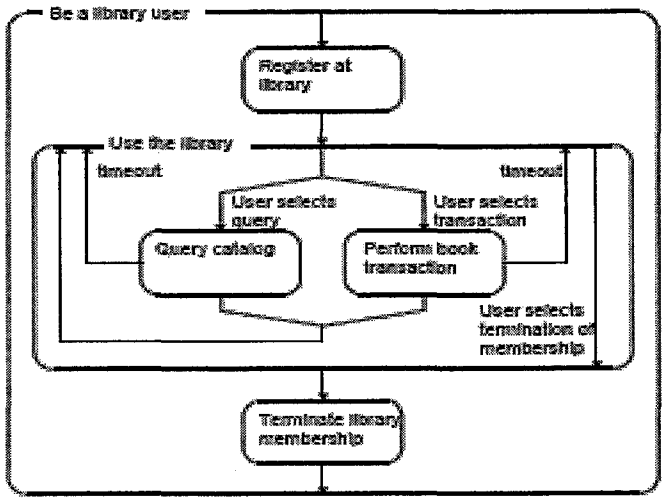


Figure 5: Scenarios for 'Be a library user' Use Case

This state chart based composition of scenarios calls for better perspectives in performing verification.

The following covers the advantages of a state based approach for the basis of verification:

- assess the adequacy of a scenario not only in isolation but also in its context
- verify that the specification expresses required properties of a system properly
- manage partial specifications

Analysis:

The **Glinz Scenario** approach suggests methods for improving the quality of requirements with scenarios and throws light on a strategy for composing series of state charts. The approach of composition calls for a concrete basis for easily performing design verification on the model since the model is supposed to capture most of the functional details. The notion of consistency in requirements is emphasized over other requirement qualities; this is a new step towards obtaining complete requirements because most of the requirement models are aimed at yielding accuracy to the model at the cost of consistency. The concept has some degree of ambiguity and incompleteness because of the lack of a systematic approach. The system embraces quality and the state chart based class diagrams are represented both individually and as combined. During composition of all state charts, the

relationships between various state charts are well analyzed and merged to capture the whole system behavior consistently. The non integration of functional requirements and high level security requirements are given hardly any attention in this approach. Also, the approach denies dealing more with the technical details of the design and hence has to go a long way to work for an accurate design.

2.5 KeY Project

KeY system [94] introduces an add-in to the conventional CASE [4] tool by integrating formal methods into the design specification phase.

- Target Language – Java
- Visual Modeling – UML
- Adding further constraints – OCL [9]
- Parser used for OCL- HuBmann's parser [94]
- CASE tool – TogetherCC [Together Control Center]
- Formal Verification – Axiomatic semantics of Java, subset of Java called "Java Card"
- Formula support - Deduction Component
- Theorem prover – DL theorem prover [94]

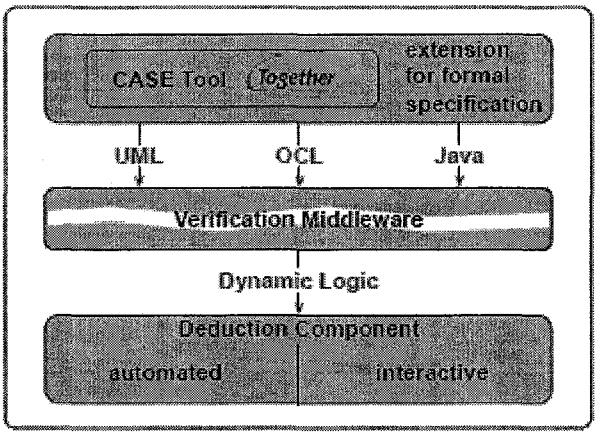


Figure 6: KeY System Architecture

The KeY System architecture (Figure 6) shows the presence of three specific components:

2.5.1 Modeling component

It is comprised of a CASE tool together with the extension of formal verification. This extension serves in creating OCL specifications and in processing these new specifications with already available specifications. Processing the specifications is performed by using external programs, libraries and TogCC's pattern mechanism. Once these specifications are processed, they are sent to the verification component. The frequently used OCL constraints are contained in templates called "KeY Idioms" which can easily be instantiated by the user. The user chooses one specific idiom from a set of KeY idioms which are a library of predefined constraints and instantiate it to the current target model by setting idiom specific parameters. To further improvise the formal design concept, there is the usage of "KeY" patterns in which several design patterns are stored. In the case of KeY pattern, the generated interface objects are annotated with OCL constraints generated from OCL template files and then instantiated using OCL template instantiator. The UML view is then exported from TogCC in XMI format which is followed by the conversion of OCL constraints also in the XMI format. Complying with TogCC's single source philosophy, OCL constraints are stored as comments in the user Java files. An authoring tool proposed in [76] helps in generating specifications; this eliminates the need for mastering OCL for writing specifications.

2.5.2 Verification component

The verification component is aimed at transforming the processed software model, the partial Java implementations and the specifications to Java Dynamic Logic Proof obligations. The verification in the KeY system is called "horizontal verification".

The formal Modeling is based on OCL constraints and therefore helps in specifying the associations among attributes, entities and operations clearly.

For example, in the context of a basic credit card application, the formal specification can be:

Context *c:BasicCreditCard::debit(sum:Integer)*

Pre: *debitPermit(sum)*

Post: *c.account.balance = c.account.balance@pre - sum*

Pre: not *debitPermit(sum)*

Post: *c.account.balance = c.account.balance@pre*

Pre: *true*

Post: *c.bankLine = c.bankLine@pre*

The above illustrated OCL specification is composed of various precondition clauses, so in order to avoid ambiguity, the KeY verification component converts this specification to:

Context *c:BasicCreditCard::debit(sum:Integer)*

Pre: *true*

Post: *c.bankLine = c.bankLine@pre* ***and***

if *debitPermit@pre(sum)*

then *c.account.balance = c.account.balance@pre - sum*

else *c.account.balance = c.account.balance@pre*

2.5.3 Deduction component

The deduction component will convert the OCL properties to JavaDL formulas. In order to reason about Java partial code, the Java statements are parsed and the data structures are given to the interactive Key prover. The Key prover proves the DL proof obligations obtained from the verification component, making use of rules in Java DL calculus using symbolic program execution.

At first, the specifications are converted to logical format based on specific logic; for each class specified in the model, there is a corresponding particular type in the logic.

For example, if the 'Basic credit card' supports a sub type called "junior card", the property of the junior card is logically represented as:

$$\begin{aligned}\forall c{:}JuniorCard.\quad & (juniorBankLine \geq c.bankLine \wedge \\ & c.account.balance \geq -juniorBankLine \wedge \\ & c.bankLine \geq 0 \\ & \rightarrow \\ & c.account.balance \geq -c.bankLine)\end{aligned}$$

The logical expressions used in KeY are based on Dynamic logic, which is an enhanced

version of the first order predicate logic. The Java DL is the Dynamic logic version compatible for proving Java proof obligations and program constructs. Once the conversion to logical formula is done, then the correctness of these logical formulas is verified using the Java deductive calculus. The deduction is performed using the symbolic program execution and Program Transformation rules.

Analysis:

The **'KeY' project** is intended to take a step more in early design verification for industrial applications. Usually, the industrial software approaches are less concerned towards verifying the model at a phase earlier than the advanced design stage or implementation stage. The 'KeY' project demonstrates that it is possible to perform earlier formal verification practically and it ensures technical feasibility. The formal verification approach in KeY project is based on theorem proving. The formal verification component is provided as an add-in to the commercial CASE tool, TogCC. This add-in helps to even include more constraints to the model at the earlier design phase unlike most other practical approaches. The verification component in KeY takes in the domain model, any additional constraints and the predefined constraints and the soundness of the model is confirmed by logical verification by passing through a deduction mechanism. The logic used in KeY id Java Card Dynamic logic which is a mutation of Dynamic logic derived from predicate logic. This is beneficial for verification since predicate logic is known to express the properties of the system in the most effective way. The symbolic program execution and the Program Transformation strategies provide a solid basis for formal verification. The proof rules are based on traditional rules and heuristics. There is also room for re using fragments of proof for different applications in the same domain. Since the selection of models and constraints are from a pre defined set of patterns and idioms, the verification approach is applicable only to particular set of domains. The functionality of the system can be extended by including more reusable patterns and idioms of higher abstractions so that a wide range of applications can benefit from verification.

2.6 The OCL-First Order Predicate Translation Approach

The OCL-FOL translation [10] discusses a specification technique for converting OCL [9] based specifications to first order logic formulas. The translation of OCL into first order logic is done in three important steps:

- Extracting the signatures from class diagrams
- Extracting formulas from class diagrams
- Translating using the axioms and restrictions

2.6.1 Extracting the signatures from class diagrams

The vocabulary for the class diagram, D is defined by understanding the set of types, functions and relation symbols from diagram.

Vocabulary, $\sum = \sum d$ of S S-first order structures to denote system states.

Syntax of basic types is represented by data signature as

$\sum d = (S_D, \Omega_D)$ S_D a set of data sorts defined for class diagram D.

Ω_D a set of operations defined for class diagram D.

S_D is a superset of {Boolean, string, integer, and real}.

2.6.2 Extracting formulas from class diagrams

Once the vocabulary for the system is defined, then formulas are extracted from the signature of class diagrams. This is done by adding some additional symbols to the existing initial data signature.

$$\sum^* = \sum d \cup \sum tr$$

Formulas extracted are represented as a triplet:

Translation, $Th_D = (\wedge Ax_{ADT} \wedge Ax_D \wedge Constr_D)$

Ax_{ADT} denotes axioms of abstract data type used to represent built-in data types + axioms for abstract data types i.e., set, bag, sequence etc. Ax_D denotes interdependencies among functional and relational symbols extracted from D

$Constr_D$ denotes formulas representing restrictions on system states.

Example:

In the context of a **Patient Admittance System**:

If there are two objects "user" and "card"

User can take the role of nurse or doctor

Card can take the role of *nurseidentity* and *docidentity* accordingly.

The association "owns" between these two entities tells that user owns the identity card.

Then the following axiom, Ax_D, holds:

$\forall$ u: user $\forall$ c: card (c $\in$ nurseidentity (u) $\leftrightarrow$ u $\in$ nurse(c))

$\forall$ u: user $\forall$ c: card (c $\in$ docidentity (u) $\leftrightarrow$ u $\in$ doctor(c))

A user can hold a maximum of one identity card

This gives rise to the $Constr_D$:

$\forall$ u: user (size (identities (u)) $\leq$ 1).

2.6.3 Translating using the axioms and restrictions

The translation of constraints from OCL to FOL has to take into consideration the details of class diagram too. The OCL expressions basically are translated to first order term of appropriate abstract data type and the OCL expressions of Boolean type are converted to first order formulas.

Translation works by structural recursion on expressions. When certain OCL features are translated, new functions or predicate symbols are introduced and the axioms that constrain these new symbols are also followed along with the conventional rules. There are various translation rules devised for translating OCL invariants, built in types, associations, conditional statements, equality to first order logic.

2.6.4 Translating pre and post conditions

In the context of a credit card application:

OCL→Context c: creditcard :: debit (sum: integer)

Pre: debitPermit (sum)

Post: c. acct. bal = c. acct. bal @ pre – sum

Pre: not debitPermit (sum)

Post: c. acct. bal = c. acct. bal @ pre

Translated to:

((debitPermit (sum) → bal (acct) = bal (acct) – sum)) $\wedge$

($\neg$ (debitPermit (sum) → bal (acct) = bal (acct) – sum))

Here the exact proof steps are done sequentially by allowing the precondition and

invariants to execute through a set of program statements. It is for this purpose that the program verification or the dynamic logic approach is essential.

Analysis:

The **OCL-first order translation** approach introduces the concepts for transforming the OCL constraints to predicate logic. This approach is important in the formal verification arena due to a variety of reasons. OCL is a high level specification language which can express the properties of software systems. It inherits the common characteristics that are currently present in most specification languages. The rules and strategies suggested presents a way for converting the properties expressed in OCL to first order logic and hence these properties can be verified using the methods of theorem proving. Optimization and heuristic mechanisms are discussed for the process of theorem proving. This translation process with necessary modifications was used in the KeY project for verifying properties. The translation process was basically defined for the OCL constraints connected with the class diagrams and so there is the existence of some semantical incompleteness. The approach stresses the usability of the formulas and provides alternative translations for model elements. The implementation is made as flexible as possible so that it can be applied to other theorem proving systems.

2.7. Supporting Use Case Based Requirements Engineering - the UCed approach

The UCed approach [83] is oriented towards formalizing Use Cases and specifications which can be executed. A simulation environment is also integrated to the approach, which illustrates the proposed approach. The concept covers most of the requirements engineering activities involving elicitation, clarification, composition and simulation. A restricted form of Natural Language is used for describing the Use Cases. The idea is implemented by a tool called UCed -Use Case Based Requirements Engineering Tool [84]. UCed is a Requirements Engineering Use Case edition tool implemented towards Use Case acquisition, domain model creation, composition of Use Case scenarios using the edited domain model and finally synthesizing state chart diagram from the Use Case scenarios by an algorithm.

A Natural Language notation based on Cockburn's template is used for the informal description of Use Cases. UCed generates a domain model as a part of the process using a domain edition tool. A domain model is created from the syntactical analysis of the Use Case specifications. The domain model is depicted as a high-level class model which captures domain concepts and their associations. An operation is defined by a set of 'Added' and 'Withdrawn' conditions.

Once the Use Cases and domain model are constructed, then the approach generates the formal design, in this case, the state chart diagram. An algorithm is used to generate these hierarchical types of finite state transition machines from the Use Cases.

The generation of state chart from Use Cases is based on the following facts:

- When an operation, op is applied to a state s1, where cond (s1) represents all conditions effective at state s1, then the application of this operation along with the added and withdrawn conditions produce the new state, s2. Withdrawn conditions are those conditions that are eliminated after the operation execution and Added conditions are condition that hold after operation execution.
- Each state is characterized as a set of predicates which are based on the domain model entities. Also, a state, 's1' is a sub-state of a state 's', if its characteristic predicates include those of 's' and any transition arising from a state 's' also applies to all sub-states of 's'.

State chart generation is produced from the Use Cases based on the following elements:

- Operation effects (Added and Withdrawn conditions)
- Relation between states and transitions

Every state in the state chart diagram is represented by a set of conditions based on the added and the withdrawn conditions of operations specified in the domain model.

2.7.1 Verification

The above mentioned approach builds a domain model and uses an algorithm for generating the state chart based on the domain model, therefore errors are expected to occur:

- Among the elements in the domain model
- Between domain model and Use Cases
- Among Use Cases
- Inconsistencies in state chart

The syntactical analysis of the domain model takes care of the redundancy in declaration and that the references made in the condition are valid. Since the approach is based on a timed automaton, the inconsistencies experienced in the 'after' delays and 'before' delays can be detected. Light weight Use Case verification is also done against the domain model at the time of Use Case syntactical analysis.

The verification in UCed approach is based on the following rules:

- each operation in a Use Case must refer to a concept operation in the domain model
- each condition must refer to an entity declared in the domain model and any atomic value must be a possible value of that entity

The algorithm does not permit any inconsistent states detected by an undesirable set of characteristic predicates. The approach depicts postconditions as contractual statements of guarantees after the successful execution of the Use Case and a precondition conveys the set of conditions which should definitely occur for the execution of the operation.

Analysis:

The **UCed** approach proposes a strategy for the requirements engineering processes of requirements elicitation, requirements composition and generation of a formal requirements model from the composed requirements. The representation used for informal requirements is a modification of Cockburns template for Use Cases. The representation using Use Cases helps to express the requirements more clearly and makes the approach more comprehensive. The formal requirements model used is a state chart diagram especially because the state chart diagrams are known to provide a well formed pictorial representation of flow of events in various scenarios. A domain edition tool was used to compose the Use Cases and to create a domain model and an algorithm is used to generate state charts from the Use Cases. The approach succeeds in integrating all the Use Cases to obtain a whole system state chart intended in capturing the system's behavior. The whole

process starting from Use Cases capturing to state chart generation is illustrated by a tool called Use Case based Requirements Engineering tool-UCed and a simulation component was added to the tool to view the whole process. This ensures technical feasibility to the concept and the system becomes more understandable to the end user and developers. The domain model is created based on developer's assumptions and intuitions. Since the whole system is dependent on the domain model, the necessity for verifying the domain model and state chart model is indispensable. Although, the UCed tool provides a verification component for performing a light weight verification of the invariants and provide naming consistency checks for domain model, a domain verification module will tend to make the system more perfect.

2.8 Comparison of Related Works

A comparison of related method with the proposed *Predicate based Sequential Verification* approach (PSV) are briefed in the Table 1 below:

Approach	**Model driven(M) Or Logic based (L)**	**Consistency Low(L) High (H)**	**Ease of Verification Low(L), High(H)**	**Verification of scenarios Present(P), Absent(A)**	**Verification of domain model Present(P), Absent(A), Light-weight(L)**	**Improvements to domain model (suggested(S),not suggested(NS)**
Giese-Heldal	M	L	L	P	A	NS
SUM	M	L	H	A	A	NS
Rule based	M	H	H	P	A	NS
Glinz	M	H	H	A	A	NS
KeY	M,L	H	L	P	A	NS
UCed	M	H	H	A	L	NS
PSV	M,L	H	H	P	P	S

Table 1: Comparison of Related works

All the approaches mentioned above are aimed at describing strategies for attaining a good requirement model free of any requirement contradictions and inconsistencies, to guarantee a sound design capable of an error free implementation.

Chapter 3. Background, Definitions and Conventions used for Proposed Approaches

This chapter provides an introduction to formal methods and formal verification which presents a background for proposed approaches. The definitions and conventions used for the proposed approaches are also discussed in this chapter.

3.1. Formal Methods

Formal methods have a definite impact in the requirements phase; these methods aid in detecting errors, ambiguities and missing requirements and also promote analyzing the working of the system at an earlier time. Formal methods are based on mathematical techniques used for requirement specifications, design and verification of software as well as hardware systems [78]. Formal methods play a remarkable role in the expression of the requirements and model verification for the software systems. According to [62], formal methods aid in modeling complex systems as mathematical entities and also help in verifying the properties of system thoroughly rather than by empirical testing. Formal methods are also useful for writing precise requirement specifications and in performing requirement verification for safety pertaining to the real time safety critical systems. The Ariane5 rocket crash happened immediately within the 40 seconds after its launching on the 4-th of June, 1996. An error occurred while translating a 64-bit floating point number to a signed 16-bit number which in turn caused the rocket and the back up to fail. This error could have been avoided if the important scenarios were studied, analyzed and proven properly by some sort of formal methods [22]. Another good example, which demands the need for formal methods, is the bug found in Intel Pentium's division algorithm which ended up having a huge cost. Simulation can help to detect these errors, but testing all the possible combinations of relations and scenarios become highly impractical while considering the cost of applying them. It is at this juncture that formal methods can be really comforting.

Unlike other specification methods, formal methods impart a logical perspective and emphasize the provability of the system. Although, the research on formal methods has been blooming from the 70s, their adoption in industrial projects has been much slower.

The reasons behind this situation are [23], [97]:

- Many of the formal methods are elaborate and complex
- The software engineers are devoid of the knowledge for type theory, which is the basis of formal methods
- Some suggest that formal methods are really hard to master and in some cases impractical for usage
- Formal notations are too isolated and only a handful of relevant case studies are available

There are three important aspects to formal methods---model centered, property centered and process centered. The model oriented aspect of formal methods is aimed at building a model describing the systems behavior in terms of mathematical entities like functions, sets, lists, and associations. Some of the model centered formal methods include the Z language [46] and the B specification method [27]. The second perspective of formal methods specifies the properties of the system logically by using such as the algebraic methods and the usage of axioms. The algebraic methods conveys the properties of system using the basic elements of algebra and mathematical equations whereas the methods using axioms specify properties using logic systems such as the first order logic, higher order logic, prepositional logic, predicate logic and so on. The final process centered aspect of formal methods is mainly concerned with concurrent/parallel systems. They make use of Hoare communicating sequential processes (CSP) [19] as well as protocol specification languages to bring out the specifications and properties of the system.

According to [62], the application of formal methods is a three-step process:

1. Writing Formal Specification to make the requirements accurate
2. Formal Verification to prove the requirements right and to verify the design
3. Implementation to produce code from the verified formal design

An important benefit of using formal specification is that, if it is possible to convert the set of requirements to a formal specification, it is guaranteed that the set of requirements is

complete otherwise it can be deduced that there are some missing components in the requirements list. The use of formal notations in specifying requirements results in the emergence of good complete requirement specifications for both functional and non-functional requirements. The precise nature of formal requirement specifications gives little room for requirement ambiguities and misinterpretations. Since there are concrete rules for producing a formal specification of requirements, the consistency in the specifications is preserved largely. The validation of the requirements also becomes easier due to the clarity in formal representation.

An impediment in the application of formal methods is an issue with usability [62]. A well defined formal specification can describe a system well with less ambiguity, but a problem in creating such a formal specification concerns the difficulty in translating Natural Language.

On the bright side, formal methods have the ability to hike up safety and sometimes can reduce the cost of some standard safety critical systems. Lately, formal methods are even known to be successfully put to practice in the hardware sector, but the software sector is yet expected to travel a long way to embrace formal methods. Some of the industry verification groups like IBM, Motorola, SUN, Intel etc are trying to apply formal methods to achieve safer and more economically feasible software development results [93].

3.2. An insight into verification

Verification generally refers to a software engineering process oriented towards building a software product right in accordance with the specified requirements. There are two important aspects to verification in the context of software engineering- the first one is requirements verification, which should necessarily be conducted in the early requirement analysis phase and the second type is design verification. These early requirement and design verifications have utmost importance because they decide the fate of advanced software development stages. In most of the industrial applications, verification happens only at the later design stages prior to implementation, but performing the verification only at the advanced design phases can prove to be dangerous especially for software systems. This emphasizes the need for requirement and early design verifications.

The most common approaches used for verifying requirements include conducting reviews, traceability analysis, prototyping, interface analysis and so on [91]. Design verification is inevitable at two important stages of design; one is at the early design stage and the other at the end of the formal design creation stage. Early design verification starts immediately after the requirement analysis phase, that is, with the building of a conceptual or a domain model. Since a little more technical detail is added with the requirements, the designer is compelled to make some assumptions of his own. The correctness of a good primary design model is justified by the fact that the model is coherent to all its users and it clarifies all the associations between the various entities and finally obeys all the necessary functions specified in the requirements document. The final quality of the system is dependant on this preliminary model and so utmost care should be given to choosing an appropriate representation for the domain model. In addition, verification of the model against the basic user requirements is a prerequisite for ensuring model correctness.

There are various types of design verifications:

- Functional verification
- Non-functional (e.g. Performance) verification
- Reliability (Safety) verification
- Security verification

These different types of verifications are interrelated and they share a single task of confirming that the system requirement specifications are met irrespective of any exceptional situations and that the desired functional behavior is preserved in all possible conditions. The basic verification assumptions are that there exist good specification criteria for the system and feasible verification methods.

A lot of Software disasters can be traced to faulty requirements and bad assumptions made concerning requirements or to the incomplete specifications. One of the most important problems encountered in writing good requirements is that most software developers are not much experienced in writing good requirements. In order to gain sufficient expertise in expressing requirements, the developers should be exposed to examples of good as well as

bad requirements documents to identify the differences in them. For the last decade, different checklists and catalogues have been made available to the specification developers for the production of good requirements.

There is a unique difference between a requirement and specification. A requirement is a physical property that should be observed to resolve the problem while a specification consists of a list of accurate requirements and their behaviors. A requirement specification document contains a set of requirements needed to build a successful software product and it is from this set of requirements that the future design decisions are made. Since the whole stream of subsequent design and implementation activities rely on these requirements, the question arises as to how to make a requirements document as complete and consistent as possible. All the major possible scenarios should be investigated and assessed; this gives an opportunity for all the users of the system to present ideas of their own.

The most important basic properties necessary to yield good requirements are listed below [73], [42]:

- *Clarity:* It suggests that the essence of the requirements statement should not mislead the different users of the system. There should be only one exact interpretation possible for the sentence and complex language structures should be avoided as possible.
- *Generality:* This property assures that the design possibilities for implementing the requirement are not closed.
- *Concise:* It ensures the succinctness of the requirement, that is the requirement, should be brief but at the same time withholds clarity, completeness and comprehensibility.
- *Accessibility:* The requirement should be feasible technically and economically.
- *Consistency:* The requirement should not cause any contradictions to the already stated requirements.
- *Easily verifiable:* At the end of the conceptual design and formal design stages, the design framework should be capable of verification against the requirements to ensure that the important properties still hold right.

The approach to writing good requirements should be black box oriented. Although the implementation experts should have knowledge about the requirements, it is always better for the designers and stakeholders to generate a good requirement specification document. Yielding the right requirements reduces the development effort and guarantees safe future for software development.

We cannot neglect the antagonistic nature of some requirements, such as the requirement that some systems should be both secured and reliable. In this case, it is the responsibility of the developer to assign priority among the requirements.

3.3. Formal Verification

The idea of formal verification is to verify that the designed model, whether the primary or the final formal model, conforms to the properties associated with the requirements. A system model is comprised of entities and relationships between the entities and each entity is known to have a set of specified constraints; therefore we can picture the design as a set of mutually participating entities possessing different expected behaviors under respective scenarios. In particular scenarios, the status of entities changes through various states and transitions. This calls for the need to investigate the behavior of system model in each scenario against the expected requirements and their properties. If we are able to identify most of the relevant scenarios and verify the model's expected behavior under these scenarios, the system model passes the test to proceed to the next level of advanced design stage or programming phase. This objective can be guaranteed by formal verification techniques. The two major formal verification techniques are Model Checking [30] and Theorem proving [60]. Since our proposed approaches are logic based, theorem proving and related concepts are described in the next section.

3.3.1 Theorem proving and Related Concepts

Theorem proving is an effective formal verification technique used for verifying design models against formal specifications. In the context of formal verification, theorem proving can be stated as a method for logically or mathematically proving the relationship between the formal specification and the system model by applying some sort of proof calculus.

Verification using theorem proving is done by a series of inference steps and axioms (induction, simplification etc) from which a proof can be derived to achieve the goal. Automated theorem proving works by sub dividing the complex goal into a set of easier sub goals and these sub goals are proved using primitive or classical logic or even heuristics. Theorem proving can impart the highest level of confidence regarding accuracy and therefore is applicable to safety critical, security critical and complex systems. On the negative side, using theorem provers demand clear-cut understanding of requirements and a great deal of technical efficacy. The requirement specifications can also benefit from theorem proving by revealing contradictions prevalent in the specification. The difficulty for using theorem proving is framed in [60] as: *" the very formality of formal logic detracts from its clarity as a tool of communication and understanding, and the "natural" applications of mathematical logic in the pre-digital world were in pure mathematics and there was little interest in the added value of formalization."*

Usually, the theorem proving algorithms rely on the following proof systems:

- *Model elimination*
- *Resolution*
- *Tableaux*
- *Connection technique*

Examples of theorem provers are listed below:

- *HOL* theorem prover [55]
- *Isabelle* theorem prover [72] uses both first order and higher order logic
- *Boyer-Moore* [79] uses first order logic
- *PVS* [82] uses higher order logic

Some software systems depend on proof checkers for verification. Proof checkers possess a set of pre-defined proofs whose correctness have previously been verified; hence we have to indirectly rely on the users for the proofs. On the other hand, automated theorem proving eliminates this issue. The emergence of automated theorem provers (ATP) was one of the greatest and fastest achievements in the field of theorem proving [92]. The input to an automated theorem prover is usually a theorem which is expressed in some logic

depending on the ATP and the output may be a confirmation of the property or elaborate proofs or error reports and counter examples in occurrence of a violation. Some commonly used logics are predicate logic, higher-order logic, propositional logic, but most of the mechanized theorem provers make use of predicate or first order logic. In some cases, the ATP requires a degree of human assistance for proving intermediate goals. A majority of the modern ATP systems use a graphic form of the 'resolution' proof system [96]. *Resolution* theorem proving method [80] is exploited by the automated theorem provers, yet the significant proofs involve, at some stage, definite participation by humans. *Resolution* reveals the importance of interaction between the automated theorem provers and the user [28]. NASA adopted ATP to verify the safety properties of aerospace systems, the verification is performed by generating proof obligations and these proof obligations along with the resultant proofs are sent to an independent proof checker. One of the problems encountered in automating theorem proving is with the quantifier instantiation since the instantiation is often based on assumptions. This problem can be controlled up to a limited extend by integrating efficient proof search strategies and a considerably small amount of human involvement.

The major issues concerned with theorem proving are:

- Choice of logic
- Choice in theorem proving style
- Property type to be proven

The choice of logic used is usually:

- Propositional logic [57]
- First order predicate logic [75]
- Temporal logic [40]
- Higher order logic [90]

Most of the theorem proving systems use the above logics or use combination of the above listed logics. In this thesis, only propositional logic and predicate logic would be

considered.

Propositional logic is based on rules of transformations and axioms to prove theorems; it is otherwise termed as "zero order" logic since it does not deal with predicates. Axioms are propositions or statements that are assumptions which are not deducible, but are taken for granted. Propositional logic expresses the structural relationships of mathematical objects formally. The expressions in propositional logic are composed of indivisible units and therefore relations and functions cannot be expressed. Propositional logic is represented by propositional or sentential operators applied to one or more propositions producing new propositions. The usual propositional operators are:

implication '->', conjunction(/\), disjunction (\/), negation(~) etc.

For example: 'raining results in cold & wetness' is expressed as:

Raining -> cold /\ wetness

First order logic, unlike propositional logic, views the world to be comprised of objects (e.g. people), relations (e.g. brother) and functions (e.g. smaller than). An atomic or a simple sentence featuring a single verb is expressed in first order logic as:

Atomic sentence = *predicate* (*term 1,..., term n*) or *term1 = term2*

Term = *function* (*term1,..., term n*) or *constant* or *variable*

where a predicate is an expression that reveals the truth about something.

Example: *Likes (x, y)*

The important characteristic of first order logic is quantification. First order logic has sufficient expressive power because of quantification. The two basic quantifiers are 'for all (*A*)' and 'there exists (*E*)'.

For example;

Everyone likes someone: $(Ax)\ (Ey)$ `likes(x, y)`

The complex sentences are made from atomic sentences connected by 'connective' operators like conjunction, disjunction etc.

The next issue is related to the style of proving. There are two basic proving styles:

- Bottom up (or) forward style of proving

- Top down (or) backward style of proving

The Bottom up approach commences from the sub goals and work towards the main goal to be proved through rules of inferences. On the other hand, the top down approach starts from the main goal and work towards proving all the sub goals from the main goal, hence it is called 'goal directed' approach.

3.3.2 Linking requirements with logic

The requirements have to be expressed with logic for performing verification. Verification using theorem proving imparts a more logical behavior to informal requirements expressed in Natural Language. Very few techniques are available today for conducting a formal verification using theorem proving prior to the detailed design phase. So, a theorem proving approach is proposed in this thesis to perform early design verification of domain model and Use Case requirements. Before proceeding to the proposed approaches, it is necessary to analyze the different theorem proving strategies for verification.

Basically there are four different strategies of theorem proving used for verifying the system under development namely [23]:

1. Proof oriented:

The proof oriented systems, unlike model checking, do not take into account all the possible states for the system. A proof calculus is selected to derive proofs for proving properties of the system. The proofs check whether the system satisfies the given properties. For example, a proof calculus used to avoid the checking of infinitely different models of a collection of predicate logic formulas in order to establish the validity of a sequent. Proving properties using proof calculus avoids building and checking different models of logical formulas to conform validity of the implication to be proved.

2. Property based:

Instead of verifying the whole system's behavior, specific properties of the system are verified in this strategy. For instance, higher order logic is a property oriented logic which can prove important properties of the system.

3. Sequential Verification:

Unlike property based verification, sequential verification tends to verify the sequential

transformation of a course of actions. These courses of actions can be a set of sequential program code lines or a scenario as long as there is sequential continuity in the fragment. Given a set of input and output specifications, the verification problem is to establish that the set of input specifications being true before the execution of the program segment, the segment terminates and after termination results in the set of output specifications.

For example:

WHILE a! = 0

begin

b = b +2;

a = a-2

end while

Here, given the input specification, *a, b = X, Y* and the output specification, *a=X+Y*, the program is proved as:

The assignment, *a + b = X +Y* is implied first, then it is given to the loop such that this statement holds true every time the loop is executed. Finally, it should be shown that the proposition holds true every time the control goes to the evaluation of the loop.

4. Precondition and Postcondition based:

This is a special case of sequential verification where a termination is guaranteed. For a given precondition, the pre-postcondition verification should ensure that the postcondition should hold after a sequential successful execution of action steps. Here, the verification is supported by a set of rules applied to each step to imply the final postcondition. This can be a program based verification or scenario based verification because for both of these types, the verification starts with precondition working towards the reachability of the desired postcondition through the method of sequential transformation.

For instance in the context of an ATM application:

Function: *Validation*

Precondition: *User card is inserted*

Steps:

1. *ATM asks Pin*
2. *User enters Pin*
3. *ATM checks pin*

Postcondition: *User pin is valid*

This means that for the function, 'Validation' to occur, the precondition, '*User card is inserted'* should be true and by the end of the function execution, the condition' *User pin should be valid'* should be proved true.

3.3.3 Floyd-Hoare Triples

Each scenario can be pictured in terms of two sets of predicates. The first set of predicates refers to the precondition of the scenario and the second set of predicates refers to the postcondition of the scenario.

Hence, we can view each scenario as a Floyd-Hoare triple represented by:

$(| \varphi |)\ S\ (| \Psi |)$

where 'φ' denotes the precondition of the scenario, 'Ψ' denotes the postcondition of the scenario and 'S' denotes the whole set of scenario steps sequentially.

Actually, the Hoare triple was formulated to verify the functional properties of programs, but as explained in the previous section 3.3.2 each scenario can be structurally assumed to be similar to program segments. The alternative method used for PSV approach relies on this interpretation of verification with Hoare triples.

Hence, the meaning of the above Floyd-Hoare triple can be stated as:

'If the scenario is subjected to a situation fulfilled by the 'φ' precondition predicate set, then the execution of the scenario segment, S, should yield 'Ψ' postcondition predicate set.'

For example:

$(| i = j |)\ \mathbf{S}\ (| (i >= j)\ \&\ (j = k) |)$

This means that if we execute S through the situation, *'i=j'*, then the final state after the execution of S will definitively result in '*(i >= j) & (j = k)'*.

It is important to note that representation using Hoare triple gives more freedom to the developer/theorem prover. For instance, in the above example, the triple does not deal with the situation when *i!=j*.

For proving Hoare triples, a proof calculus is developed similar to the proof calculus used for propositional and predicate logic proofs. In the case of propositional and predicate

logics, the proofs were derived by delving into the formula based structure and in most cases the proofs will be terminated by the implication rules. The only exception to proving Hoare triples comes from the fact that the proofs have to deal with two new logical entities, the precondition, φ, and the postcondition, Ψ. The Floyd-Hoare triples are proved using rules related to one kind of statement as well as rules for compositional statements.

There are two types of correctness associated with Floyd Hoare triples:

- Partial correctness
- Total correctness

Partial correctness

The partial correctness for a Hoare triple, $(|\ \varphi\ |)\ S\ (|\ \Psi\ |)$, states that if the scenario, *S*, is executed , given any situations which satisfies, φ then the result of the execution will be Ψ, if *S* reaches termination.

That is,

$$|=_{par} (|\ \varphi\ |)\ S\ (|\ \Psi\ |) \text{ is true.}$$

According to partial correctness, a scenario should not necessarily terminate to guarantee Ψ.

Total correctness

The total correctness for a Hoare triple, $(|\varphi\ |)\ S\ (|\ \Psi\ |)$, states that if the scenario, *S*, is executed at any situations which satisfy, φ then the result of the execution will be Ψ, and *S* always terminates.

That is,

$$|=_{tot} (|\ \varphi\ |)\ S\ (|\ \Psi\ |) \text{ is true.}$$

According to total correctness, a scenario is guaranteed to be terminated to yield Ψ.

3.4 Domain Model and related concepts

Models are absolute abstractions of systems intended in absorbing the problem domain by using discrete representations. According to Martin Fowler [56] a domain model is "*An object model of the domain that incorporates both behavior and data, it creates a web of interconnected objects, where each object represents some meaningful individual, the*

model is a visual representation of conceptual classes or real situation objects in a domain". The Rational Unified Process [61] defines domain model as "an incomplete object model centered on specifying products, deliverables and events pertaining to the domain and that this model never implies the responsibilities owned by the users."
Over the last few years, domain driven design has gained much potency, the domain driven approach for establishing a design cannot be accepted as a technology, rather it should be considered as a strategy to tackle highly complex problem domains. Often, "domain driven design" and "model driven design" are interpreted as the same by the software community, in real, domain driven design lays the foundation for models. In most contexts, domain model is termed as the class diagram which depicts the classes or entities, the attributes and operations related to entities and the relationships between entities, more specifically the responsibility of classes are delivered to operations referring to classes. With respect to [59], a conceptual class is explained in terms of its symbol, intension, and extension. The symbol indicates the picture representing the concept, intension describes the concept and extension refers to the instances connected with the concept.

The important properties that should be considered for a good domain model are:

- Consistency
- Correctness
- Abstractness
- Completeness

Since a domain model represents the whole system in general, there experiences difficulty in expressing all the individual component details. Each component as itself may consist of smaller entities and relations and operations pertaining to these entities, this notion of component presents a higher degree of abstraction to the model. A domain model usually represent a variety of applications where each application consists of different concepts that could be represented individually or sometimes a single name is used to represent more than one entity, hence there is a degree of inconsistency concerning the domain model. Because of the abstractness in domain models, it is obvious that some concepts, attributes or relationship information can be left out from the domain model, this contribute to the issue of incompleteness in the model. There is also the possibility of

expressing incorrect relationships between entities or wrong cardinalities associated with relations; hence, it is necessary to ensure correctness for a good domain model. In order to create a good domain model, all the above discussed issues are to be resolved.
The information embedded in the domain model falls under the following [24]:

- knowledge about the conceptual entities
- knowledge about the domain scope
- information on Use Cases
- functional behavior

3.4.1 Creating a domain model

The type of domain model to be constructed depends on the type of application, whether it is process driven or a structural based application. Usually, an activity diagram is essential concerning the process driven applications whereas a class diagram proves to be more prominent for structurally important systems. But for most of the applications the UML class diagram is used to picture the domain. The following steps aids in creating a domain model [51]:

- Identify all the conceptual classes representing the domain objects
- Establish generalization relationships between classes
- Represent each class by a 'description diagram'
- Enumerate the attributes relating to each class and represent them in the description diagram
- Establish associations, aggregations and cardinalities related to the associations between classes.
- Discover operations pertaining to functions of each class and represent it in the description diagram

The most commonly accepted standard representation for domain modeling is the UML domain model. A typical UML model example from is shown below in Figure 7:

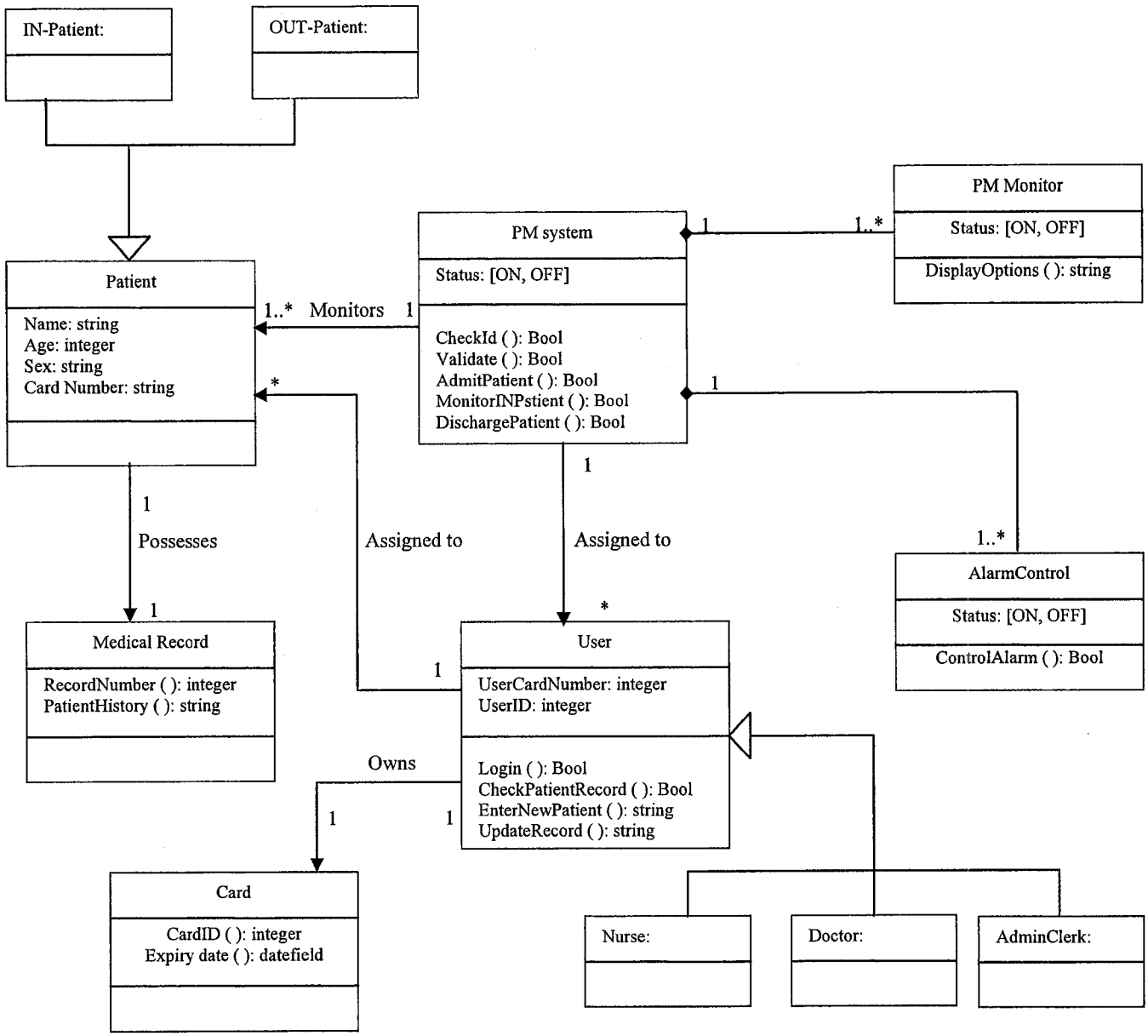


Figure 7: An example of UML class model

3.4.2 Domain model representation used in PSV approach

The Domain model that we use for our approaches, consists of entities and the operations related to the entities based on *'contracts'*. The domain operations are specified using 'contracts' [11]. The notion of contracts was first introduced in the *'Design by contract'* approach by Bertrand Meyer. A contract is defined as "defining a set of expectations

between the two parties, that vary in strength, latitude and negotiability, and specified penalties [for contract violation] [12]". Designing by contracts describes the design with respect to behavior of the components used in the design and the communication among these components. Usually, the component behavior is expressed by means of operation preconditions, postconditions and invariants. An operation, according to the contract concept, is expected to work when the preconditions are true; after completion of the operation, the postconditions should be the result and the invariants (set of conditions) should hold true before, after and during the whole operation. Even though, there are many elements associated with the contracts in spite of the preconditions, postconditions and invariants, these three elements are known to be 'basic fabrics' [11] of contracts. In the proposed approaches, the postconditions of operations are expressed in terms of '*AddedConditions*' and '*WithdrawnConditions*' [86]. The '*AddedConditions*' are those conditions which will appear and should hold true after the operation is executed, while the '*Withdrawn*' conditions are the list of conditions that do not have an impact on the operation after its execution.

For example, in the context of ATM application (domain model Figure 8), the ATM operation "*display welcome message*" is described as:

Operation : display welcome message
AddedCondition : ATM Display is welcome message
WithdrawnCondition : ANY ON USER Card

The operation, *'display welcome message'*, suggests that after this operation, the *Addedcondition*, *' ATM Display is welcome message'* should be necessarily true, that the concept *'ATM'* which has the attribute *'Display'* should hold the attribute value *'welcome message'*. It is to be noted that the value *'welcome message'* for the attribute *'Display'* of *'ATM'* is defined in the domain model. Meanwhile, the *WithdrawnCondition*, *' ANY ON USER Card'* should be neglected after the operation, that is, when this operation is finished, the concept,*'USER'* has the attribute, *'Card'* which can take *'ANY'* value, that means the attribute value of *'Card'* is not considered, hence any value on attribute *'Card'* should be removed. The following Table 2 depicts the syntax of a domain model used in the proposed approach.

Domain components	*Constituents of domain components*
System Concept	Concept, Aggregate, Attribute, Value set, Operation set
Concept	Concept, Aggregate, Attribute, Value set, Operation set
Aggregate	Aggregate, Attribute, Value set, operation set
Attribute	Value set
Operation set	Operation
Operation	Precondition, AddedCondition, WithdrawnCondition
Value set	values

Table 2: Domain Model Syntax used for the proposed approach

It is not necessary that every operations should contain *AddedConditions* and *WithdrawnConditions*. An operation is compelled to contain *AddedConditions*, but can be devoid of *WithdrawnConditions*. This is because the Addedconditions are those conditions which should be true and should be necessarily satisfied after the successful completion of the operation. For example, the operation *'ask PIN'* (Table 2), is defined as:

Operation : ask PIN

Addedcondition : USER PIN is requested,

ATM Display is pin enter prompt

Here, after the completion of the operation, *'ask PIN',* it is required that the concept *'USER'* has attribute *'PIN'* having the value *'requested'* and the concept, *'ATM'* attribute *'Display'* has the value *'pin enter prompt'*.

An example of a Domain model depicting the concept of ATM example, taken from [85] is shown below in Figure 8:

```
Concept:ATM, Attributes:Display,Transaction status
 Operation:display welcome message
  AddedCondition:ATM Display is welcome message
  WithdrawCondition:ANY ON USER Card
 Operation:ask pin
  AddedConditions:User PIN is requested, ATM
   Display is pin enter prompt
 Operation:ask user validation
  AddedCondition:User Validation status is bank
                 inquired
 Operation:display operation menu
  AddedCondition:ATM Display is operation menu
 Operation:eject card
  AddedCondition:User Validation status is
                 card ejected
 Operation:display error message
  AddedCondition:ATM display is error message
Concept:User, Attributes:PIN, Validation status
 Operation:insert Card
  AddedCondition:USER Validation status is
                 card inserted
  WithdrawCondition:ANY ON ATM Display
 Operation:enter pin
  AddedCondition:USER Validation status is
                 pin entered
 Operation:select cash withdrawal
  AddedCondition:USER Transaction is cash withdrawal,
   ATM Transaction status is withdrawal initiated
  WithdrawCondition:ANY ON ATM Display
```

Figure 8: ATM domain model example

3.5. Modeling requirements with Use Cases

Use Case [81] depicts the functional behavior of the system under development. As soon as the problem is defined and the requirements are stated informally at the initial planning stage, it becomes essential to structure the requirements and study them for deriving a formal requirement specification. Modeling the requirements using Use Cases imparts a defined and an understandable structure for the requirements. A Use Case model consists of a collection of Use Cases and there exists flow of control among these Use Cases. Use Case modeling commences with the identification of the principal actors and Use Cases and continues with descriptions and defining relationships among the Use Cases. At further stages, additional information and extensions are added to the Use Cases as needed. Basically Use Case modeling delivers the functionality of the system with the help of Use Case diagrams. We use the concept of Use Cases to model the informal requirements

which is in its most crude form, the ambiguous English language. Use Cases picture the functionalities of the domain very thoroughly due to its articulate behavior.

3.5.1 Use Cases and Use Case scenarios

Use Cases are regarded to be one of the effective ways to model requirements and guide through the path of software development cycle. The industry standard UML notation has accepted Use Case representation to be one of its *"best communication tools"* for requirement analysis [2]. According to Alistair Cockburn [26], Use Cases can be regarded as a *"scaffolding"* to associate various levels of requirements and user perspectives, business and data constraints.

Some of the different interpretations for Use Cases are proposed by Larman[51], Jacobson[45], Booch[16].

A Use Case description includes:

- A Use Case name and the basic purpose of the Use Cases.
- General Requirements such as the permissions of the users of the Use Case
- Use Case scenarios which describe the course of actions to be performed sequentially under different possible situations
- Constraints which describe conditions and rules attached to the Use Case
- Diagrammatic representations.

The core elements of Use Case modeling are Actors and Use Cases. An example of a Use Case diagram is depicted in Figure 9.

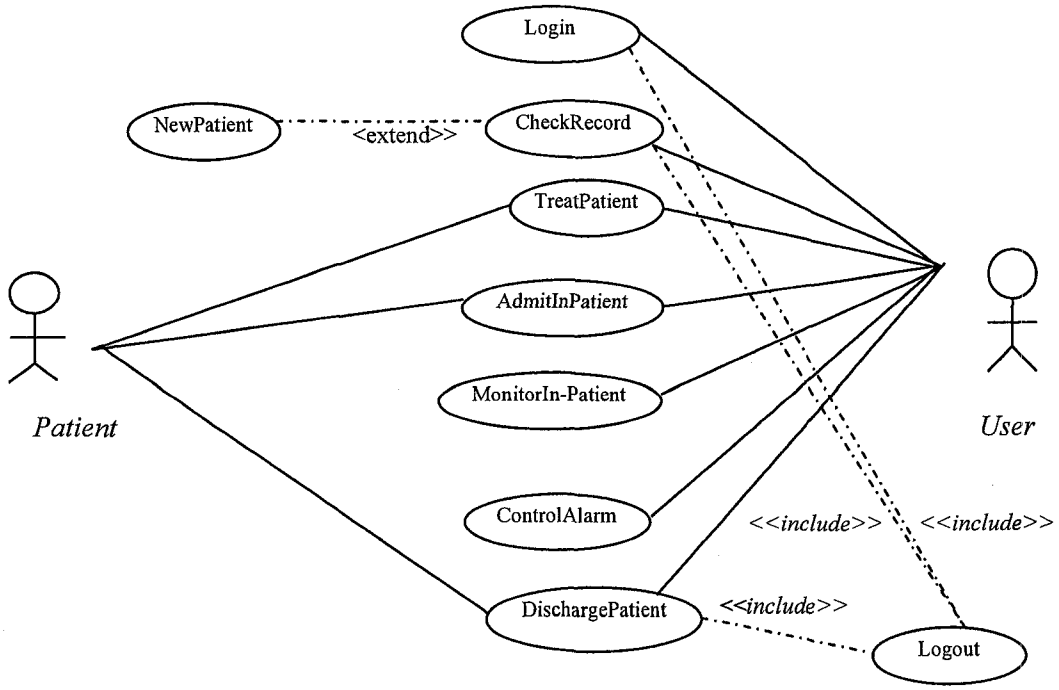


Figure 9: UML Use Case diagram for Patient Management System

Use Cases:

A Use Case is a set of sequential lines of actions depicted by scenarios. It can also be stated as a set of 'related interactions' between the actors the system. Every Use Case holds a 'contract' relating to the system's behavior to perform a unique function. For example, in the context of an ATM application, 'withdrawal' Use Case takes care of all situations of money withdrawal in the system. A Use Case can be included by other Use Case, this helps to avoid redundancy since some common elements in both the Use Case can be avoided. During some abnormal situations, it is possible to extend one Use Case's behavior to another.

UML provides a number of pictorial representations for Use Case scenarios. The most common among them are the Use Case diagrams and the Use Case sequence diagrams.

Use Case scenarios:

A scenario comprises a subset of actions from the Use Case, that is, a particular course of actions aimed at producing an outcome. The end users of the system possess a goal that should be executed by the system. The Use Case consists of scenarios which can satisfy the users' goal and some scenarios that can violate the goal. The success scenarios are expected to occur in normal circumstances and they occur frequently, hence called "happy scenarios". The scenarios which occur frequently but do not lead to the desired goal are termed the "unhappy scenarios" and those scenarios which do not occur frequently and lead to abnormal or exceptional situations are referred to as "exceptional scenarios".

3.5.2 Use Case Template and Representations in Predicate based Sequential Verification (PSV) approach

The Use Case format used in the proposed approach is a modified form of Cockburn's template. The following Figure 10 presents the UML notation for the abstract syntax of Use Case used by this template [86]:

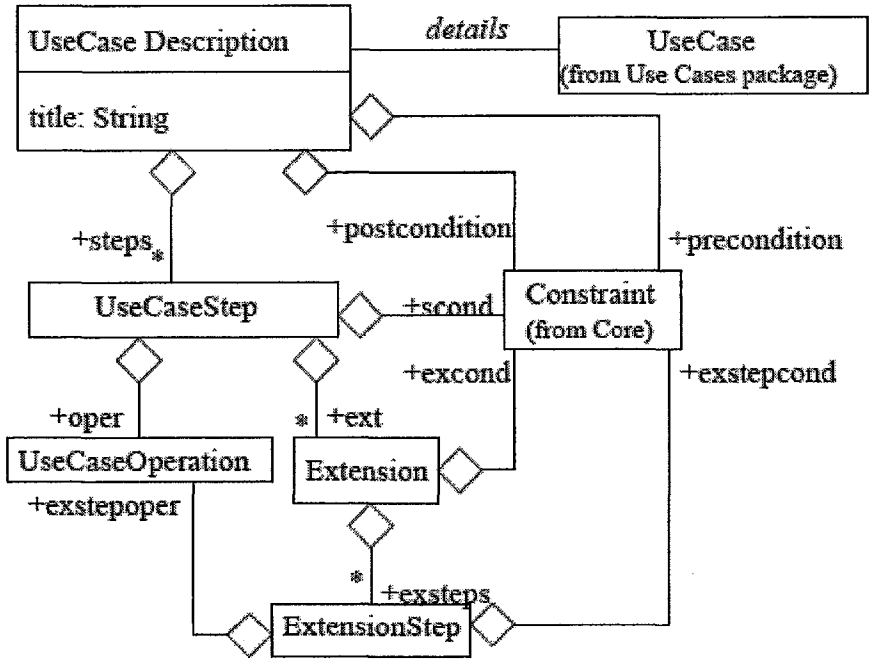


Figure 10: Use Case Abstract Syntax

The Use case abstract syntax as depicted in Figure 10 consists mainly of the Use Case description. The Use Case description comprises of Use Case steps which are individually defined by the Use Case operations. The Use Case description also includes extensions (if any) to each Use Case step to define the alternative flow of actions and constraints that should be obeyed by the Use Cases. The Use Case operations in turn are defined by postconditions resulting from the execution of the operation.

A Use Case can be represented by a four-element tuple namely *[Title, Precondition, Steps, Postcondition].* 'Title' represents the identity of the Use Case, 'Precondition' specifies all the set of conditions that should be satisfied before performing the Use Case, 'Steps' indicates the course of actions possible in the Use Case and 'Postcondition' refers to the list of conditions that result after executing the Use Case. Each Use Case step in 'Steps' is related to an operation which performs the functions indicated by the Use Case step. So, a Use Case step can be represented by a tuple [*SCond, Oper, Ext*] where *SCond* denotes a condition that should hold for the step, *Oper* indicates an operation and *Ext* is a set of extensions that exhibit alternative behaviors which can occur at a Use Case step.

The elements of a Use Case are: *Title, Primary Actor, Goal, Precondition/Postcondition pair, Use Case STEPS, Extensions* (if any), *ANY Extension* (if any), *Extension points* (if any). An example of Use Case for a patient monitioring system (PM system) is shown in the Figure 11 below:

Title: LOGIN
Primary Actor: USER
Goal: To grant access to PM system to perform activities like admitting patients
Precondition: PM system is ON, USER is NOT logged in
Postcondition: USER is logged in

Use Case STEPS:

1. USER inserts card
   Extension point→ card inserted
2. PM system asks PIN
3. USER enters PIN
4. PM system checks PIN
5. If USER PIN is valid, PM system displays MENU options
6. After 45 seconds, PM system ejects card

Extensions:

*1
*1a USER clicks CANCEL
*1b PM system ejects card
1.a. USER Card is invalid
1.a.1. PM system starts system status alarm
1.a.2. After 20 seconds, PM system ejects card
2.a. After 60 seconds
2.a.1. PM system starts system status alarm
2.a.2. After 20 seconds, PM system ejects card
4.a. USER PIN is invalid and USER number of attempts is < 4
4.a.1. Go to step 2
4.b. USER PIN is invalid and USER number of attempts is = 4
4.b.1. PM system starts system status alarm
4.b.2. After 20 seconds, PM system ejects card

Figure 11: Example of a Use Case

[The above example from Figure 11 is refered further to describe the elements of the Use Case]

The notations and grammatical structure used for the above Use Case decription in Figure 11 used for the proposed approaches are described below.

a. Title : indicates the name of the Use Case, e.g: LOGIN

b. Primary Actor: triggers the Use Case, e.g: USER

c. *Goal:* states the objective of the Use Case on its successful completion, e.g: To grant access to PM system to perform activities like admitting patients

d. Precondition/Postcondition : states the preconditions and postconditions of the use case, e.g:

Precondition: PM system is ON, USER is NOT logged in

Postcondition: USER is logged in

e. Use Case STEPS: the ordered sequence of Use Case steps specifying Use Case operation relating to each Use Case step

Each Use Case step in turn can be:

1. *Concept operation instance:* indicates the operation of an instance of the concept (a concept is an actor or the system under consideration)

   A concept operation instance has the format :

   Delay specification/ condition statement/ determinant/operation of the entity

   Delay specification: an *after delay* e.g: *after 30 sec* (or) a before delay e.g: *before 30 sec*

   Condition statement: IF condition *THEN*

   Determinant: 'a', 'an' or *'the'*

   Operation of the entity: indicates the action (operation) taken by the entity

   Example: *USER inserts card*

2. *Branching statement:* it helps to transfer control from one step to another

   The format is:

   Delay specification/ condition statement/ goto step-reference

   Example: *Goto step 2*

A Use Case step can also contain *step extensions* which describe alternate behaviors for the step. A *step extension* specifies an ordered set of extension operations under specific conditions. A *step extension* can be a *concept operation instance* (or) a *branching statement* as described above.

Example: *STEPS:*

1. *USER inserts card*

   Extensions:

1.a. USER Card is invalid

1.a.1. PM system starts system status alarm

1.a.2. After 20 seconds, PM system ejects card

f. ***ANY Extension:*** an ordered set of step extensions that can be applied to all Use Cases steps, it is indiacted by a'*' preceeding the step.

Example: *STEPS:*

1. USER inserts card

Extensions:

**1*

**1a USER clicks CANCEL*

**1b PM system ejects card*

g. ***Extension Points:*** locations which indicate *extension Use Cases*

Example: *1. USER inserts card*

Extension point → card inserted

A Natural Language representation is used for expressing *operations* and *conditions* in Use Case as described in [86]. The *conditions* in a Use Case are specified in 'predicative phrases' [86]. A 'Definite Clause Grammar' (DCG) [34], a contextual grammar for NL, is used for describing the grammar for this Natural Language representation. The context used for Use Cases in the PSV process is the domain model (please see section *3.4.2*). The following Table 3 shows a partial DCG for conditions as described in [86].

condition → pred-phrase
condition → pred-phrase, conj, condition
condition → negation, condition
pred-phrase → noun-phrase(N), verb, value(N)
Noun-phrase(C) → determinant, [C] {concept(C)}
Noun-phrase ([C,A]) → determinant, [C], [A], concept-attribute (C,A)}
Value(N) → determinant [A], {discrete (N), value (N,A)}
Value(N) → comparison {not (discrete(N))}
conj → [AND]/ [OR]
verb → {be-form}/ {derived from (become)}

Table 3: Partial DCG for conditions

The above DCG (Table 3) contacts the domain model through *concept, concept-attribute* and *values* (refer section *3.4.2*).

E.g. {'*User*' [concept], '*Identification*' [attribute], ('*invalid*', '*valid*') [values]}

3.6 State chart diagrams

State chart diagrams [37] present a more formal way of understanding the domain model. State charts exhibits the dynamic behavior of an entity by visualizing the events, triggers, responses and transitions related with the entities and internal data flows. Unlike other preliminary models like the class diagram, sequence diagrams etc, state chart diagrams are not static since the state chart diagram depicts the entities' course of events that occur at various situations and thus defines the complex and abnormal functionalities of entities. It is necessary to use state chart diagrams when we need to analyze the character of an entity through different states within the whole system. The state charts depict the entities at all available states depending on the events occurring upon them. To simplify, a state of an entity can be described as the status of the list of the attributes of the entity at an exact point of time. When events are applied to the state, the state of the entity may change. The state diagram expresses all these changes in states in different situations. For instance, When the 'start' button is pressed on a microwave oven, the oven starts running from the previous state (idle), the pressing of the start button being the event and 'idle' state being the initial state causes 'running' to be the resultant state. The representation using state chart diagram is one of the state based requirement specification methods. A state chart diagram can be thought of as a graphical representation of a state machine. The state machine is the responsibility of a model component like a classifier in the context of that state machine [70]. Depending on the context, various semantic constraints are conveyed by the state machine. A finite state machine is an abstract machine comprising of finite sets of states, transitions associated with the states and the list of events that cause the transitions. At a fixed point in time, an abstract state machine is subjected to only a small set of actions brought out a set of events. Due to the after effects of actions, the behavior of the state machine changes to a corresponding set of subsequent resultant states. Bertrand Meyer [11] understands the state machine to comprise of predecessor states and triggers collectively called the 'precondition invariants', the resultant state is termed the

'postcondition invariants' and event signatures.

Based on the graphical representation of state machines, the state chart diagram consists of nodes and edges forming a directed graph. The important constituents of the state chart are made up of states and transitions. A state represents an object's (or the whole system's) status at a particular point in time. A state transition [70] is described as a change in the state. A transition is triggered by the occurrence of an event which causes the state to transform from one status to another. The event can occur either external or internal to the object/entity. It can be viewed as a result caused by calling a function or operation.

An example for UML state machine diagram [17] is shown in Figure 12:

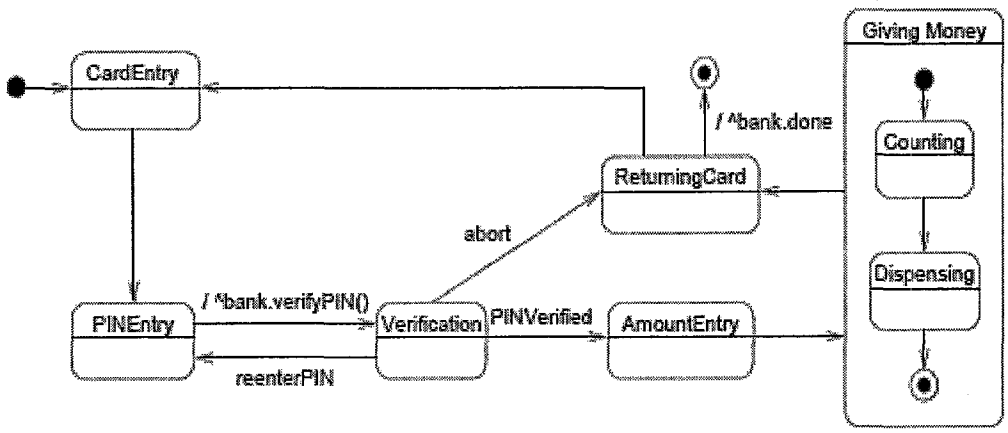


Figure 12: Example of a UML state chart

The two important issues related to state charts are:

- Completeness
- Consistency

Completeness demands the presence of definite transitions for all states when subjected to all the possible events. Consistency requires that a single event should cause only one transition. Each individual state needs to be checked in case of conflicts pertaining to transitions at various hierarchy levels. The ordering for firing of the transitions should be defined for concurrent systems. In case there are more transitions starting from the same

state and fired by the same event, then their guards should not be true at the same time.

3.7 Summary of the chapter

The importance of formal verification and the logical aspects of formal verification discussed in this chapter provide a clear background for the verification approaches proposed in this thesis. The concepts, definitions and the conventions used in the proposed approaches are described effectively with appropriate pictorial representations. In a nutshell, this chapter serves better to present and understand the proposed verification approaches described in Chapters 4 and 5.

Chapter 4. Predicate based Sequential Verification

The design of a system is expected to be free of errors and inconsistencies to guarantee a successful software product. Although Use Cases extract the user requirements, they are not enough to model the complete set of user requirements. Use Cases tend to explain the relationships between end users and system as a solution to the problem to be solved. If a domain model does not exist for the system, then the design starts right from the Use Case with respect to these relationships, thus many details originally addressed in the problem, in other words, the problem description will be lost. So, to preserve the origin of the domain concepts, a domain model is essential. In order to achieve this goal, proper attention has to be given in the development of domain model at the requirement analysis and early design phases. According to the SiN algorithmic approach [38], [88], "the domain model will be incomplete and no model can ever be 'proved correct'. But, this statement is not completely true as it denies validity of the domain model. As suggested by some domain V&V approaches like the B-Method [88], it is possible to verify and validate some aspects and components of domain model.

The reasons for domain model errors and inconsistencies are mainly due to the following:

- Domain models have high degree of abstraction
- In most of the applications, the domain model is constructed based on the designers' assumptions.
- The semantically incomplete domain specification and definition languages

Verification and Validation of domain models is essential before the implementation of the system because of the above mentioned issues [47]. Usually, as soon as a domain model is built, the domain researchers try to encode it into domain definition languages (DDL) like Planning Domain Definition language PDDL [3]. Some may use domain engineering tools like syntax and operator consistency checkers [54]. But, in spite of these tools, a realistic domain model is still open to numerous errors and inconsistencies [88].

This chapter focuses on a verification strategy based on predicate representation for detecting the requirement errors and inconsistencies inherent in the domain model. The proposed verification is termed 'Predicate based Sequential Verification (PSV)'. The PSV module was able to detect requirement errors present in the domain model on Patient Management system case study (Chapter 6) and provide directions to improve the domain model in cases of model inconsistencies and errors.

4.1. The Predicate based Sequential Verification (PSV)

The motivation for the *PSV* process comes from the necessity of domain model requirements verification in accordance with the above mentioned issues.

The main objectives of *PSV* process are:

- Verification of domain model against the Use Case scenarios
- Detecting the invariants for the Use Case scenarios and verifying that the domain model meets the invariants
- Reporting errors or inconsistencies present in the domain model
- Suggesting improvements in the domain model to tackle the errors and inconsistencies.

4.1.1 The Predicate based Sequential Verification (PSV) process

The whole process is depicted in the following Figure 13:

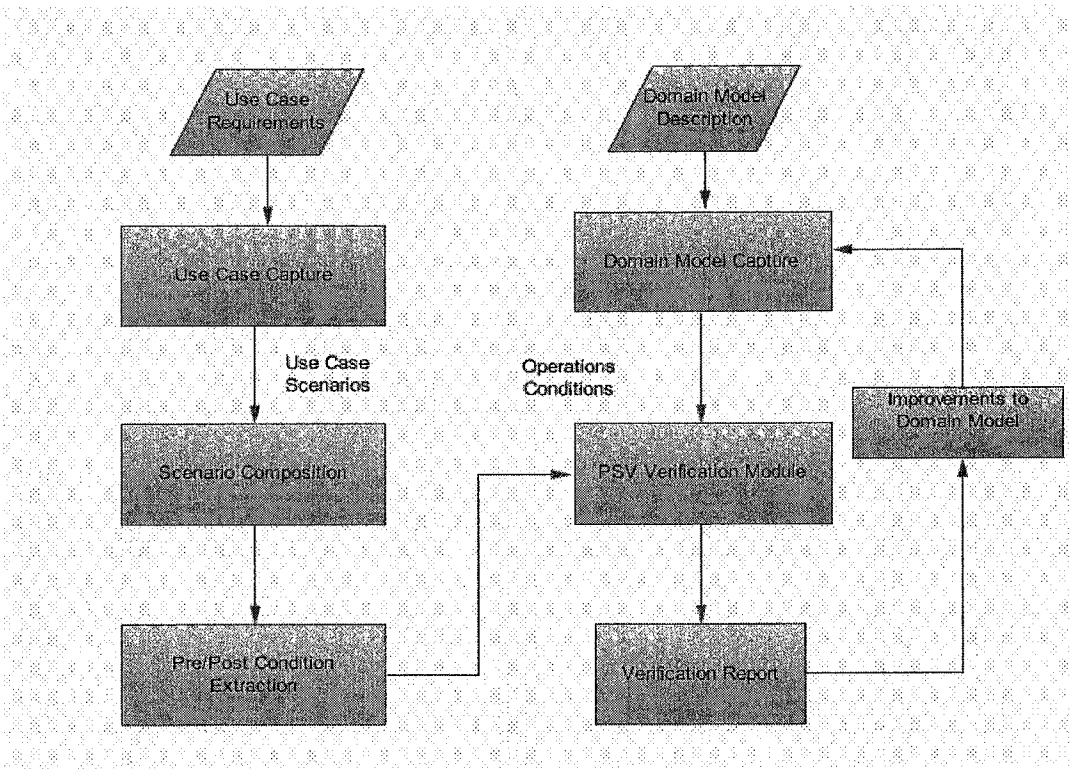


Figure 13: PSV Architecture

The *Predicate based Sequential Verification* process starts with Use Case requirements and domain model as inputs. The Use Cases are extracted from the Use Case requirements and the domain model is captured from the information conveyed by the domain model experts. The Use Case requirements are composed to obtain semi-formal Use Cases represented in the form of a modified Cockburns template as specified in section *3.5.2* of Chapter 3. Each Use Case is further disintegrated to produce scenarios by a 'scenario composition' module, which extracts scenarios from the semi-formal Use Case representation based on certain rules (The rules will be described in the next section). A scenario describes one particular sequence of actions resulting in one unique set of conditions to be met by the scenario. The 'scenario composer' represents each scenario in terms of expected preconditions / postconditions with respect to the user's Use Case

scenario requirements, and an ordered set of steps which depicts the operations of the scenario. The domain model is designed with actors, entities, attributes related to the entities and operations concerned with the entities, as described in section *3.4.2* of Chapter 3. The operations are defined in terms of *AddedConditions* and *WithdrawnConditions* in the domain model. The verification strategy follows the sequential nature of scenarios and incrementally builds the post conditions of the scenario based on domain model operations. Each scenario is taken by the *PSV* verification module and the precondition and postcondition for the scenario are recorded in predicate format. A 'Verification Sequence' is generated as a result of verification. The verification sequence starts with the recorded precondition predicates, these precondition predicates are passed to the first step in the scenario. For each step sequentially in the scenario, the operational conditions corresponding to the Use Case step are extracted from the domain model description. The status values of the precondition predicates will be updated by the operational conditions of the first step in the scenario according to the verification algorithm discussed in the succeeding section. The first line of verification sequence now contains the predicates caused by passing the precondition predicates to the operational conditions corresponding to the first step. This first line of verification sequence is then 'passed' to the next sequential scenario step. The operational conditions for this step are extracted from the domain model description and the verification process is repeated to generate the next verification sequence line. In this manner, the whole verification sequence is generated and the final verification sequence line corresponds to the postcondition of the scenario resulting from the *PSV* verification process. This postcondition obtained from *PSV* verification is verified against the recorded desired postcondition of the scenario. If all the conditional predicates in the desired postcondition of the scenario are included in the postconditions obtained from the *PSV* process, then the verification of the scenario is successful. At the same time, in case, the postcondition of the *PSV* process contains more predicates than those in the desired postcondition, the *PSV* module suggests that the domain model is inconsistent with the Use Cases requirements. So, the *PSV* process generates a report indicating the necessary action to be taken to remove the inconsistencies in the domain model. The inconsistencies are caused due to the assumptions and logical guess works of the developer while developing the domain model. The *PSV* module traces

back the verification sequence and suggest the necessity of updating certain operational post conditions to resolve the inconsistency. Once the inconsistency is manually resolved by the developer, the domain model is subjected to the *PSV* process again. If the result of this verification yields the postcondition to be same as the desired postcondition of the scenario with no inconsistencies, the verification becomes successful with a consistent domain model and the verification report is displayed. In cases where the *PSV* postcondition does not contain all the desired postcondition predicates or is entirely different from the desired postcondition predicates, then a 'verification unsuccessful' report is generated.

4.1.2 Informal requirements capture and representation in PSV process

Ususally, the user requirements are expressed in the form of Natural Language textual statements. Representing informal requirements using Natural Language (NL) has some merits, in spite of NL's innate ambiguous nature. It is easy to communicate with various users of the system by using NL. For instance, NL requirements can serve as working documents for system designers, testers, and editors of user manuals and to establish agreements among end users and developers [36]. At the same time, NL representation of requirements demands ambiguity analysis, completeness and consistency checks. These checks are known to consume much time and manual effort. Besides this difficulty, it is impossible to perform verification involving NL requirements. In order to deal with these NL issues, the NL requirements need to be structured and expressed by a concrete representation. The Use Case representation is used to model the informal NL requirements for the proposed approach. Use Cases happen to picture the functionality of the system very thoroughly due to its articulate behavior. The structuring of requirements using Use Case representation helps to formalize and verify requirements easily and at the same time, preserves the requirements' inherent informal nature to a certain extend. A restricted form of Natural Language representation with improved formal semantics is adopted to represent the Use Case for achieving technical feasibility.

4.1.2.1 Use Case Composition in PSV process

The Use Case representation in *PSV* process is a modified form of the Cockburn's template

for Use Cases. This modified Use Case template expresses Use Cases in structured English while preserving a certain degree of formality. Hence, the Use Case representation in *PSV* process can be considered as 'semi-formal'. The Use Case *operations* and *conditions* are represented in Natural Language using a definite clause grammar for Natural Language description. The informal Use Case requirements in *PSV* process are composed by a Use Case composer to obtain semi-formal Use Cases, according to conventions described for Use Case composition in Chapter 3. Please refer to section *3.5* of Chapter 3 for the description on Use Case Natural Language representations, Use Case template and Use Case composition used in *PSV* process.

A Use Case with simple scenarios used in the *PSV* process is represented in the following Figure 14:

Title: User Identification
Primary Actor: User
Goal: To validate the user
Precondition: ATM status is ON and ATM display is Welcome message
Post condition: ATM status is ON and ATM display is operation menu
STEPS
1. User inserts card in card reader slot
   Extension point→ card inserted
2 .ATM asks PIN
3. User enters PIN
4. ATM asks user validation to bank
5. Bank checks user id
6. If user identification is valid, ATM displays operation menu
Extensions:
  1.a. User card is not valid
    1. a.1. ATM ejects card
    1. a.2. ATM displays error message
    1. a.3. After 30 sec, ATM displays welcome message
  5.a. User identification is invalid
    5. a.1. ATM ejects card
    5. a.2. ATM displays error message
    5. a.3. After 30 sec, ATM displays welcome message

Figure 14: User Identification Use Case for ATM system

The function of the above Use Case (Figure 14) is to validate a user for performing ATM

transactions. The name of the Use Case is specified by the *title* '*User Identification*'. The initiator of this Use Case is '*User*'. The *preconditions* of the Use Case state that the status of ATM machine should be '*ON*' and the ATM system displays the '*welcome*' message. The *postconditions* suggest that after the successful completion of the Use Case, the status of the ATM machine should still be '*ON*' and the ATM system should display 'operation *menu*' to the user. The *STEPS* section, Use Case steps *1-6* indicates the ordered operations to be carried out by each Use Case step. The *extension* section explains the alternate course of actions for particular steps based on certain conditions. For instance, after the complete execution of '*step 1*', if the condition on extension step '*1a*' (*user card is not valid*) becomes true, then extension steps, '*1a1, 1a2 and 1a3*' should occur sequentially. An *extension point* is indicated at step 1(card inserted) which refers to an *extension Use Case*.

4.1.2.2 Scenario Extraction from Use Case and Scenario Composition

Extraction of the scenarios from the Use Case is performed by categorizing the conditions in the Use Case to fall under these divisions:

1. ***Main*** precondition and the normal action sequence of the Use Case ('Happy' or 'Normal' or 'Primary' Scenario)
2. ***Extension step*** condition sequences of the Use Case ('Unhappy' or 'Secondary' Scenario)

The first category is the action sequence which describes the successful completion of the Use Case with no abnormal circumstances. This type of scenario is extracted by following the normal sequence (successful Use Case sequence) without considering the extension steps. The first category is derived from the Use Case's main precondition/postcondition pairs as:

Precondition:
STEPS:

From the above ATM example from Figure 14, the first category scenario type from the

normal sequence, 1-2-3-4-5-6 is extracted as:
*First category sequence (*1-2-3-4-5-6)*:*
Precondition: ATM status is ON and ATM display is Welcome message
STEPS: 1-2-3-4-5-6

The second category corresponds to the alternative scenarios where the successful goal of Use Case is not achieved, but these scenarios occur often due to errors or mistakes from the actors. These action sequences refer to the *extension steps* of the Use Case. The second category is derived from the Use Case's extension condition sequence as:

Extension step:
Extension condition (guard):
Extension condition sequence:

For the example from Figure 14, the second category scenario type from the *secondary sequences, 1- 1a- 1a1-1a2-1a3* and *5-5a-5a1-5a2-5a3* are:

*Second category sequence (*1- 1a- 1a1-1a2-1a3)*:*
Extension step: step 1
Extension condition (guard): step 1a
Extension condition sequence: 1a1-1a2-1a3

*Second category sequence (*5-5a-5a1-5a2-5a3)*:*
Extension step: step 5
Extension condition (guard): step 5a
Extension condition sequence: 5a1-5a2-5a3

The precondition for the first category is the '*Precondition*' and the precondition for the second category is the '*Extension step*'. The scenario extractor in PSV process extracts scenarios based on the above described conventions.

4.1.3 Domain model composition in PSV process

The domain model representation and conventions used for the PSV process are described in section *3.4.2* of Chapter 3. The operations in domain model are expressed by means of '*Added*' and '*Withdrawn*' *conditions*; these *conditions* are represented using the DCG grammar for Natural Language descriptions as specified in Table 3 of Chapter 3, with the following exceptions:

- For '*AddedConditions*', the verbs are to be specified in either '*to be*' or '*to have* ' form and the verbs should be in *present* tense.

  That is , only the following verbs are allowed for *AddedConditions*:

  [is, are, is not, are not, has, have, has not, have not, can, can not]

- For '*WithdrawnConditions*', the format should be:

  | *'ANY'* | | *'ON'* | *concept_attribute(or) sub-concept_attribute*|

4.1.4 Assumptions for PSV Verification

Assumption 1:

Use Cases should be written in a simple restricted Natural Language format according to the 'Definite Clause grammar (DCG) as explained in section *3.5.2* (please refer Chapter 3). Although it is possible to express all the requirements in the below described simple English format, we do not appreciate the use of iterations while writing requirements in Natural Language. This format supports only simple conditional words like "IF".

Assumption 2:

The domain model operations are supported by DCG and the operations are expected to be specified using '*AddedConditions*'and '*WithdrawnConditions*' only. The *AddedConditions* and *WithdrawnConditions* are written in *restricted verbal formats* as described in section 4.1.3.

Assumption 3:

The extraction of scenarios from Use Case is achieved by categorizing the action sequence in Use Case to strictly fall under the two categories as mentioned in section *4.1.2.2*

Assumption 4:

All the Use Case steps should have corresponding entities, attributes, and operations, if any, defined in domain model description.

Assumption 5:

Conversion of operational conditions to predicates in *PSV* verification is in accordance with rules explained in the next section (section *4.1.5*).

Assumption 6:

Operations in the domain model are specified based on the notion of 'contracts' as described in section *3.4.2* of Chapter 3.

Assumption 7:

The verbs in preconditions/postconditions of scenarios and the *AddedConditions* of domain operations can take either 'to be' or 'to have' forms.

The verbs are restricted to:

[is, are, is not, are not, has, have, has not, have not, can, can not]

4.1.5 Predicate Conversion and Representation

For performing *PSV* verification, a common logical representation is needed for the preconditions and postconditions of the scenarios, and the domain operation conditions specified in the Domain model. The logical representation used in PSV process is termed the *logical predicate* representation. According to this representation, the semi-formal DCG based domain operations and pre/postconditions are converted to logical predicates. A *logical predicate* is similar to a predicate used in first order logic and takes only one value as its argument. The general representation for a *logical predicate* is:

'Term (value)'

E.g: Ax (b)

Here, 'Ax' is a predicate term which can take the value 'b'.

In the context of *PSV*, 'Ax' represents the *entity* (An entity could be a concept or an attribute of a concept) and 'b' represents the *value* possessed by that *entity*.

Since, the pre/postconditions and domain operations are in the semi-formal DCG based NL formats and takes only 'to be' or 'to have' verb formats (refer Assumption 7), they can be easily converted to *logical predicates*.

The predicate conversions of pre/postconditions and domain operations are performed

based on the following predicate conversion rules. There are four basic conversion rules:

a) The *first* rule applies to the *pre/postconditions* of scenarios and *AddedConditions* of domain operations
b) The *second* rule applies to the *WithdrawnConditions* of domain operations
c) The *third* rule deals with *connectives* like 'and' etc.
d) The *fourth* rule applies for *delay* clauses like 'after 30 sec'

Conversion rule 1:

The pre/postconditions and *AddedConditions* have the basic format:

[Determinant] concept_attribute / verb ('to-be' or 'to have' forms)/ value

(Please refer to section 3.5.2 of Chapter 3 and Assumption 7 of PSV process for the above format description)

They can be converted to the *logical predicates* as:

Concept. attribute (value)

E.g *: Precondition:* ATM status is ON

Is converted to *ATM. status (ON)*

Conversion rule 2:

The WithdrawnConditions format:

|*'ANY'*| |*'ON'*| |*concept_attribute(or) sub_concept_attribute*|

(Please refer section 4.1.3 for this format description)

They are converted to *logical predicates* as:

Concept. attribute ('ANY')

E.g: *WithdrawnCondition: ANY ON* User Card

Is converted to *User.card (ANY)*

Conversion rule 3:

Connectives like 'and' will suggest the later part of the sentence to be another predicate and the conjunction word is converted to ','. The conditional verbs like 'If' are neglected for simplifying the conversion.

For example:

Let us take the precondition:

Precondition: "ATM status is ON and ATM display is Welcome message." is translated to *logical predicates* as:

ATM. status (ON), ATM. Display (Welcome message)

Conversion rule 4:

The *delay* phrases usually take the form,

|before/after| |number| |unit of time|

(Please refer section 3.5.2 of Chapter 3 for the format description)

These *delay* phrases are converted to *logical predicates* as:

For *'after-delay': TIMEOUT ('number' 'unit of time')*

For *'before-delay': TIME ('number' 'unit of time')*

E.g 1: After 30 sec, ATM display is error message

Is converted to

TIMEOUT (30 sec), ATM. Display (error message)

E.g 2: Before 20 sec, ATM Display is error message

Is converted to

TIME (20 sec), ATM. Display (error message)

4.1.6 Algorithm for PSV Verification

The main objective of the *Predicate Sequential Verification (PSV)* process is to check the Domain model against the Use Case scenarios to confirm that the model satisfies the post conditions, as desired by the Use Case scenario requirements. The input to the PSV module is a Use Case, the domain model, and the expected postconditions for the Use Case. The scenarios will be extracted from the Use Case using the scenario composer as described in section *4.1.2.2.* For each extracted scenario, the preconditions and the recorded (expected) postconditions are translated to *logical predicates.* The logical predicates corresponding to the precondition of the scenario (otherwise called 'logical-predicate-precondition') is passed to the first semi-formal scenario step. A NL parser parses the scenario step (in accordance with the parsing rules of DCG) and retrieves the *AddedConditions* and *WithdrawnConditions* (if any) of the domain operation

corresponding to the scenario step. Then, the next scenario step is subjected to the same process as above and the process is repeated until the last scenario step is processed. The set of *logical predicates* corresponding to the *last* scenario step are the '*actual postconditions*' obtained after the scenario execution. Finally, the *recorded* postconditions of the scenario are verified against these '*actual postconditions*' to determine whether the domain model conforms to the expected requirements of the scenario. The verification algorithm for the PSV process is shown in Figure 15:

```
Procedure Verification (UC: Use Cases model; D: domain model)
Begin
        SC [] := Extract_Scenario (UC);
        For each scenario, MS, in SC []
        Begin
            MainPre []:= Extract_Convert_Pre (MS, D);
            MainPost []:= Convert_Post (MS, D);
            Verification_Sequence []:= NIL;
            PSV_Verify (Verification_Sequence [], MainPre [], MainPost [], D, MS);
        End
End
```

Figure 15: Procedure Verification

```
Procedure PSV_Verify (Ver_Seq [], MainPre [], MainPost[], D, MS)
Begin
        N: = count_steps (MS);
        i := 1;
        j := 1;
        Ver_Seq [i]:= Pre [];
       Repeat for each sequential step, s in MS
       Begin
            COND [add[],withdraw[]) := Extract_Convert_Post_Op_Dom (s,D,add[],
                                                                    Withdraw[]);
            Final_Ver_Seq [j] := Pass_COND (add [], withdraw [], Ver_Seq[i]);
            i: = i + 1;
            Ver_Seq[i]:= Final_Ver_Seq [j];
            j: = j +1;
       End
       UNTIL (j: = N);
       Post_Ver_Seq []:= Final_Ver_Seq [N];
       Match_PostUC_PostVerSeq (Post_Ver_Seq [], Final_Ver_Seq [], Post [], MS, D);
End
```

Figure 16: Procedure PSV_Verify

```
Procedure Match_PostUC_PostVerSeq(Post_Ver_Seq [], Final_Ver_Seq [], Post [], MS, D)

Begin
        IF (Post_Ver_Seq []:= includes (Post []))
          Begin
            IF (Post_Ver_Seq []! = Post [])
               Begin
                  Display ("Verification successful but with inconsistencies in domain
                              model");
                  IMD: = Improve_Dom_Model (MS, D, Final_Ver_Seq [], Post []);
                  Verification (UC, IMD);
               End
            ELSE IF (Post_Ver_Seq [] == Post [])
                  Display ("Verification Successful & Domain model is up-to-date with
                  no inconsistencies", Final_Ver_Seq [], Post_Ver_Seq [], Post [])
          End
        ELSE
          Display ("Erroneous Scenario (OR) Domain Model and the Requirement
                specialists and domain Experts need to be informed", Final_Ver_Seq [],
                                                                Post_Ver_Seq [], Post []);
End
```

Figure 17: *Procedure* Match_PostUC_PostVerSeq

Sub-Procedure	Parameters	Return value	Description
Extract_Scenario	UC: Use Case Model	SC[]	To extract the different use case scenarios from use cases based on the extraction description section 4.1.2.2
Extract_Convert_Pre	MS: Scenario D: Domain model	MainPre[]	For each scenario, *MS, in SC,* the precondition is recorded and converted to *logical predicates*
Convert_Post	MS: Scenario D: Domain model	MainPost[]	For each scenario, *MS, in SC,* the expected postcondition is recorded and converted to *logical predicates*
Extract_Convert_Post_Op_Dom	s: scenario step D:domain model add[]:AddedConditions Withdraw[]:Withdrawn Conditions	COND[add[],withdraw[])	The *AddedConditions* and *WithdrawnConditions* are extracted from the domain operation corresponding to the step, s and they are converted to *logical predicates* according to the conversion rules described in section 4.1.5.
Pass_COND	add[]:AddedConditions Withdraw[]:Withdrawn Conditions Ver_Seq[]:Verification sequence	Final_Ver_Seq []	'passing' operation to obtain the final verification sequence as explained in section 4.1.7
Improve_Dom_Model	MS:Scenario D:domain model Final_Ver_Seq [] : final verification sequence Post[]:expected postcondition of scenario	IMD	Operation suggests improvements needed for the domain model in case of domain model consistencies by analyzing the final verification sequence from the PSV-Verify procedure and the expected postcondition stored in Post[]

Table 4: Description of PSV sub-procedures

4.1.7. Description of the algorithm

The algorithm is represented by the following 'flow' diagram in Figure 18:

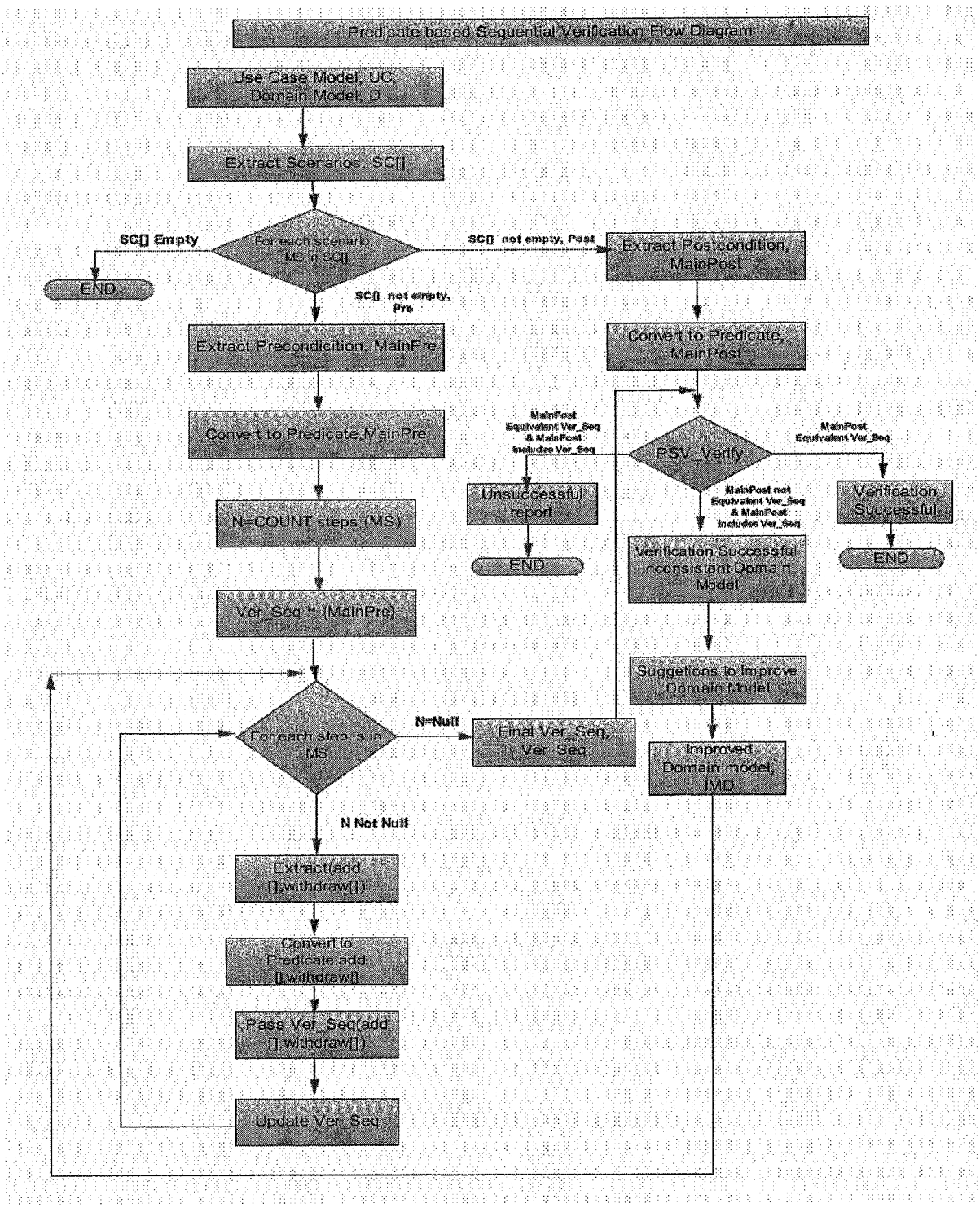


Figure 18: PSV Flow Diagram

The algorithm for *Predicate based Sequential Verification (PSV)* is described by three major Procedures. The first Procedure *'Verification'* is the parent procedure which calls the child Procedure *'PSV_Verify'* which in turn calls its child Procedure, *'Match_PostUC_PostVerSeq'*.

The PSV process starts with the Procedure, *'Verification'* takes in two arguments, the Use Cases Model, *UC* and the Domain model, *D*. The function, *'Extract_Scenario (UC)'* extracts the scenarios from the *UC* and stores it in the parameter, *SC []*.
For each scenario, *MS*, the precondition and the *expected* postcondition is recorded and converted to *logical predicates* by the PSV module using the functions, *'Extract_Convert_Pre (MS, D)'* for precondition and *'Convert_Post (MS, D)'* for the *expected* postcondition. These two functions are implemented using the extraction and composition mechanisms described in section *4.1.2.2* and the logical predicate translation process described in section *4.1.5*.

After the conversion of preconditions and postconditions to logical predicates for the scenario, MS, is performed, they are stored by the PSV module in the parameters, *MainPre[]* and *MainPost[]*. The procedure, *'PSV_Verify (Ver_Seq [], MainPre[], MainPost[], D, MS)'* is called by passing the logical-predicate- preconditions (*MainPre[]*), the expected logical-predicate-postconditions (*MainPost[]*), domain model (*D*) and scenario (*MS*). A *verification sequence, Ver_Seq,* is used to hold the results of verification. The *verification sequence* is initialized to logical-predicate-preconditions, *MainPre []*. The parsing and the conversion process starts by taking the first step, s, in the scenario. The *AddedConditions* and *WithdrawnConditions* are extracted from the domain operation corresponding to the step, s and they are converted to *logical predicates* according to the conversion rules described in section *4.1.5*. This extraction and translation are performed by the function *'Extract_Convert_Post_Op_Dom (s, D, add[], withdraw[])'*. The logical-predicate-*AddedConditions* are stored in *add[]* parameter and the logical-predicate-*WithdrawnConditions* are stored in *withdraw[]* parameter.

Now, the PSV verification commences by passing the *verification sequence*, *Ver_Seq* to the logical-predicate-*AddedConditions* are stored in *add[]* and the logical-predicate-*WithdrawnConditions,* stored in *withdraw[]* parameter, which corresponds to the step, s. This 'passing' is executed by the function, *'Pass_COND (add[], withdraw[], Ver_Seq[i])'*. The *'Pass_COND'* function performs the 'pass' by finding the common *concept_attributes* in the predicates stored in the *verification sequence*, Ver_Seq, and the logical predicates stored in *add[]* and *withdraw[]* parameters. If there any common *concept_attribute* between *Ver_Seq* and *add[]*, then the *value* of that concept_attribute in Ver_Seq is substituted with the *value* of the same concept_attribute in *add[]* and the *Ver_Seq* is updated. If *withdraw[]* is not empty and if there are common *concept_attributes* in the updated *Ver_Seq* and *withdraw[]*, then those *concept_attributes* are removed from the *Ver_Seq* and the *Ver_Seq* is updated again. If there are any new predicates in *add[]* which are not in the Ver_Seq, then these predicates gets added to the *Ver_Seq* and a final updating is performed to *Ver_Seq*. After the new *Ver_Seq* is obtained, the next scenario step, s is parsed and the *AddedConditions* and *WithdrawnConditions* corresponding to this step are extracted and translated to logical predicates in *add[]* and *withdraw[]* respectively. The function *'Pass_COND'* is repeated with the newly obtained *Ver_Seq*, and the new *add[]* and *withdraw[] conditions*. The whole process is repeated until the last scenario step is reached.

The very last set in the verification sequence is the postcondition of the scenario generated by the PSV verification process. This PSV postconditionof the scenario is recorded and is sent to the next Procedure, *'Match_PostUC_PostVerSeq'* along with the whole verification sequence (*Ver_Seq*), the recorded desired postconditionof the scenario (*Post[]*), the domain model (*D*) and the scenario (*MS*). In this Procedure, the obtained PSV postcondition of the scenario (now present in *Ver_Seq)* is verified against the recorded desired postcondition of the scenario (*Post[]*). If all the predicates in the *Post[]* are included in the *Ver_Seq*, but at the same time, if the *Ver_Seq* contains some additional predicates not in *Post[],* then the PSV module reports that the verification is successful but there are some inconsistencies in the domain model descriptions. In that case, the PSV module generates a report regarding what caused the inconsistencies and how to deal with these inconsistencies. The

improvements needed for the domain model are handled by the Procedure, *'Improve_Dom_Model (MS, D, Final_Ver_Seq [], Post [])'*. Usually, the inconsistencies are caused by giving some wrong or unnecessary assumptions while defining operations, or due to the presence of some unnecessary withdrawn Conditions. So, the developer and the domain model expert should consult with each other to remove these inconsistencies and once, the domain model is updated, the new domain model becomes a candidate to the whole process of PSV verification. The PSV verification is repeated until the domain model is free of requirement inconsistencies. When the predicates in PSV postcondition and the desired postcondition are the same, the verification is terminated conveying the 'verification successful' message with verification report. In case, the PSV postcondition predicates are not same as the desired postcondition predicates, the verification goes unsuccessful and needs the extreme attention of the developer, user and the domain model expert.

Illustration:

The whole approach is illustrated by taking the 'happy scenario' (Figure 14 page number: 68) namely:

Precondition- 1-2-3-4-5-6 (postcondition).

The precondition is:

ATM. Status (ON), ATM. Display (Welcome)

The next step is: step 1: *User inserts card in card reader slot.*

The domain operation corresponding to this step is to be retrieved from the Domain model.

From Figure 8 (Chapter 3): we have the definition for insert card operation as:

Concept: User Attributes: PIN, Validation status

Operation: insert Card

AddedCondition: USER Validation status is card inserted

WithdrawnCondition: ANY ON ATM Display

So, the conversion for step 1 will be:

User. ValidationStatus (card inserted), ATM. Display (ANY)

According to our predicate based sequential verification, the first line is traversed through the second line, then, the values for the attribute which is same as in first line will be substituted by the attribute value in second line and if there is any 'ANY' value, that predicate will be discarded. All other predicates in both the lines which are dissimilar remain the same.

So, the verification of first line (precondition – *line 1 below*) and second line (*line 2 below*) yields:

ATM. Status (ON), ATM. Display (Welcome) --- line 1

User. ValidationStatus (card inserted), ATM. Display (ANY) ---- line 2

Verification sequence 1 through 2 is:

ATM. Status (ON), User. ValidationStatus (card inserted)

Here, we have the common attribute ATM Display, but the value for this attribute is 'ANY' in the second line and hence the whole attribute is discarded, otherwise it would have been substituted with the value in second line.

The next line in Use Case scenario (Figure 14) is: 'ATM asks PIN', the corresponding operation predicate from the domain model (Figure 8) for this Use Cases step is:

Operation: askPIN

Added Condition: User PIN is requested, ATM Display is pin enter prompt

Since there is no withdrawn condition, the added condition is converted to:

User. PIN (requested), ATM. Display (pin enter prompt) ---- line 3

The verification sequence line 2 is

ATM. status (ON), User. ValidationStatus (card inserted) ---- VS line 2

So, the verification sequence, VS line 2 through line 3 becomes:

ATM. Status (ON), User. ValidationStatus (card inserted), User. PIN (requested), ATM. Display (pin enter prompt) --- VS line 3

Here, all the predicate attributes in line 3 gets added to the VS line 2 and since there are no common predicate attributes in line 3 and VS line 2, no substitution occurred in the VS line

3.

Hence, according to the above rules and strategy, the verification sequence for happy scenario (Figure 14): *Precondition- 1-2-3-4-5-6 (postcondition)*, results in:

Verification Sequence:

Pre: ATM. Status (ON), ATM. Display (Welcome)

Operation: ***'insert card'***

ATM. Status (ON), User. ValidationStatus (card inserted)

Operation: ***'ask PIN'***

ATM. Status (ON), User. ValidationStatus (card inserted), User. PIN (requested), ATM. Display (pin enter prompt)

Operation: ***'enter PIN'***

ATM. Status (ON), User. ValidationStatus (pin entered), User. PIN (requested), (ATM. Display (pin enter prompt)

Operation: ***'ask user validation'***

ATM. Status (ON), User. ValidationStatus (bank inquired), User. PIN (requested), ATM. Display (pin enter prompt)

Operation: ***'check user id'***

ATM. Status (ON), User. ValidationStatus (bank responded), User. PIN (requested), ATM. Display (pin enter prompt), User. Id(valid)

Operation: ***'display operation menu'***

ATM. status (ON), User. ValidationStatus (bank responded), User. PIN (requested), ATM. Display (Operation menu), User. id (valid)

Figure 19: Verification Sequence for PSV

The expected postcondition in predicate format for the happy scenario in Figure 14 is:

Pre: ATM. status (ON), ATM. Display (Welcome)

Post: ATM. Status (ON), ATM. Display (Operation menu)

The last line (postcondition from PSV verification sequence) in the verification sequence is (Figure 19):

ATM. status (ON), User. ValidationStatus (bank responded), User. PIN (requested), ATM. Display (Operation menu), User. id (valid).

The two postcondition statements are compared. It can be seen that the verification sequence conforms to the informal postcondition because all the predicates in the *expected* postcondition is implied in the postcondition of the verification sequence. It can also be noted that the verification sequence's postcondition also contains some additional predicates namely *User. ValidationStatus (bank responded), User. PIN (requested) and User. id (valid)*. The presence of such predicates is because of the inconsistencies present in the domain model operations. So, the implementation of *PSV* system states that these additional predicates should be modified in the respective operations of the domain model to maintain consistency with requirements. The *PSV* system also provides the designer with some suggestions for achieving this goal by commenting on the withdrawn conditions to be included for necessary operations (please refer section *4.1.8*). Although the PSV system provides suggestions on inconsistencies, the PSV process relies also on the judgement of the developer to correct the operations which led to inconsistencies The *PSV* system also helps in identifying the invariants for scenarios by detecting the predicate that is common in all the lines of verification sequence, for instance , in the ATM example, the predicate, *ATM. Status (ON)* is an invariant. The output of *PSV* system will yield a perfect Domain model free of any contradictions and inconsistencies.

4.1.8 PSV Output Analysis and Correctness of domain model from PSV Verification

The outputs obtained from *PSV* process falls under the following three categories:

Case 1:

All the *expected* postcondition predicates are strictly contained in the actual *PSV* postconditions (*Ver_Seq)*

Case 2:

All or any of the *expected* postcondition predicates are missing in the *Ver_Seq* or causes contradictions in attribute values between *Ver_Seq* and *expected* postcondition predicates

Case 3:

All the predicates in expected postcondition and Ver_Seq are equal

Case 1 suggests that the verification is a success, but the domain model contains some inconsistencies. These inconsistencies are caused by the presence of 'additional' predicates in the *Ver_Seq*.

For example (this example is taken from Chapter 6-Case study) in Figure 20:

Input file 1:

Precondition: PMSystem.status(ON), User.status (NOT logged in), User. CardStatus(irregular)
Postcondition : Timeout(20sec), UserCard (NOT inserted), User (NOT logged in), PMSystem(ON)

Input file 2:

Pre : PMSystem.status(ON), User.status(NOT logged in)

Use Cases step 1a: insertcard
User. Card(inserted), User.CardStatus(irregular)

Use Cases step 1a1: startSystemStatusAlarm
PMSystem.Alarm(System status), PMSystem.Display(ANY)

Use Cases step 1a2: ejectCardAfter20sec
Timeout(20 sec), User.Card(NOT inserted), PMSystem.Alarm(ANY), User.Identification(ANY) User. NumberOfAttempts(ANY)

Output:

Verification Sequence:
VS line 1:
PMSystem.status(ON), User.status(NOT logged in)
VS line 2:
PMSystem.status(ON), User.status (NOT logged in), User.Card (inserted), User. CardStatus(irregular)
VS line 3:
PMSystem.status(ON), User.status(NOT logged in), User.Card(inserted), User. CardStatus(irregular), PMSystem.Alarm(System status)

VS line 4:

Timeout(20 sec), PMSystem.status(ON), User.status (NOT logged in), User.Card(NOT inserted), User. CardStatus(irregular), PMSystem.Alarm(System status)

Result of verification:

The actual obtained postcondition is :

Timeout(20 sec), PMSystem.status(ON), User.status(NOT logged in), User.Card(NOT inserted), User. CardStatus(irregular), PMSystem.Alarm(System status)

The desired postcondition is:

User.Card(NOT inserted), Timeout(20sec), User (NOT logged in), PMSystem.status(ON)

The verification is successful because the actual postcondition satisfies the desired postcondition.

Invariant for the Use Cases:

1. PMSystem. status(ON)
2. User. status(NOT logged in)

Report of suggestions:

The domain model is inconsistent with the user requirements because of the additional predicates, User.CardStatus(irregular), PMSystem.Alarm(System status)

The reasons may be :

1. Adequate withdrawn conditions are missing in input file operations lines 1a2.

Figure 20: PSV verification output for inconsistent domain model

The above *PSV* verification output (Figure 20) suggests that the verification is successful, but the domain model is inconsistent due to the presence of additional predicates, *User. CardStatus(irregular)* .Also, the *PSV* report suggests that 'adequate *WithdrawnConditions* are missing in input file domain operation corresponding to line *1a2*'.

This is a typical example of *case 1*. The *PSV* module takes care of the inconsistency by tracing back the *PSV* process until it reaches the step which caused the additional predicate. Once this step is identified, it proceeds to the domain operation of the next step and tries to provide a new *WithdrawnCondition* for the *concept_attribute* along with the

existing *WithdrawnCondition* set for that domain operation. So, in the *PSV* process, this new *Withdrawncondition* eliminates the additional predicate which caused the inconsistency.

In the above example (Figure 20), the inconsistency was caused by the additional predicates, *User.CardStatus(irregular)* and *PMSystem.Alarm(System status).* The predicate, *User.CardStatus(irregular)* was introduced by the domain operation at scenario step, 1a. Hence, a new *WithdrawnCondition*, *'ANY ON User CardStatus'* is introduced for the domain operation corresponding to step *1a1*. Once this condition is introduced, the predicate, *User. CardStatus (ANY)* will be identified as a *WithdrawnCondition* and will be eliminated by the *PSV* process. If this new WithdrawnCondition is not introduced, the value of *User.Card status* remains the same for subsequent operations which in turn can cause contradictions for some operations. For instance, after the execution of domain operations for step *1a2*, the status of the user card still remains *irregular.* It may not be the case that the user card should still be irregular and the PM system Alarm is System status (ON) after the card is ejected. *PSV* process identifies these conditions and suggests improvements to the domain model by 'trace back' mechanism as described above. The PSV process preserves consistency and correctness in domain model. This case is further analyzed in Chapter 6.

Case2 causes the PSV verification to be unsuccessful because of the absence of some or all of predicates in *expected* postcondition in *Ver_Seq*.

For example (Figure 21): (this example is taken from Chapter 6-Case study)

Input file 1:

Precondition: PMSystem.Display(MENU), User. status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient)

Postcondition: Patient.Status(monitoring) , Doctor.status(assigned)

Input file 2:

Precondition: PMSystem.Display(MENU), User. status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient)

Use Cases step 1: choosePatientAdmissionfunction

Patient.Status(admission initiated), PMSystem.Display(ANY)

Use Cases step 2: promptsDoctorName
Doctor.Name(asked), PMSystem.Display(doctor name prompt)

Use Cases step 3: enterDoctorName
Doctor.Name(entered), PMSystem.Display(doctor name)

Use Cases step 4: promptsVitalSigns
PMSystem.Display(vital signs prompt)

Use Cases step 5:entersVitalSigns)
Patient.Status(vital signs entered), PMSystem.Display(vital signs)

Use Cases step 6: connectsCables
Patient.Status(INPatient)

Use Cases step 7: startsPatientMonitoring
Patient.Status(monitoring)

Output:

Verification Sequence:

VS line 1:
PMSystem.Display(MENU), User.status(logged in), PMSystem.status(ON),
Doctor.Decision(admitPatient)
VS line 2:
User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient),
Patient.Status(admission initiated)
VS line 3:
User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient),
Patient.Status(admission initiated), Doctor.Name(asked), PMSystem.Display(doctor name
prompt)
VS line 4:
User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient),

Patient.Status(admission initiated), DoctorName(entered), PMSystemDisplay(doctor name)

VS line 5:

User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient), Patient.Status(admission initiated), Doctor.Name(entered), PMSystem.Display(vital signs prompt)

VS line 6:

User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient), Doctor.Name(entered), Patient.Status(vital signs entered), PMSystem.Display(vital signs)

VS line 7:

User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient), Doctor.Name(entered), Patient.Status(INPatient), PMSystem.Display(vital signs)

VS line 8:

Patient.Status(monitoring), User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient), PMSystem.Display(vital signs)

Result of verification:

The actual obtained postcondition is :

Patient.Status(monitoring), User.status(logged in), PMSystem.status(ON), Doctor.Decision(admitPatient), PMSystem.Display(vital signs)

The desired postcondition is:

Patient.Status(monitoring) , Doctor.status(assigned)

The verification is not successful because the actual postcondition differs from the desired postcondition.

Report of suggestions:

The domain model needs to be reconsulted on the predicate 'Doctor(assigned)'

The reasons may be :

1. Adequate added conditions are missing in some operations

Figure 21: PSV verification output for unsuccessful verification

The *PSV* output above is unsuccessful because the *logical predicates* in the to *expected* postcondition, *Doctor.status(assigned),* is missing from the actual obtained *PSV* postcondition:

'Patient.Status(monitoring),User.status(loggedin),PMSystem.status(ON),Doctor.Decision(admitPatient), PMSystem.Display(vital signs)' . In this case, the *PSV* report suggests that the domain model should be consulted to check for the missing predicate, *Doctor.status(assigned).*

Case 3 is the situation when *PSV* verification is successful and the domain model is consistent since all the *logical predicates* in *expected* postcondition is equal to the *logical predicates* in *Ver_Seq.*

For example (Figure 22): (this example is taken from Chapter 6-Case study)

Input file 1:

Precondition : Patient.Status(monitoring), PMSystem.status(ON), Patient.VitalSigns(out of physiologic limits)
Postcondition : PMSystemAlarm.status(ON), Patient.Status(monitoring), PMSystem.status(ON)

Input file 2:

Pre : Patient.Status(monitoring), PMSystem.status(ON), Patient.VitalSigns(out of physiologic limits)
Use Cases step 1: readVitalSigns Patient.MonitoringStatus(vital signs reading), Patient.VitalSigns(ANY), PMSystem.Display(ANY)
Use Cases step 2a: checkVitalSigns Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(out of physiologic limits)
Use Cases step 2a1: startPatientStatusAlarm PMSystemAlarm.status(ON), Patient.VitalSigns(ANY), Patient.MonitoringStatus(ANY)

Output:

Verification Sequence:

VS line 1:
Patient.Status(monitoring), PMSystem.status(ON), Patient.VitalSigns(out of physiologic limits)
VS line 2:
Patient.MonitoringStatus(vital signs reading), Patient.Status(monitoring), PMSystem.status(ON)
VS line 3:
Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(out of physiologic limits), Patient.Status(monitoring), PMSystem.status(ON)
VS line 4:
PMSystemAlarm.status(ON), PMSystem.status(ON), Patient.status(monitoring)

Result of verification:
The actual obtained postcondition is :
PMSystem.Alarm(Patient status), PMSystemAlarm.status(ON), PMSystem.status(ON)

The desired postcondition is:
PMSystem.Alarm(Patient status), PMSystemAlarm.status(ON), PMSystem.status(ON)

The verification is successful and the domain model is consistent with requirements because the actual postcondition is exactly the desired postcondition with invariants.

Invariant for the Use Cases:
1. PMSystem.status(ON)
2. Patient.Status(monitoring)

Figure 22: *PSV verification output for successful verification/consistent domain model*

The *PSV* output from the above example shows successful verification and no domain model inconsistencies because the *logical predicates* in both *expected* postcondition and Ver_Seq are the same, '*PMSystem.Alarm(Patient status), PMSystemAlarm.status(ON), PMSystem.status(ON)'.*

4.1.9 Implementation details

The *PSV* approach is implemented as an add-in to the Use Case based Requirements Engineering tool-UCed. UCed follows a similar kind of semi-formal NL representation for

Use Cases and domain model as used in PSV process. The Use Case capture and composition was performed by the Use Case edition tool for UCed. The Domain model was also generated by the Domain model edition tool of UCed. The extraction of scenarios and predicate conversion was done by a NL based parsing mechanism in Prolog. The *PSV* verification is done in Prolog by using facts and rules.

The Use Cases are entered in a field descriptor format by the Use Case edition tool. The preconditions and post conditions for frequently occurring scenarios of each Use Case is recorded also in UCed. The domain model edition tool keeps track of all the concepts, concept attributes, and operations in terms of added conditions and withdrawn conditions. The parsing mechanism in Prolog takes the Use Cases and precondition/postcondition of scenarios as input and converts it into the corresponding logical predicates by mapping it to the domain model. The Prolog *PSV* module, implemented in SWI-Prolog verifies the scenarios and domain model as described by the PSV algorithm. *The snapshots of PSV outputs are shown in Chapter 6 (Case study).*

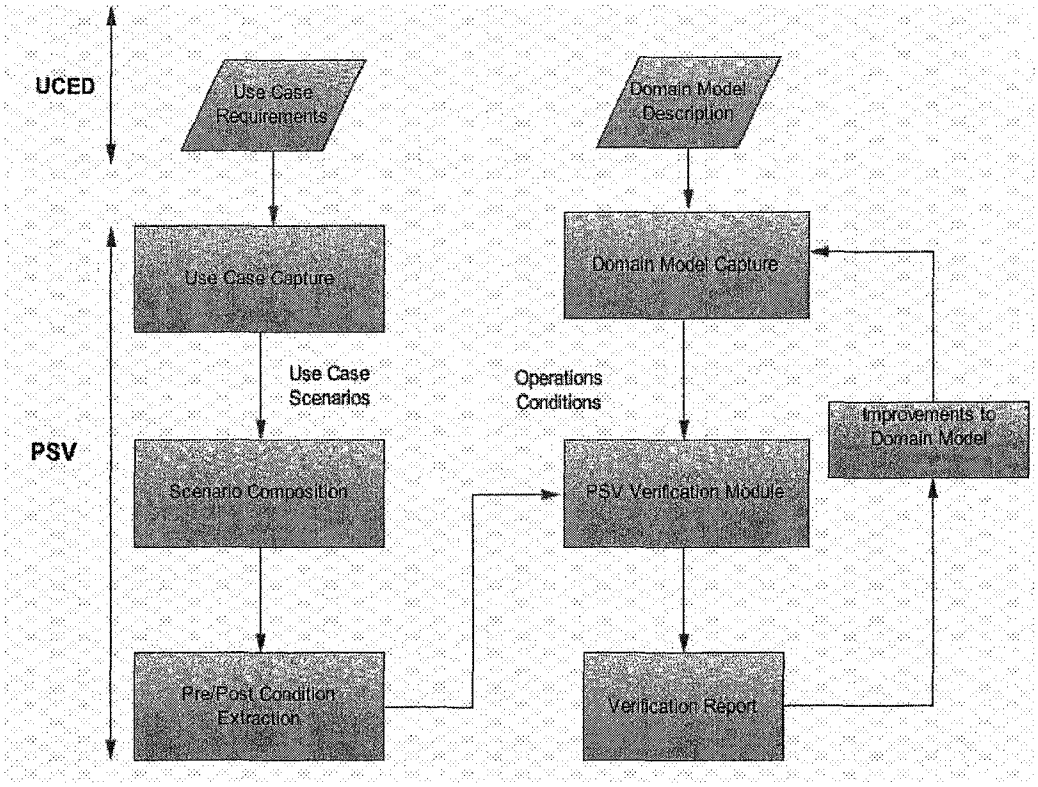


Figure 23: Diagrammatic representation of PSV implementation

The above block diagram (Figure 23) describes the implementation details.From the above diagram (Figure 23), it can be seen that the strategy used for use case composition and the domain model composition used for the PSV process are extracted from the UCed tool. The PSV module performs:

A. Extraction of preconditions and respective postconditions of the use case scenarios from the composed use cases in UCed and converts the extracted pre/postconditions of the scenarios to logical predicate representation
B. Conversion of domain model descriptions obtained from UCed tool to logical predicate representation
C. Verification of postconditions (logical predicate format) from the PSV process against the *expected* postconditions (logical predicate format) of the respective use case scenarios.
D. Generation of the Verification Report and suggestions for improving the domain model in case of model inconsistencies.

A case study is proposed in *Chapter 6* to further analyse and illustrate the *PSV* process elaborately.

4.2. An Alternative Proof based strategy for Scenario Verification

4.2.1. Scenario Sequential Verification (SSV)

The objective in the proposed proof based approach is provided as a future enhancement to logically prove the Use Case scenarios. The Program Transformation [23] method used for logical verification of programs is adopted to describe this proof based strategy. Program Transformation is used for verifying whether a program fragment satisfies the informal user requirements. Program Transformation methods convert program fragment to a logical representation and this logical program fragment are proved using an appropriate proof calculus to verify that the program fragment does not contradict the informal user requirements. The style of verification is termed Sequential Verification [50] (termed "Sequential Verification" due to the sequential nature of verification of scenarios) where the verification proceeds sequentially (please refer to section *3.3.2* of Chapter 3). This scenario verification method based on Program Transformation is termed as *'Scenario*

Sequential Verification (SSV) Strategy'. The proof calculus used for SSV strategy is a modified Hoare-triple proof style. A Hoare triple as described in section *3.3.3,* expects a program fragment to have a state before the execution of program fragment and another state after the execution of the program fragment. The body of the program fragment consists of set of executable steps which performs some kind of function. Similarly, a scenario has a precondition, denoted by φ set of operational steps denoted by S, postcondition denoted by Ψ; the scenario can be represented by the Hoare triple:

$(|\varphi|)\ S\ (|\Psi|)$

The proof based strategy for *PSV* method can be expressed as:

1. Extract the informal requirements, I, to be expressed in the predicate logical formula, Ψ_I
2. Express the sequential set of conditions pertaining to the domain operations of the scenario steps to satisfy Ψ_I, expressed in terms of sequential set of logical predicates, S
3. Prove that S will satisfy Ψ_I

4.2.1.1. Hoare Proof Calculus

The calculus for Hoare triples was suggested by C.A.R. Hoare and hence called Hoare logic. The proof rules for Hoare logic include simple rules involving assertions to more complex and compound rules of composition.

In this approach, the scenarios are devoid of any iterations and complex conditions as mentioned earlier. Therefore, a subset of Hoare calculus rules are only required from the Hoare proof calculus for the verification of scenarios.

The rules used for verifying scenarios are:

- Rule of Sequential Composition
- Rule of Assignment
- Implied rule
- Propositional rules

4.2.1.1.1 Rule of Assignment

Hoare [18] states that "Assignment is undoubtedly the most characteristic feature of

programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic."

As suggested by Hoare, the rule of assignment is a simple rule involving one triple and is devoid of any premises.

The rule of assignment is stated as:

$$\frac{}{(|\ \Psi\ [A/x]|)\ x = A\ (|\ \Psi\ |)}$$

Ψ [A/x] is the formula obtained by taking Ψ and replacing all free occurrences of x with A. This means Ψ [A/x] is Ψ with A in the place of x. This rule is mostly applied backwards which means that Ψ is already known, then we replace x in Ψ with A, the resulting formula becomes A (| Ψ |).

This is represented in tableau proof as:

(| Ψ [A/x]|)

x=A

(| Ψ |) Assignment

The arrow pointing upwards indicates that this rule is applied in the backward direction. (please refer section *3.3.1* of Chapter 3 for top-down and bottom-up strategies)

Example:

(|Z +Y = X+Y|) Assignment

Z = Z + Y

(| Z = X +Y|)

4.2.1.1.2 Rule of Sequential Composition

The rule is represented as:

$$\frac{(|\ \varphi\ |)\ S1\ (|\ \mu\ |) \qquad (|\ \mu\ |)\ S2\ (|\ \Psi\ |)}{(|\ \varphi\ |)\ S1;\ S2\ (|\ \Psi\ |)}$$

This rule is otherwise known as 'compound rule' since it deals with more than one triple. The basis for proving the scenarios is composition. If two operational steps of scenario are considered as S1 and S2 represented by the triples:

$(| \varphi |)\ S1\ (| \mu |)$ and $(| \mu |)\ S2\ (| \Psi |)$

Then if the postcondition of S1 is the precondition of S2, the rule of sequential composition states that:

$(| \varphi |)\ S1; S2\ (| \Psi |)$

This rule is best applied to the bottom up proving style described in section *3.3.1* of Chapter 3, hence a suitable postcondition of S1 represented by ' μ ' is discovered and the sub goals, $(| \varphi |)\ S1\ (| \mu |)$ and $(| \mu |)\ S2\ (| \Psi |)$ have to be proved. As soon as the goal S1 with the precondition, φ, is executed, the postcondition, μ, is obtained and this is the precondition to S2 which when executed will result in Ψ, the desired postcondition of the main goal.

Example: If there are two fragments

$\{y=B \wedge x=A\}\ t:=x\ \{y=B \wedge t=A\}$

And

$\{y=B \wedge t=A\}\ x:=y;\ y:=t\ \{x=B \wedge y=A\}$

Then by using the rule of sequential composition, the result will be:

$\{y=B \wedge x=A\}\ t:=x;\ x:=y;\ y:=t\ \{x=B \wedge y=A\}$

4.2.1.1.3 Implied rule

The implied rule states that:

$$\frac{\vdash_E \varphi' \rightarrow \varphi \qquad (| \varphi |)\ S\ (| \Psi |) \qquad \vdash_E \Psi \rightarrow \Psi'}{(| \varphi' |)\ S\ (| \Psi' |)}$$

According to the implied rule, if the formula, φ' implies φ and Ψ implies Ψ', and the triple, $(| \varphi |)\ S\ (| \Psi |)$, is already proved, then $(| \varphi' |)\ S\ (| \Psi' |)$ can also be proved. The validity of the sequent, $\vdash_E \varphi' \rightarrow \varphi$ is proved by using the proof calculus of natural deduction by taking φ and other arithmetic rules as premises. The implied rule strengthens

the precondition rather than postcondition. This is better because since the actual postcondition is to be checked with the desired postcondition, the actual postcondition is to be deduced from the precondition. So we start with the precondition and prove towards deriving the postcondition, which means strengthening of precondition. If the postcondition is strengthened than the precondition, that is, if we try to start with the postcondition to imply the precondition, then it may result in erroneous conclusions.

Example of implied rule:

$|-_{par}$ (|y =5|) x = y + 1 (|x = 6|)

(| y = 5|)

(|y + 1 = 6|) Implied by general mathematical rule

x = y +1

(| x=6|) Assignment

The backward arrow suggests that the rule is applied for backward style proving.

4.2.1.2 Propositional rules

The proof based SSV strategy also uses propositional logic rules in proving the PSV process.

The main propositional logic rules used are:

- Rules of Conjunction
- Rules of disjunction
- Rules of Implication

Conjunction rules:

1. and-introduction (/\i)

A B

---------- /\i

A /\ B

A and *B* are formulas which are premises [39] for the rule. If there are two formulas *A* and *B*, then by and-introduction rule *A* /\ *B* can be deduced.

2. and-elimination (/\e1 and /\e2)

```
   A /\ B                    A /\ B
---------- /\e1          ---------- /\e2
     A                         B
```

If there exists a proof for *A* /\ *B*, then by using /\e1 *A* can be deduced and by using /\e2, *B* can be deduced.

Example for /\i, /\e rules using tableau proof:

1. a /\ b premise
2. s premise
3. b /\e2, 1
4. s /\ b /\i 3, 2

Disjunction rules:

1. or-introduction

```
   A                         B
---------- \/i1          ---------- \/i2
  A \/ B                    A \/ B
```

If there are any premises, *A* or *B*, then *A\/B* can be deduced

Implication rules:

1. Implication elimination (->e)

```
   A    A->B
   ---------- ->e
       B
```

This rule is otherwise called Modus Ponens. It states that if there is the formula *B* and it is known that *A*-> *B*, then *B* can be deduced.

2. Modus Tollens

$$\frac{A \rightarrow B \quad \neg B}{\neg A}\ MT$$

If A implies B, A->B and B is not true, then it can be deduced that A cannot be true.

Example for implication rules by tableau proof:

1.	a-> (b->c)	premise
2.	a	premise
3.	¬c	premise
4.	b->c	->e 1,2
5.	¬ b	MT 3, 4

4.2.1.3 Scenario Sequential Verification (SSV) method

The SSV method uses tableau proof style for proving the *PSV* process. The rule of sequential composition is the inspiration behind the tableau proof style.

A scenario, S can be viewed as:

S1;

S2;

……..

Sn;

Where S1, S2 … Sn are the scenario steps representing domain operations.

According to the tableau proof style for Hoare triples, the following sequent should be proved:

$$\vdash_{tot} (| A |)\ S\ (|B|)$$

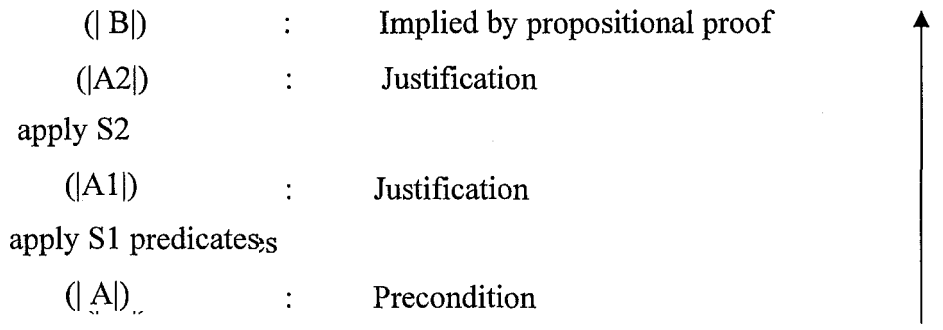
Where *A* and *B* represents the preconditions and postconditions of the scenario repectively.

According to the *PSV* process, the verification commences by extracting postconditions (*AddedConditions* and *WithdrawnConditions*) from the domain model that are *in logical predicate* representation corresponding to the scenario steps *S1, S2 .. Sn*. Then the *logical*

predicates representing each step are 'passed' sequentially to obtain the actual postcondition of the Use Case scenario. Finally, the actual postcondition is matched with the desired postcondition of the scenario to obtain the verification result.
Similar to this *PSV* verification process, the *SSV* method starts with the precondition *logical predicates* and does the 'passing' mechanism in *PSV* based on rule of sequential composition and finally deducts the actual postconditions of the scenario by using rules of Hoare calculus and propositional logic.

The result of applying the domain operation predicates of first scenario step yields a set of 'temporary' postcondition predicates which acts a sub goal. Then the domain operation predicates of the next scenario are applied to this result by rule of assignment and rule of sequential composition and the proof continues until the domain operation predicates corresponding to the last scenario step are applied. The result of applying the last set of operation predicates reveals the actual postcondition by the Implied rule based on propositional logic rules. A separate propositional proof is derived to imply that the actual obtained postcondition is same as that of the required postcondition of the Use Case scenario.

The structure of the proof is as follows:
The scenario is: (|A|) S1; S2; S3; ..... Sn; (|B|) where A and B are precondition and postcondition formulas and S1, S2..Sn are scenario steps.
If the scenario has only two steps, that is, two segments, then the proof strategy is:



The backward arrow indicates that the proof proceeds in the backward proving style. Sequential Verification uses backward proving style because most of the Hoare triple rules

are applied for backward proving styles. The *SSV* method can be illustrated by the ATM example of happy scenario *(1-2-3-4-5-6)* from Figure 14 of Chapter 4.

Prove the sequent:

Precondition

(ATM. Status(ON) /\ (ATM. Display(welcome message))

Operation: ***insert card***

User. Validation status(card inserted)) /\ (ATM. Display(ANY))

Operation: ***ask PIN***

(User.PIN (requested)) /\ (ATM. Display (pin enter prompt))

Operation: ***enter PIN***

(User. Validation status (pin entered)),

Operation: ***ask user validation***

(User. Validation status (bank inquired)),

Operation: ***check user id***

User. Validation status (bank responded)) /\ (User. PIN (requested)),

Operation: ***display operation menu***

(ATM. Display (Operation menu)) /\ (user. id (valid))

$\models$ ***(ATM. status (ON)) /\ (ATM. Display (Operation menu))***

Proof:

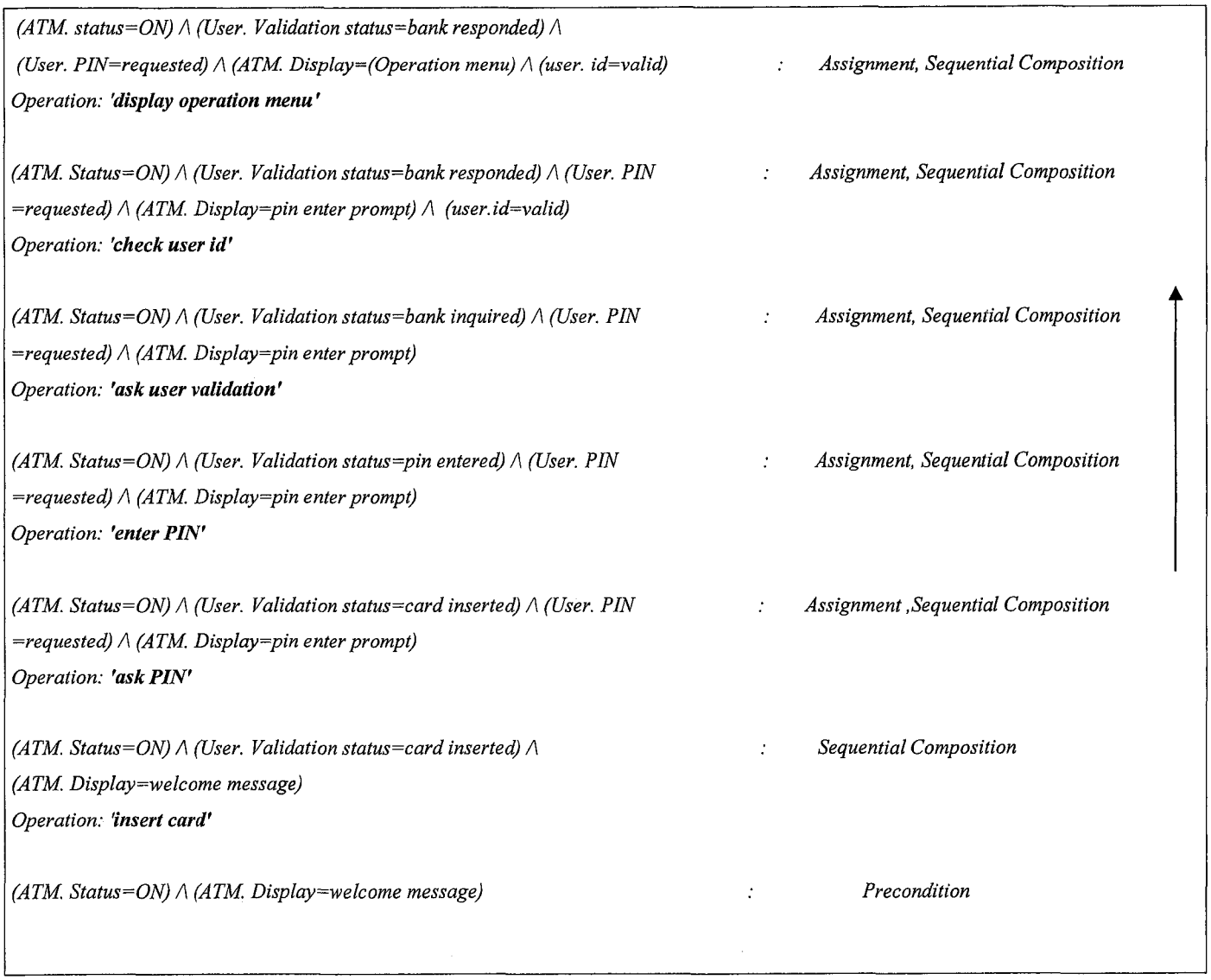


Figure 24: SSV proof

The proof starts in the backward proving strategy by taking the precondition and applying the next operational predicate segment to the precondition by using the rule of assignment and sequential composition to obtain the resultant predicates. The proving proceeds by applying subsequent operation segments until the final 'resultant set' of predicates is reached which represent the postcondition obtained from PSV process.

The final PSV postcondition predicates are proved further to check whether the PSV postcondition imply the Use case postcondition. This proving strategy is devised by using a propositional proof where the PSV postcondition and precondition form the premises and

the UCL-post (*expected* postcondition from the use case scenario) forms the proof output. Propositional proof to imply the postcondition of the verification sequence imply the postcondition of the Use Cases scenario, that is :

Prove the sequent:

[(ATM. status=ON) /\ (ATM. Display=welcome message)],

[(ATM. Status=ON) /\ (User. Validation status=bank responded) /

(User. PIN=requested) /\ (ATM. Display=Operation menu) /\ (user. Id=valid)]

|= (ATM. Status=ON) /\ (ATM. Display=Operation menu)

Figure 25: Propositional proof for SSV postcondition/desired postcondition verification

1. (ATM. Status =ON) /\ (ATM. Display=welcome message)	:	*Premise*
2. (ATM. Status=ON) /\ (User. Validation status=bank responded))/\ (User. PIN =requested) /\ (ATM. Display=Operation menu) /\ (user. Id=valid)	:	*Premise*
3. (ATM. Status=ON)	:	*/\e1,1*
4. (ATM. Display=Operation menu)	:	*/\e2,2*
5. (ATM. Status=ON) /\ (ATM. Display=Operation menu)	:	*/\i,3,4*

4.3 Summary of the chapter

In most systems, construction of a domain model from informal requirements is based on assumptions and perspectives of the designer because the informal requirements are never complete. This calls for the necessity to verify the domain model against the requirements to ensure that the model does not stray from requirements [29]. The domain model is supposed to be the backbone of design since the whole system is dependent on it at later stages of development. Hence, the PSV system proposes a promising improvement in this earlier design verification and suggests modifications on the domain model to obtain consistency with requirements. The PSV system also identifies the invariants present in the scenario. Since , the PSV verification passes the precondition through the whole steps of scenario, the scenario is indirectly verified to see whether the desired postcondition is satisfied by the scenario steps. The chapter extablishes an alternative method for PSV

process by a logical proving approach based on Hoare logic. The predicate representation used for expressing the conditions automatically gives a more formal nature to the Use Cases thereby trying to bind the formal and informal requirements closer. The PSV process enables the developer to bring out a clearer perspective to domain model descriptions.

Chapter 5. State Machine Verification against Use Case Requirements

In chapter 4, a strategy for verifying domain model against Use Case requirements is discussed. The result of this verification helped to conclude that the scenarios written in semi-formal Natural Language and the domain model are free of basic requirement errors and inconsistencies. This completes a 'phase 1' verification of the early design phase revealing a more consistent domain model. A more detailed formal design is to be generated from the requirements and the domain model in the advanced design stage of software development. Such a formalized design possesses a low degree of abstraction and accommodates more details to the design model. The generation of a formalized design model is important in software development since it eases the final implementation of the system. This chapter discloses a 'phase 2' verification approach to early design phases. A verification strategy is proposed to verify the formal model against the Use Case requirements. The formal design model considered in the proposed State Machine Verification (SMV) approach is the state chart diagram, since it is expressive, formal, comprehensive and more understandable to users and designers. The proposed SMV approach is termed *'Semi-Automated Validation'* [30]. The approach is phrased 'semi-automated' because the extraction of pre/postconditions of the NL based Use Case Requirements is guided by manual intervention (developer/end user).

5.1. Relevance of Semi-Automated Validation

The *Semi-Automated Validation* for SMV discloses a strategy for verifying the state chart diagram for requirement errors and inconsistencies. oMost of the systems usually focus on the implementation errors and discard the design errors due to lack of time and extent of effort involved. The resolution of implementation errors is easier because of the high degree of formalism present in the detailed design. For instance, The CASE tool [4] provides automatic testing methods based on the condition that the design used for generating these test cases are much formalized. According to [8], the automatic generation of test cases is possible only if the design model is formal. The following figure (Figure

26) shows the concept of verification in software development.

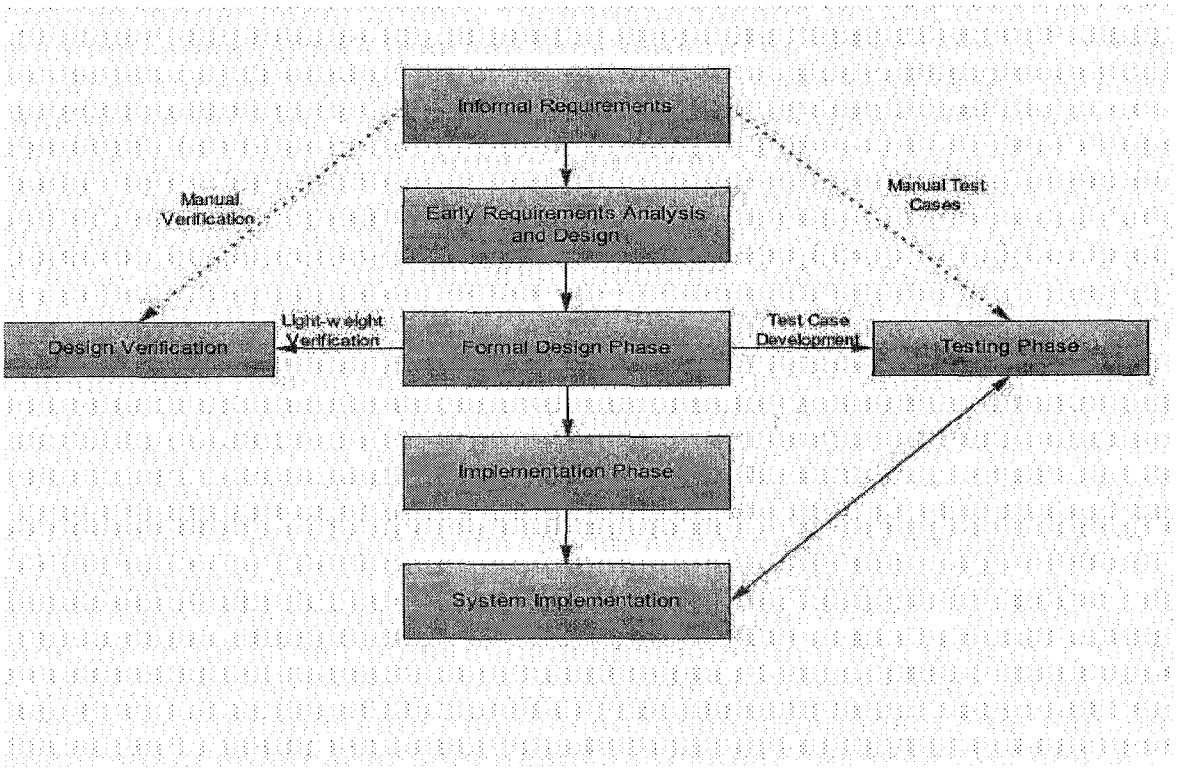


Figure 26: Verification concept in software development

The element of formalism facilitate the automation of testing and verification and hence testing in the implementation phase involves less time and effort, on the other hand, performing the verification only in the implementation phase, in some cases, can prove disastrous and can often lead into a stalemate. The missing picture in the latter verification strategy is the idea that, if the early requirement and design verification phases are subjected to verification, then the frequency and intensity of errors arising from implementation phase testing can be reduced at large.

The *PSV* verification takes care of the early requirement and domain model verifications uncovering requirement contradictions and domain model errors. So, the next focus should be on the detailed formal verification of the design derived from the requirements and domain model. Generating formal designs especially state chart diagrams from requirement is usually based on algorithmic methods and technical assumptions from the designer/developer. Therefore, verifying these formal designs is inevitable to guarantee a

consistent and error free formal model to proceed with implementation. Such a formal model is realized by the phase 2 *Semi-Automated Validation* approach discussed in this chapter.

This formal validation of state chart diagrams is an extension of the *PSV* verification discussed in Chapter 4.While the *PSV* verification yields a consistent and error free domain model, the state chart verification ensures a consistent and error free formal design model (state chart diagram). The whole idea of this verification is based on our work explained in [30].

5.2. The Semi-Automated Validation approach

The objective of this approach is to provide a mechanism for validating a design model represented by state machines, against Use Case requirements. The conversion of Use Cases to state machines is essential to refine system behavior and to keep consistency between requirements and design [37]. The goal is to provide a *semi-automated* validation approach that could be applied in the context of iterative elaboration of state machines from Use Cases. The underlying idea of *semi-automated* Validation is to check for a given precondition/postcondition of a Use Case scenario in the Use Case requirements, the postcondition is satisfied by the corresponding state in a state chart diagram. This validation is possible if there is a specific mechanism to represent the Use Case scenarios and state chart diagram into similar form of logical operational statements. Once this conversion to logical statements is done, the preconditions to be verified will be generated automatically from these Use Case based logical statements. Then the postconditions corresponding to these Use Case based precondition logical statements will be compared against the postconditions of state chart based logical statements.

Diagrammatic representation of the Semi-automated Validation model:

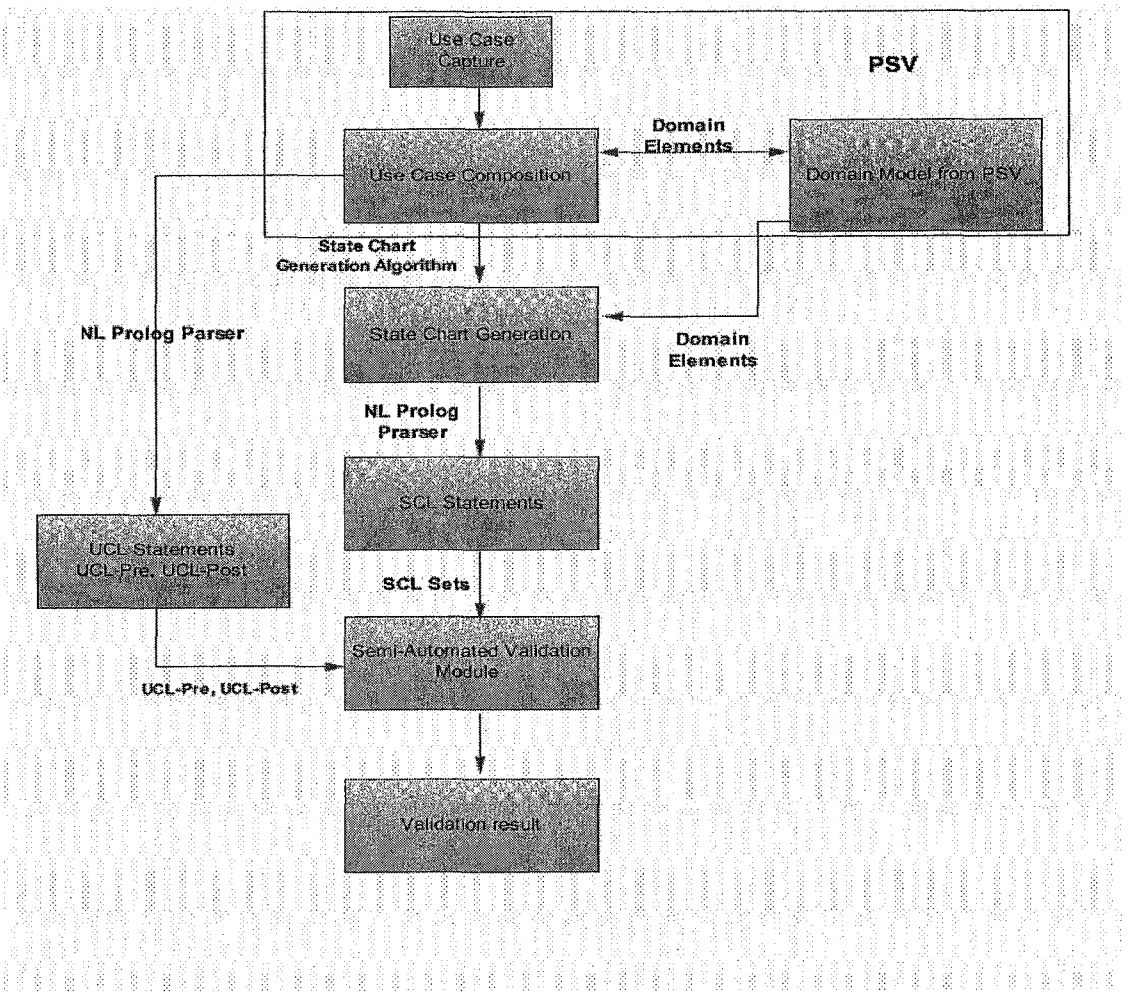


Figure 27: Semi-automated validation process

The SAV architecture is depicted in Figure 27.

The *Semi-automated Validation* process can be divided into the following major sequences:

1. Conversion of Use Case scenarios to Use Case scenario based logical statements. These scenario based logical statements will be referred as 'UCL-pre' and 'UCL-post' corresponding to the scenario's pre and postconditions respectively.
2. Conversion of state chart diagrams to state chart based logical statements. These state charts based logical statements referring to the states will be referred as 'SCL'.

3. Matching the UCL-post statement with the corresponding SCL statement for a given precondition.

The extraction of UCL-pre and UCL-post from the Use Case Scenarios is performed with manual intervention. The logical conversions of UCL-pre, UCL-post and State chart model as well as the generation of SCL Verification Sequence and the Verification process are performed automatically by the Semi-Automated Validation module.

5.2.1 Use Case Representation for Semi-automated Validation

The Use Case representation in *semi-automated* validation process is the same Use Case representation used in the PSV process *(Please refer to sections 3.5.2 of Chapter 3 and section 4.1.2.1 of Chapter 4).*

The following Use Case example for ATM cash withdrawal application is referred in the succeeding sections:

Title: Cash withdrawal
Primary Actor: Bank
Precondition: ATM is ON and ATM display is Welcome message
Postcondition: ATM is ON and ATM display is operation menu.

1. User inserts card in card reader slot
2. ATM asks PIN
3. Before 60 sec, user enters PIN
4. ATM asks user validation to bank
5. Bank checks user id
6. If user identification is valid, ATM displays operation menu
7. User selects cash withdrawal
8. ATM asks withdrawal amount
9. Customer enters withdrawal amount
10. ATM checks customer update
11. If user withdrawal OK, ATM dispenses cash in cash dispenser slot and ATM ejects card
12. User takes cash and ATM displays operation menu

1. a. User card is not valid
   1. a. 1. ATM ejects card
   1. a. 2. ATM displays error message
   1. a. 3. After 30 sec, ATM displays welcome message
2. a. After 60 sec,
   2. a. 1. ATM ejects card
   2. a. 2. ATM displays error message
   2. a. 3. After 30 sec, ATM displays welcome message
5. a. User identification is invalid
   5 a. 1. ATM ejects card
   5. a. 2. ATM displays error message
   5. a. 3. After 30 sec, ATM displays welcome message

10 a. User withdrawal is not OK
   10. a. 1. ATM displays error
   10. a. 2 After 10 sec, Go to step 8

Figure 28: Use Case for cash withdrawal

5.2.1.1 Extraction and 'Logical' conversion mechanism for preconditions and postconditions for Use Case scenarios from the Use Case

The pre/postconditions for Use Case scenarios are extracted from the Use Case and they are converted to a formal representation. The formal representation in semi-automated validation is the termed '*logical predicate*' representation used for *PSV* verification (please refer section *4.1.5* of Chapter 4). After conversion of pre/postconditions to *logical*

predicate representation, they will be known as '*UCL-pre*' and '*UCL-post*' respectively.

Since the Use Case requirements are expressed in a crude NL form which reflects an NL based semi-formal behavior and the state chart is expressed in a formal structure, a common 'logical' representation is inevitable for use case requirements and the state chart model, in order to perform Semi-Automated Validation. Therefore, the 'logical predicate' representation used for the PSV process is adopted to represent the use case requirements and the formal state chart model.

The extraction of UCL-pre and UCL-post from the use cases are performed with manual intervention. This extraction is guided by the following categorization and conversion rules.

The preconditions/postconditions for a Use Case are generated by categorizing Use Case condition steps to fall under three types of situations:

Type1: The *main* preconditions and their respective postconditions of Use Case (primary scenario)

Type2: *Extension* step pre-conditions/ postconditions sequences of Use Case

Type3: *Condition sequences* in Use Case.

Logical* Conversion rules for *Type 1, Type 2 and Type 3 are based on the logical conversion rules described for predicate representation used in the PSV process.

Assumptions:

The conversion rules for Use case scenarios are performed manually by maintaining a naming consistency between the semi-formal NL Use Case requirements and the domain model descriptions for entities (concept/sub concepts/attributes). The entities in the semi-formal NL use case requirements are converted manually by obeying the naming convention used for entities in the domain model description.

For ***Type 1,*** the precondition is the main precondition of the Use Case and the postcondition is the main postcondition of the Use Case. This precondition/postcondition pair corresponds to the 'successful' scenario of the Use Case.

Example: From Figure 28:

Precondition: ATM status is ON and ATM display is welcome message

Postcondition: ATM status is ON and ATM display is operation menu

This pre/post condition pair will be converted to logical predicates as:

UCL-pre: ATM. Status (ON) and ATM. Display (welcome message)

UCL-post: ATM. Status (ON) and ATM. Display (operation menu)

Type 2 scenarios result from *extension steps* of Use Cases (please refer to *section 4.1.2.2* of Chapter 5). For this type, the precondition is extracted as:

|Extension step| | 'AND'| | 'IF'| |Extension step condition|

The postcondition of the scenario will be extracted as:

|Extension sub-step 1| | 'AND'| | Extension sub-step 2| | 'AND'|.....|Extension sub-step'n'|

For example (from Figure 28):

Use Case extension scenario:

1. User inserts card in card reader slot

4. a. User card is NOT valid

1. a. 1. ATM ejects card

1. a. 2. ATM displays error message

1. a. 3. After 30 sec, ATM displays welcome message

The verbs which indicate point of time like 'after', 'before' etc will be converted to the conditional verb 'IF'.

Precondition: *User inserts card in card reader slot and IF User card is NOT valid THEN*

Postcondition: *ATM ejects card and ATM displays error message and IF After 30 sec, ATM displays welcome message*

The above pre/postconditions can be converted to logical predicates as:

UCL-pre: User. ValidationStatus (card inserted) and IF User.Card (NOT valid)

UCL-post: User.ValidationStatus (card ejected) and ATM. Display (error message) and IF TIMEOUT (30sec) THEN ATM. Display (welcome message)

Type 3 does not refer to scenarios, but they refer to conditional statements or clauses in Use Case steps, starting with the keyword 'IF'. It is necessary to include this type for semi-automated validation since there is a possibility of 'conditional' errors in the state machine during the state machine generation process.

Type 3 statements take the general format:

|IF| |condition clause| |THEN or ", "| |effect clause|

The precondition is specified by the '*condition clause*' and the postcondition is specified by the *'effect clause'*

Example: from Figure 28:

If User identification is valid, ATM displays operation menu

Precondition: User identification is valid

Postcondition: ATM displays operation menu

This pre/post pair will be converted to logical predicate representation as:

UCL-pre: User. Identification (valid)

UCL-post: ATM. Display (operation menu)

The following Figure 29 shows the pre/postconditions from the Use Case after conversion:

UCL-pre: ATM. Status (ON) and ATM. Display (welcome message)
UCL-post: ATM. Status(ON) and ATM. Display (operation menu)----------(A)

UCL-pre: User. ValidationStatus (card inserted) and IF User. Card (NOT valid)
UCL-post: User. ValidationStatus (card ejected) and ATM. Display (error message) and
IF TIMEOUT (30sec) THEN ATM. Display (welcome message)---------(B)

UCL-pre: User. PIN (requested) and ATM. Display (pin enter prompt) and IF TIMEOUT(60 sec)
THEN
UCL-post: User. ValidationStatus (card ejected) and ATM. Display (error message) and
IF TIMEOUT (30sec) THEN ATM. Display (welcome message)

UCL-pre: ATM. TransactionStatus (amount checking) and IF User. Identification (invalid)
UCL-post: User. ValidationStatus (card ejected) and ATM. Display (error message) and
IF TIMEOUT (30sec) THEN ATM. Display (welcome message)

UCL-pre: User. Identification (valid)
UCL-post: ATM. Display (operation menu)

UCL-pre: ATM. TransactionStatus (amount checking) and IF User. Withdrawal (NOT OK)
UCL-post: ATM. Display (error message) and IF TIMEOUT (10sec) THEN
ATM. Display(withdrawal amount) -------(C)

Figure 29: UCL-pre/UCL-post for ATM cash withdrawal Use Case

5.2.2 State Machine Representation/Generation for Semi-automated Validation

The formal requirement design model that is of prime interest to this approach is the state transition diagram, termed interchangeably as the state chart or state diagram. A state chart represents one or more Use Cases in terms of states and transitions (please refer section 3.6 of Chapter 3 for elements of a state chart). This state model could be produced manually or automatically from Use Cases and domain model by a state chart generation algorithm.
The formal representation for state charts used in semi-automated validation is represented by a tuple:
[E, S, S0, SN, T]

E represents events which include system concept operations and actor concept operations or timeout conditions when the particular state exceeds the time limit permitted for the state.

S is a set of states

S0ϵ*S* is the initial state of the state model.

SNϵ*S* is the final state of the state model

T is the set of transitions associated with states.

A transition is comprised of a 'start' state which is satisfied before the occurrence of the transition, the event which causes the transition, the 'resultant' state resulting from the transition.

Example: from Figure 32: "1— insert card/ ->2" describes a transition from state 1 to state 2.

Transitions can include guard conditions or trigger/reactions. *Triggers* are operations from actors in the environment and reactions are operations of the system in response to triggers.

Each state is represented by a set of *predicates* and *transitions*, if any.

For example: State, S6 from Figure 32 is:

S6 [ATM status is ON, User Validation status is card ejected, User PIN is requested, ATM Display is error message]

There are four predicates for state 6, *ATM status is ON, User Validation status is card ejected, User PIN is requested,* and *ATM Display is error message*

A state can have any of these three conditions:

1. Only *one transition* based on *a single* set of transition conditions (*normal* state), E.g. state S6 in Figure 32.
2. *More* than one transition, each transition specified by a single set of transition conditions (*compound* state), E.g.: state S2 in Figure 32
3. No transitions (*simple* state), E.g.: state S13 in Figure 32

The following Figure 30 represents a *compound* transition:

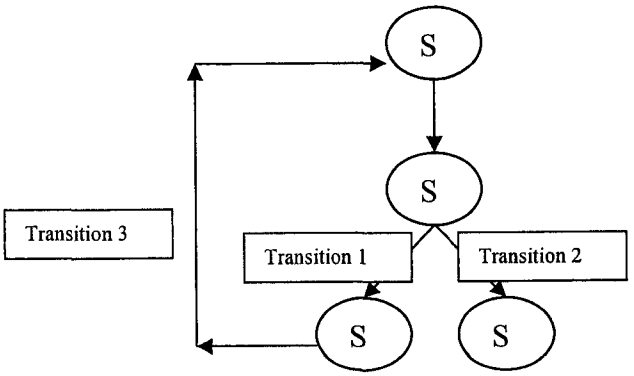


Figure 30: A compound transition representation in semi-automated validation process

A graphical representation for state chart diagram used in *semi-automated validation* for the ATM cash withdrawal example (Figure 28) is shown below [30] in Figure 31:

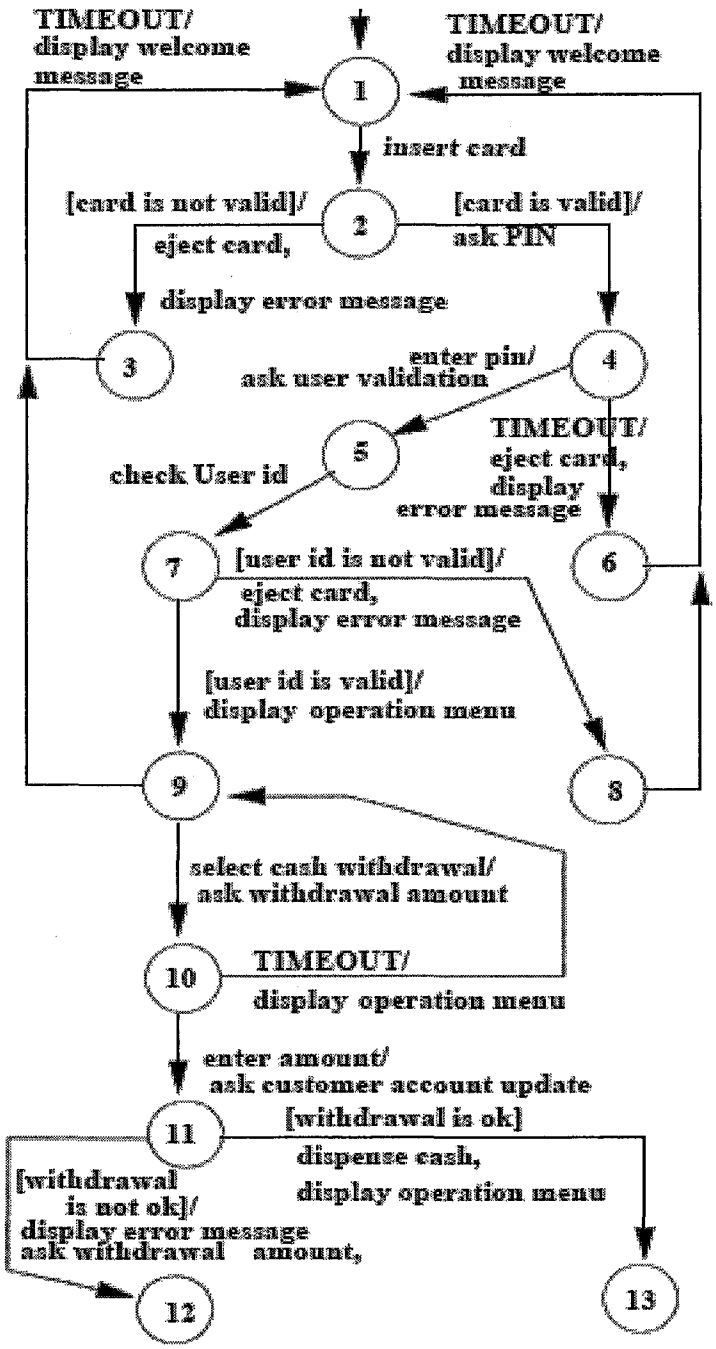


Figure 31: State chart for ATM cash withdrawal

For convenience in describing state chart conversion to *logical predicate* representation, a documented (textual) form of the above graphical state chart for *ATM cash withdrawal* Use Case is shown in the Figure 32 below:

S1: [ATM status is ON, ATM Display is welcome message]
S2: [User Validation status is card inserted, ATM status is ON]
S3: [Timeout:30.0 second, ATM status is ON, User Validation status is card ejected, User Card is NOT valid, ATM Display is error message]
S4: [ATM Display is pin enter prompt, ATM status is ON, User PIN is requested, User Validation status is card inserted]
S5: [ATM Display is pin enter prompt, ATM status is ON, User PIN is requested, User Validation status is bank inquired]
S6: [ATM status is ON, User Validation status is card ejected, User PIN is requested, ATM Display is error message]
S7: [ATM status is ON, User PIN is entered, User Identification is checking]
S8: [ATM status is ON, User Validation status is card ejected, User Identification is invalid, User PIN is requested, ATM Display is error message]
S9: [ATM Display is operation menu, ATM status is ON, User Identification is valid]
S10: [ATM status is ON, ATM Display is withdrawal amount, User Transaction is cash withdrawal]
S11: [ATM status is ON, User Transaction is cash withdrawal, User withdrawal amount is entered, ATM Transaction status is amount checking]
S12: [ATM Transaction status is amount checking, User withdrawal is NOT ok, ATM status is ON, ATM Display is error message, User Transaction is cash withdrawal]
S13: [ATM Display is operation menu, ATM Transaction status is cash dispensed, ATM status is ON, User Validation status is card ejected, User Transaction is cash withdrawal]

Set of Transitions:
1-- insert card/ ->2
2--[user card is not valid]/eject card, display error message->3
2-- [user card is valid]/ ask PIN->4
3--[TIMEOUT (Timer:30.0 second)]/display welcome message-->1
4--enter pin /ask user validation-->5
4--[TIMEOUT (Timer:60.0 second)]/eject card, display error message-->6
5--check User id/-->7
6--[TIMEOUT (Timer:30.0 second)]/display welcome message-->1
7--[user id is valid]/display operation menu->9
7--[user id is not valid]/eject card, display error message->8
8--[TIMEOUT (Timer:30.0 second)]/display welcome message-->1
9---select cash withdrawal/ask withdrawal amount-->10
10---enter amount/ask customer account update-->11
11--- [User withdrawal is NOT ok]/display error message -->12
11--- [User withdrawal is ok]/dispense cash, eject card, display operation menu-->13

Figure 32: Textual representation of state chart for ATM cash withdrawal

5.2.2.1 State chart conversion to 'logical predicate' representation

The state chart should be converted into the same logical representation as the UCL pre/postconditions. Basically, it is possible that any state diagrams irrespective of how it is generated, will consist of characteristic predicates, which describe each state and transitions associated with the states. So, it is possible that the following conversion strategy could be used to translate any state chart diagram into a more formal logical format for validation since semi-automated validation approach describes states in terms of predicates and transitions.

The predicates representing each state should always have the verbs in 'to be' or 'to have' forms. Hence, the same conversion rules used for Use Case pre/postconditions along with some additional rules described below can be applied to states in state chart to obtain their *logical* representation. Since there are no pre/postcondition elements in state chart, the logical conversion is applied to each state to obtain state chart based statements called *SCL statements.*

Since the states have transitions, additional SCL conversion rules are to be specified for states. There are three conversion rules described for the three conditions specified above for a state (*simple* state, *compound* state, *normal* state)

The conversion strategy can be summarized to:

1. Each state of the state chart and the associated SC transitions, if any, are converted to SCL statements sequentially.

2. If there is an SC transition associated with a state, then the state will be converted to SCL along with its SC transition condition. This SC transition condition will be written by using 'IF' clause in the conversion. Then, the next immediate SCLs will be the conjunction of *logical* representations of resulting states of that transition scenario until an 'already visited' state or a resulting state with no transitions (*simple* state) is reached.

3. If there is more than one SC transition associated with a state, then the first transition with its sequence is converted to SCL followed by the conversion of the succeeding transitions.

5. In case of *compound* state where many course of actions are possible depending on the different transitions, the SCL conversion will take into consideration the *predicates* associated with the state and the transitions one at a time in a sequential style.

6. In the situation, where a state, *s3* looping back to an already visited state, *s1* on transition, the transitions for *s1* will not considered for SCL conversion since *s1* is an already visited state or an 'already visited' state.

Conversion rule 1 for *simple* state:

A *simple* state is devoid of any transitions. So, the predicates which specify the state will be converted to logical representation according to *logical conversion rule 1 for type 1 (main pre/postconditions)*

For example: from Figure 32:

State 13, S13 does not have any transitions in the *transition list.*

S13: [ATM Display is operation menu, ATM Transaction status is cash dispensed, ATM status is ON, User Validation status is card ejected, User Transaction is cash withdrawal]

S13 is described by five predicates namely:

- *ATM Display is operation menu*
- *ATM Transaction status is cash dispensed*
- *ATM status is ON,*
- *User Validation status is card ejected*
- *User Transaction is cash withdrawal*

So, the SCL conversion of S13 will be:

ATM.Display(operation menu) and ATM.TransactionStatus(cash dispensed) and ATM. Status(ON) and User.ValidationStatus(card ejected) and User.Transaction(cash withdrawal)

Conversion rule 2 for *normal* state:

For a normal state, the following type of logical predicates will be joined to yield the final logical conversion of the state:

1. logical predicates corresponding to the state itself
2. logical predicates of the transition condition
3. logical predicates of the resulting state arising from the transition

For example: from Figure 32,

The *normal* state, S3, is described by the predicates:

[Timeout:30.0 second, ATM status is ON, User Validation status is card ejected, User Card is NOT valid, ATM Display is error message]

For State 3 (S3), there are five predicates:

- Timeout:30.0 second
- ATM status is ON
- User Validation status is card ejected
- User Card is NOT valid
- ATM Display is error message

So the logical state chart based statement (SCL) for these predicates yields:

TIMEOUT(30sec) and ATM. Status (ON) and User. Validation Status(card ejected) and User. Card (NOT valid) and ATM. Display (error message)

But, we have to take care of the State chart transition event associated with the state, S3.

In the transition list from Figure 32, there is a single transition event associated with state 3 namely:

3-- TIMEOUT (30.0 second)/display welcome message-->1

This means that on state 6 when the condition 'TIMEOUT (30.0 second)' is applied, it results in state 1, S1. In SC transition, the condition for the transition to occur is the one defined before '/' i.e. in the above example; the transition condition is the occurrence of event "TIMEOUT (30.0 second)".

In the SCL conversions, the transition condition is preceded with the keyword **"IF"**.

So, in the final conversion of state 3 to *logical* statement, the result will be as:

TIMEOUT(30sec) and ATM. Status(ON) and User. Validation Status(card ejected) and User.Card (NOT valid) and ATM. Display(error message) and IF TIMEOUT(30sec)

Now, the next SCL statement immediately after the above SCL statement will be the converted SCL statement corresponding to state 1 which is:

ATM. Display (welcome message) and ATM. Status (ON)

In this state 1, there is still a transition associated with it. But this transition is not taken into consideration for conversion because state 1 is an *already visited* state (a state resulting from a 'backward' transition of another state), in other words there is a 'backward' loop formed from S3 to S1.

So, the next SCL statement will be corresponding to state 1 without the state 1 transition.

ATM. Display (welcome message) and ATM. Status (ON)

We can summarize the above situation of SCL statement corresponding to state 3 to:

15. *TIMEOUT(30sec) and ATM.Status(ON) and User. Validation Status(card ejected) and User.Card (NOT valid) and ATM.Display (error message) and IF TIMEOUT(30sec)*

16. *ATM.Display (welcome message) and ATM.Status (ON)*

[*Please note that the numbering of the above SCL statements corresponds to the SCL verification sequence described in Figure 35*]

So, for the state 3 having a single transition, the transition results in deriving the ***sequence***, '15→16'

Conversion rule 3 for compound state:

Since the *compound state* has more than one transitions associated with it, each transition is converted to *logical predicate representation* using *conversion rules 1 or 2*. All the transitions associated with the state will be considered sequentially and will be converted to *logical representation* individually. For each transition of the state, the SCL conversion will consider all the resulting states until an *'already visited state'* is reached.

For example if we consider state 2, S2 from Figure 32:

For the *compound* state, s2:

s2 is represented by the characteristic predicate condition:

[User Validation status is card inserted, ATM status is ON]

The number of transitions for the state:

The state, S2 has two transitions associated with it in the *transition list.*

These transitions are:

2—[user card is not valid]/eject card, display error message->3

2--- [user card is valid]/ ask PIN->4

So, to convert state S2 to SCL, we have to club and "and" the predicates of state 2 with the transition conditions.

Since there are two transitions associated with state 2, we have to take care of each of the transitions separately and sequentially.

Hence, the converted SCL for state 2 for the **first transition** will be:

ATM.Status(ON) and User.ValidationStatus(card inserted) and IF User.Card(NOT valid)

The next SCL statement will be the one corresponding to state 3 which is the result of the first transition*(2—[user card - not valid]/eject card, display error message->3)*

[Please note that state 3 has a transition namely:

3 -TIMEOUT (Timer:30.0 second)/display welcome message-->1]

So the SCL conversion for the next SCL statement will be :

ATM.Status(ON) and User.ValidationStatus(card ejected) and User.Card(NOT valid) and ATM Display(error message) and IF TIMEOUT(30sec)

The next SCL statement will be the SCL corresponding to state 1:

ATM.Status(ON) and ATM.Display(welcome message)

It is important to note that though state 1 has a transition, it will not be considered for conversion because the transition is looped back to state 1 which is the 'already visited' state.

The above situation of State 2 conversion to SCL for the **first transition** *(2—[user card - not valid]/eject card, display error message->3)* can be summarized to:

4. ATM.Status(ON) and User.Validation Status(card inserted) and IF User.Card(NOT valid)

5. ATM.Status(ON) and User.ValidationStatus(card ejected) and User.Card(NOT valid) and ATM.Display (error message) and IF TIMEOUT(30sec)

6. ATM.Status(ON) and ATM.Display(welcome message)

[*Please note that the numbering of the above SCL statements corresponds to the SCL verification sequence described in Figure 35*]

So for state, s2, the first transition results in the sequence ***'4→5→6'***

Now, we have to take care of the second transition for state 2 in the similar manner.

5.2.3 Assumptions for Semi-automated validation

Assumption 1:

This semi-automated validation approach is based on the assumption that Use Cases and state machines are syntactically consistent; that is a same terminology is used in both models.

Assumption 2:

Use Cases should be written in a semi-formal Natural Language format according to the representation described in *section 3.5.2* of Chapter 3.

Assumption 3:

The use of iterations is not allowed in the semi-formal Use Case representations.

Assumption 4:

The extraction of pre/postconditions of Use Case scenarios from Use Cases and the logical representation of these pre/postconditions (UCL-pre and UCL-post) are based on the rules described in *section 5.2.1.1* in this Chapter

Assumption 5:

The formal representation of state chart and, the extraction and conversion of SCL statements from states of the state chart are based on the rules described in *section 5.2.2.1* in this Chapter

Assumption 6:

The formal state chart generated from UCed has formal characterization of states. Each state is described by set of predicates and transitions.

Assumption 7:

The *initial* state of the state chart denotes the precondition of the Use Case and the *final* state of the state chart represents postcondition of the Use Case on successful execution.

5.2.4. State chart conversion algorithm

The algorithm for state chart conversion is shown in Figure 33:

```
Procedure Generate_SCL_spec_state chart (S: state chart)
Begin
Mainpre[] = generate_SCL (s1:state1, S: state chart);
n= count (S: state chart);
Mainpost[] = generate_ SCL (sn: last state, S: state chart);
Print (mainpre, mainpost);
For each unvisited state (s ϵ S) and (s! = sn) do
 Begin   /* generating Sequence for state with transitions*/
    If (SCtransitions! = null)
      Begin
       Begin
        SCcount = number_of_trans (s);
        I = 0;
        While (SCcount! = I )
          Begin
           SCL[] = generate_SCL (s, S) U generate_SCL_trans (s, S,SC);
            SCL_Next[] = generate_SCL_resultant (s, S, SC);
            Print (SCL[], SCL_next[]);
            I = I +1;
            p = visited(s);
          End while
       End   /* end generating Sequence for state with transitions*/
    Else if (SCtransitions = null)
       Begin  /* state with no transitions ('leaf' state) */
         SCL[] = generate_SCL (s, S);
          p = visited(s);
         Print (SCL[]);
       End   /* end state with no transitions ('leaf' state) */
    End if
 End for
End
```

Figure 33: State chart conversion algorithm

Sub-Procedure	Parameters	Return value	Description
generate_SCL	s1:state1 S: state chart	MainPre[]	Generates the logical predicates for the first state in the state chart
generate_SCL	sn: state n S: state chart	MainPost[]	Generates the logical predicates for the last state in the state chart corresponding to the result of successful execution contributed by successful scenarios
generate_SCL	s: current state S: state chart	SCL[]	Generates the logical predicates for the current state, s, in the state chart
generate_SCL_trans	s: current state S: state chart SC: transition list	SCL[]	Generates the logical predicates for the transition conditions associated with the state,s, in the state chart
generate_SCL_resultant	s: current state S: state chart SC: transition list	SCL_Next[]	Generates the logical predicates for the resultant state occurring from the transition of state,s, in the state chart

Table 5: Description of sub-procedures in state chart conversion Algorithm

The state chart generation algorithm generates SCL statements for all the states in the state chart in a 'depth-first traversal' fashion. The algorithm works by deriving *Sequences* for each state resulting from transitions associated with the state represented by the sequential numbering of the consecutive SCL statements.

The whole state chart can be visualized as a *directed tree* graph where each node represents each state in the state chart and each labeled directed edge denotes a transition.

For example, the following Figure 34 represents a state chart with a total number of 11 states:

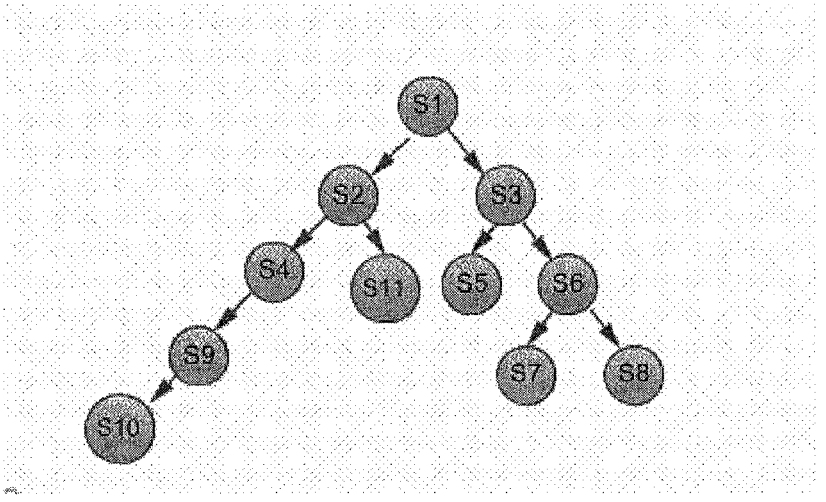


Figure 34: *Graphical Tree representation of state chart*

The state chart represented by the above Figure 34 generates the SCL statements for each state in a depth first traversal mode. The SCL conversion for this state chart commences by taking the first unvisited node, s1 denoting the state s1. The state s1 has two child nodes representing states, s2 and s3 with respective transitions along the edges. As described in the conversion rules, the first transition will be first taken into consideration for SCL conversion. The first transition results in state s2, even though s2 has two transitions resulting in s4 and s11, only the first transition resulting in s4 will be considered since the sequences are generated in a depth first fashion. So the state 1, s1 with the first transition, results in the ***sequence 's1→s2→s4→s9→s10'***. By next, the other transition for state s1 is considered resulting in the ***sequence 's1→s3→s5'***. Since all the transitions for s1 are considered for conversion, the state, s1 is marked as visited. The same process is repeated for the state s2 and the algorithm stops when all the states in the chart are visited. The SCL conversion for a node which has no children (leaf node) representing the state with no transitions (leaf state) will comprise of only the SCL predicates representing the state.

So the sequences resulting from the state chart depicted in figure 9 will yield the following sequences from SCL conversion.

Sequence 1: s1→s2→s4→s9→s10

Sequence 2: s1→s3→s5

Sequence 3: s2→ s4→s9→s10

Sequence 4: s2→s11

Sequence 5: s3→s5

Sequence 6: s3→s6→s7

Sequence 7: s4→s9→s10

Sequence 8: s6→s7

Sequence 9: s6→s8

Sequence 10: s9→s10

The procedure "*generate_SCL*" generates the SCL corresponding to a state using the NL based Prolog parser according to the above described conversion rules and conventions. The set of SCL statements obtained after the conversion of the whole state chart will be stored in a *'SCL verification sequence'*.

The algorithm takes two types of control sequences, the first control sequence is executed for states with transitions and the second control sequence is executed for states with no transitions (leaf state).

According to assumption 7, the first state of the state chart denotes the precondition and the last state denotes the postcondition of the Use Case on successful completion. So the procedure, *generate_SCL (s1:state1, S: state chart)*, will generate the *first* SCL statement in the *SCL verification sequence*. Similarly, the same procedure with a different parameter for last state, *generate_SCL (sn: last state, S: state chart)*, will generate the *second* SCL statement in the *SCL verification sequence.*

The number of transitions associated with a state is stored in the variable "*SCcount*". For each state of the state chart, the respective control sequence is executed depending on whether the state has transitions or not. If a state, s, has transitions, then for each transition

associated with the state, the predictes corresponding to the state s is generated by the procedure, *generate_SCL (s: state, S: state chart)* and is clubbed (union 'U') with the predicates corresponding to the transition written by the keyword 'IF' (*see conversion rules section 5.2.2.1*), by executing the procedure, *generate_SCL_trans (s: state, S: state chart, SC: transition list)*. The next SCL will be the resultant transition state in the state diagram if it has no transitions associated with it. Otherwise this process is repeated until a state with no transitions is reached or an already visited state is reached (*see conversion rules section 5.2.2.1*). This particular function is implemented by the procedure, *generate_SCL_resultant (s, S, SC)*.

For example state 2 in the above state diagram (Figure 32) consists of 2 courses of actions namely:

2—[user card-not valid]/eject card, display error message->3

2--- [user card – valid]/ ask PIN->4

In this case the corresponding SCL predicates for state 2 will be "anded" with the first transition condition predicates "[user card is not valid]" (see translation rules for compound state in section *5.2.2.1*). The next SCL will be the SCL conversion of state 3 which is the resultant state of transition resulting from state 2. Since s3 has a transition associated with it, the SCL predicates for s3 is clubbed with the SCL predicates of the transition separated by the keyword 'IF'. Finally, this transition of s3 results in state 1 which is an already visited state. So, the next immediate SCL statement will be the SCL predicates corresponding to s1 without its transition predicates (please refer to the conversion strategy for an already visited state) thus completing the first transition Sequence for state 2. After this the next transition 2--- [user card – valid]/ ask PIN->4 is considered similar to the first transition.

If there are no SC transitions associated with a state, then that state indicates a leaf state and the SCL conversion for this state comprises of SCL predicates representing that state.

By using the above algorithm (Figure 33), the result of the SCL conversion of the state chart generated above (Figure 32) is shown in Figure 35:

1. ATM. Status (ON) and ATM. Display (welcome message)
2. ATM. Display (operation menu) and ATM. Status (ON) and User. Validation status (card ejected) and ATM. Transaction status (cash dispensed) and User. Transaction (cash withdrawal)
3. ATM. Status (ON) and ATM. Display (welcome message) and if User. Validation status (card inserted)
4. ATM. Status (ON) and ATM. Display (welcome message) and if User. Validation status (card inserted) and if User. Card (NOT valid)
5. ATM. Status (ON) and User. Validation status (card ejected) and User. Card (NOT valid) and ATM. Display (error message) and IF TIMEOUT (30sec)
6. ATM. Status (ON) and ATM. Display (welcome message)
7. User. Validation status (card inserted) and ATM. Status (ON) and if User. Card (valid)
8. ATM. Display (pin enter prompt) and ATM. Status (ON) and User. PIN (requested) and User. Validation status (card inserted) and IF User. Validation status (pin entered)
9. ATM. Display (pin enter prompt) and ATM. Status (ON) and User. PIN (requested) and User. Validation status (bank inquired) and IF User. Identification (being checked)
10. ATM. Status (ON) and User. PIN (entered) and User. Identification (checking) and IF User. Identification (valid)
11. ATM. Display (operation menu) and ATM. Status (ON) and User. Identification (valid) and IF User. Transaction (cash withdrawal)
12. ATM. Status (ON) and ATM. Display (withdrawal amount) and User. Transaction(cash withdrawal) and IF User. withdrawal amount (entering)
13. ATM. Status (ON) User. withdrawal amount (entered) and User. Transaction (cash withdrawal) and ATM. Transaction status (amount checking)IF User. withdrawal (NOT ok)
14. ATM. Display (error message) and ATM. Status (ON) and User. Transaction (cash withdrawal) and User. withdrawal (NOT OK) and ATM. Transaction status (amount checking)
15. TIMEOUT (30sec) and ATM. Status (ON) and User.Validation status (card ejected) and User. Card(NOT valid) and ATM. Display (error message) and IF TIMEOUT (30sec)
16. ATM. Status (ON) and ATM. Display (welcome message)
17. ATM. Display (pin enter prompt) and ATM. Status (ON) and User. PIN (requested) and User. Validation status (card inserted) and IF TIMEOUT (60sec)
18. ATM. Status (ON) and User. Validation status (card ejected) and ATM. Display (error message) and IF TIMEOUT (30sec)
19. ATM. Status (ON) and ATM. Display (welcome message)
20. ATM. Status (ON) and User. PIN (entered) and User. Identification (checking) and IF User. Identification (NOT valid)
21. ATM. Status (ON) and User. Validation status (card ejected) and ATM. Display (error message) and User. Identification (NOT valid) and User. PIN (requested) and IF TIMEOUT (30sec)
22. ATM. Status (ON) and ATM. Display (welcome message)
23. ATM. Status (ON) User. withdrawal amount (entered) and User. Transaction (cash withdrawal) and ATM. Transaction status (amount checking) and IF User. withdrawal (NOT ok)
24. ATM. Display (error message) and ATM. Status (ON) and User. Transaction (cash withdrawal) and User. withdrawal (NOT OK) and ATM. Transaction status (amount checking)

23.....

Figure 35: SCL verification sequence

5.2.5. The algorithm for matching

The validation is automatically performed:

Once all the conversion of Use Case scenario pre/postconditions and state chart diagram to *logical* representations is completed, the validation of the logical statements is automatically performed. The logical format of Use Case scenarios and state chart diagram are consistent (see Figure 35 and therefore it is much easier to perform validation automatically. For each UCL-pre, the corresponding SCL statement is retrieved by a matching technique using the algorithm discussed below. The immediate ***next SCL*** (*next SCL* is actually the postcondition from the state chart) is verified to be the same as that of the UCL-post corresponding to the UCL-pre. If it is found to be the same, then the validation proceeds with the next UCL-pre. In case, if the postconditions do not match each other, the validation is interrupted by allowing the developer to see the generated validation inconsistency document. The inconsistency document details the erroneous UCL-pre, UCL-post and SCL.

Basically, for a given precondition from the Use Cases, we check that the corresponding UCL-post is implied in the respective SCL.

That is:

(UCL-pre → UCL-post) ∧ (SCL-pre → SCL-next) ⇔ (UCL-post → SCL-next)

[SCL-next means SCL-post since the next logical statement after the SCL under reference (SCL-pre), corresponds to the postcondition]

Here, for a given UCL-pre, we try to search for the SCL statement that corresponds to this particular UCL-pre. Therefore, UCL-pre and SCL-pre are the same.

Hence the logical expression can be simplified as:

(Pre→UCL- post) ∧ (Pre→SCL- next) ⇔ (UCL-post → SCL-next)

By mathematical reasoning, it can be observed that:

((x>2) → (x>1)) ∧ ((x>2) → (x>0) ⇔ ((x>1) and (x>0))

Here (x>1) and (x>0) can never be contradictory.

It is possible that the *next SCL* may contain more logical predicates than needed by the Use Case requirements. These additional logical predicates are contributed by the formal design method used to generate the state chart. The critical part to observe here is that all the expressions stated in the UCL-post should be implied by the corresponding *next SCL* statement or in other words included in the SCL.

So it can be deduced based on the above observation:

(Pre-> UCL- post) /\ (Pre -> SCL- next) => (UCL-post /\ SCL-next)

This means UCL-post and SCL-next can never be contradictory and SCL-next includes UCL-post.

```
Procedure generate_matching (S: state chart; M: Use Case)
  Begin
    IN1[]= Genrate_UCL_spec_scenario (M)
    IN2[]= Generate_SCL_spec_state chart (S)
    For each precondition, p in IN1
      Pre[]=Findcorrespre (IN2[],IN1[],p);
        Scenpost[] = scencheckpost (p, IN1[]);
          Statepost[] = statecheckpost ( Pre, IN2[]);
            If (statepost[] = includes (scenpost[])) then
                  Print ("no ambiguity");
             Else
                  Print ("ambiguity", Pre[], scenpost[], statepost[]);
  End
```

Figure 36: Algorithm for matching

Sub-Procedure	Parameters	Return value	Description
Findcorrespre (IN2[],IN1[],p)	IN1[]: UCL-pre/UCL-post from use case scenarios (edited manually by developer/user) IN2[]: SCL Verification Sequence p: UCL-pre	Pre[]	for each of the *UCL-pre,* the corresponding *SCL* statement is found
scencheckpost	IN1[]: UCL-pre/UCL-post from use case scenarios (edited manually by developer/user) p: UCL-pre	Scenpost[]	retrieves the UCL-post statement corresponding to the UCL-pre, p
statecheckpost	IN2[]:SCL Verification Sequence Pre[] : SCL corresponding to UCL-pre,p	Statepost[]	retrieves the **next SCL** state chart statement corresponding to the SCL state chart statement in *'Pre'*

Table 6: Description of sub-procedures in matching Algorithm

Figure 36 shows the algorithm for matching SCL and UCL.
The procedure *"Genrate_UCL_spec_scenario"* will generate maually all the UCL-pre/UCL-postconditions from the Use Case as described in Figure 29. The procedure *"Generate_SCL_spec_state chart"* will generate the SCL statements according to the state chart conversion algorithm described in *section 5.2.4*. The output of this procedure will be a SCL verification sequence as displayed in Figure 35.

According to the above algorithm, for each of the *UCL-pre*, the corresponding *SCL* is found by using the procedure *"Findcorrespre"* and this SCL statement will be stored in parameter, *'Pre'*. It is important to note that all the predicates in UCL-pre should be present in the respective SCL but not vice versa. The procedure *"scencheckpost"* will retrieve the UCL-post statement corresponding to the UCL-precondition, p and the procedure "statecheckpost" will retrieve the **next SCL** state chart statement corresponding to the SCL state chart statement in *'Pre'*. If all the predicates in the UCL-post are included in the **next SCL**, then there is no requirement violation.

Using the logic of the above algorithm, we have
If we take instance (A) in Use Case logical model (Figure 29),
The *UCL-pre* is:

UCL-pre: ***ATM. Status (ON) and ATM. Display (welcome message)***

It is found that the *first SCL* in Figure 35,
1. ***ATM. Status (ON) and ATM. Display (welcome message)***
matches with all the expressions in this *UCL-pre*.
So, the next step is to check the corresponding *UCL-post* with the *next SCL*.
Therefore we have the corresponding *UCL-post* (Figure 29) as:
UCL-post: ***ATM. Status (ON) and ATM. Display (operation menu)***
matches with the *next SCL* (Figure 35) namely:

*5. **ATM. Display (operation menu)** and **ATM. Status (ON)** and User. Validation status (card ejected) and ATM. Transaction status (cash dispensed) and User. Transaction (cash withdrawal)*

It can be seen that the matching turned out to be a success and hence the verification is a success. That is, the *UCL-post* is included in the *next SCL.*

Let us consider instance (B) from Figure 29:

*UCL-pre: **User. Validation status (card inserted) and IF User. Card (NOT valid)***

The matched *SCL* (Figure 35) will be

1. *ATM. Status (ON) and ATM. Display (welcome message) and if **User. Validation status (card inserted)** and **if User. Card (NOT valid)***

Now according to the algorithm, the *UCL-post* (Figure 29) is matched with the *next SCL* (Figure 35).

*UCL-post: **User. Validation status (card ejected) and ATM. Display (error message) and IF TIMEOUT (30sec) THEN ATM. Display (welcome message)***

Is matched with the *next two SCLs* (Figure 35) namely:

*5. ATM. Status (ON) and **User. Validation status (card ejected)** and User. Card (NOT valid) and **ATM. Display (error message)** and **IF (TIMEOUT (30sec)***

*6. ATM. Status (ON) and **ATM. Display (welcome message)***

In this context, a sequence of states is matched, this is because the 5-th SCL has an 'IF' key word *IF (TIMEOUT (30sec)* which reveals that there is an SC transition associated with this SCL. Hence the resultant state is also considered according to the procedure, *generate_SCL_resultant.*

Example of requirement violation:

If we consider the instance C (Figure 29):

*UCL-pre: **ATM. Transaction status (amount checking) and IF User. withdrawal (NOT OK)***

Is matched with the SCL

23. *ATM. Status (ON) User. withdrawal amount (entered) and User. Transaction (cash withdrawal) and **ATM. Transaction status (amount checking)** and **IF User. withdrawal***

(NOT ok)

The matching of UCL-post yields:

UCL-post: ***ATM. Display (error message) and IF (TIMEOUT (10sec)*** *THEN* ***ATM. Display (withdrawal amount)***

Is matched with the next SCL namely:

24. ***ATM. Display (error message)*** *and ATM. Status (ON) and User. Transaction (cash withdrawal) and User. withdrawal (NOT OK) and ATM. Transaction status (amount checking)*

In this context, the "next" SCL is devoid of the expression "**IF TIMEOUT (10sec)"** THEN **ATM. Display (withdrawal amount)"** which means that in case if the user withdrawal amount is invalid, it generates the error message but it will not ask the user to enter the amount again after 10 seconds. This is a **clear case of requirement violation** since this requirement is expressed clearly in the Use Case scenario.

In this way all the pre/postconditions of the Use Case scenarios extracted from the Use Case are validated against the states of state chart diagram to guarantee that the generated state chart diagram does not contradict or violate the basic requirements.

5.2.6 Output Analysis of Semi-automated validation

Semi-automated validation checks whether the state chart conforms to the Use Cases requirements. So the output of *semi-automated* validation reveals that, if any of the predicate conditions of the Use Case scenario's required postcondition *(UCL-post)* is not present in the state chart postcondition *(next SCL),* then the semi-automated validation module deducts that the state chart does not satisfy the Use Case requirements. The causes of errors and inconsistencies in state charts are briefed in the introduction of this chapter. So, irrespective of causes for error occurrences in state charts, the semi-automated validation module detects any errors which cause the state chart to stray away from Use Case requirements.

Complexity Analysis:

The SCL conversion algorithm iterates for the number of times equal to the number of states in the state chart. Hence the complexity of the algorithmic operation for SCL conversion is:

$O(V+T)$

Where *'V'* denotes the whole list of nodes representing all the states in the state chart and *'T'* represents the whole list of edges representing the transitions in the state chart.

The total number of Sequences that are generated from the algorithm will be approximately less than or equal to $O((V-l)+T)$ where *'l'* represents the leaf states representing the states with no transitions.

Time complexity for the state chart verification (SCL conversion algorithm + matching algorithm) yields:

$$n(T(generate\_SCL) + T(generate\_SCL\_trans) + T(generate\_SCL\_resultant)) + O(2n)$$

where *'n'* represents the node list denoting states, *T(generate_SCL)* represents the time needed for the execution of the procedure generate_SCL, *T(generate_SCL_trans)* represents the time needed for the execution of generate_SCL_trans, *T(generate_SCL_resultant)* represents the time needed for the execution of generate_SCL_resultant. O(2n) is obtained by O(n) time needed for visiting all the states in the SCL conversion algorithm along with O(n) time needed for searching the SCL statement corresponding to the UCL-pre in the matching algorithm.

State space explosion problem:

The state space explosion problem is a problem concerning Semi-Automated Validation (SAV) since SAV is a model based verification approach. The state space is known to grow rapidly beyond the available memory space and other resources as the complexity of the model increases. As the number of states grows, the performance of SAV is affected because the generation of SCL statement from state chart is based on a modified depth-first fashion. SAV process generates all available sequences from the generated state chart; hence as the number of states grows, the possibility of encountering infinite sequences cannot be neglected.

5.3. Minimal Guarantees

This formal state chart verification approach succeeds to provide the minimal guarantees of the Use Case in addition to validating the formal design. Minimal guarantees are invariants

that should be true in all action sequences of Use Case. The very first two SCL, namely the main precondition and the respective postcondition of the Use Case, are analyzed to obtain minimal guarantees. The logical predicates which are expressed similarly in these first two SCL are the invariants; they provide the minimal guarantee that all through the Use Case, these predicates are not violated.

In our example of the cash withdrawal Use Case, the first two SCL (Figure 35) have *ATM. Status (ON)* predicate present commonly. So, whatever be the Use Case behavior, this condition should be true in all situations of cash withdrawal.

5.4 Implementation details

The state chart generation and Use Case composition are performed by using the Use Case based requirements engineering-UCed tool [85]. UCed gives the picture of whole requirements engineering process starting from requirements elicitation and provides a technical convenience for explaining and demonstrating semi-automated validation. UCed tool is intended to convert the informal Natural Language requirements to a formal design in an automated fashion. The informal requirements are represented in a restricted form of Natural Language and the formal design used is the state chart diagram. UCed provides a set of tools for performing Use Case capture, Use Case composition, state chart generation, and simulation. The idea behind UCed is that the Use Cases are combined in the state chart diagrams using domain knowledge. UCed captures a group of interrelated Use Cases expressed in a restricted Natural Language format and derives formal specifications by combining the Use Cases' partial behaviors [85]. The UCed process starts with rough Use Cases and domain model and produces the formal state chart specifications. The derivation of the formal state chart diagram is done by extracting information from a high level domain model. The domain model used in UCed is similar to a UML class diagram. The reason for using UCed tool's state chart generation is because UCed takes the Natural Language requirement statements in the form of Use Cases as input, compose a domain model based on requirements and developers assumptions and then algorithmically generate state chart automatically from the requirements and domain model. Since the generation of state chart diagram from UCed is based on algorithm, the generated state chart is to be verified for its errors and inconsistencies. UCed lacks a complete detailed

formal verification module. The proposed formal verification approach explained in this chapter can aid the full state chart verification of UCed state chart. This state chart verification approach can also be used as standalone module provided the state chart's template for verification should follow the UCed state chart template. The extraction of UCL-pre and UCL-post from the semi-formal Use Case and the SCL statements from the formal state chart are performed according to the conversion rules and algorithms described in section *5.2* of this chapter. The semi-automated validation is performed by a semi-automated validation module based on Prolog facts and rules.

5.5. Integrating Use Cases

The idea of Use Case formal verification considers Use Cases individually. Use Cases are written individually according to the demands of the user. The objective of the formal *semi-automated* validation is to check that, each of these Use Cases is satisfied by the respective formal state chart model for that Use Case. With the help of a tool like UCed, it is possible to compose the formal state chart for each of these Use Cases separately. Our approach verifies each Use Case (each Use Case is written and composed by the developer as exactly specified by the user) against its formal state chart design. But, if the state charts of all the Use Cases need to be integrated and verified formally against the Use Case requirements, then it is necessary to manually integrate all the Use Cases written in semi-formal Natural Language to one single Use Case semi-formal Natural Language representation model. If this manual integration of all the Use Cases is possible, then the proposed approach can be used to generate the *logical* statements for the integrated Use Case model and the integrated state chart model for formal SMV verification.

5.6. Summary of the chapter

Formal Model Verification is essential to ensure that the design preserves the basic specifications correctly without any compromise. The Semi-Automated Validation achieves this goal by automatically generating all the scenarios sequences from the formal state chart model and performing automated verification of the postconditions of the generated scenarios against the semi-formal NL based Use Case requirements. The output report of the Semi-Automated Validation module assures that the constructed formal model is free of any requirement violations.

Chapter 6. Case Study - Patient Management System

Patient management systems are used in hospitals to regulate the activities concerning the consultations and monitoring of patients. The three major activities for a patient management system (PM System) are:

1. Patient Admission
2. Monitoring patient's vital signs (Patient Monitoring Systems-PMS)
3. Patient discharge

When the doctor concludes the patient's condition to be serious, the patient is admitted. Once the patient is admitted, the vital signs are monitored by a patient monitoring system, which is an essential part of the patient management system. The admitted patient's (in-patient) vital signs are monitored by analog devices connected to the patient monitoring system. It reads factors like blood pressure, temperature, pulse etc. on a periodic basis and informs the nurse/doctor in case of abnormal readings.

6.1 Requirements analysis and Verification Case study on Patient Management System (PM System)

6.1.1 PM System Requirements

The PM System controls treatments to patients and admits patients in case of emergencies. A patient possesses a medical record containing patient's personal information, previous medical history, a list of consulted doctors, treatment history and insurance information. When a patient consults doctor, his or her's medical history is updated to include the new treatment information. After consultation with doctor, if the doctor demands the patient to be admitted, then the patient gets the status of in-patient, otherwise the patient is considered as an out-patient. The system keeps track of which doctors and nurses are assigned to patients and in what locations of the hospital, in case of in-patients. If the patient is not admitted (out-patients), the doctor gives prescription details to the patient. A doctor, nurse or administrative clerk can admit a patient using authorized access to the system, once the doctor gives the consent for admission. As soon as the patient is admitted, the vital signs of the patient are monitored by the monitoring system. In case of difficulty

in reading vital signs or if the vital signs exceed normal limits, an alarm sounds. The doctor or the nurse or the clerk takes care of the situation and the alarm is silenced. The PM system also takes care of the patient discharge function by disabling alarms and updating the patient's history by adding all information until discharge. If the patient is a new patient, the details of the new patient including personal information and medical history are entered into the database by the hospital user.

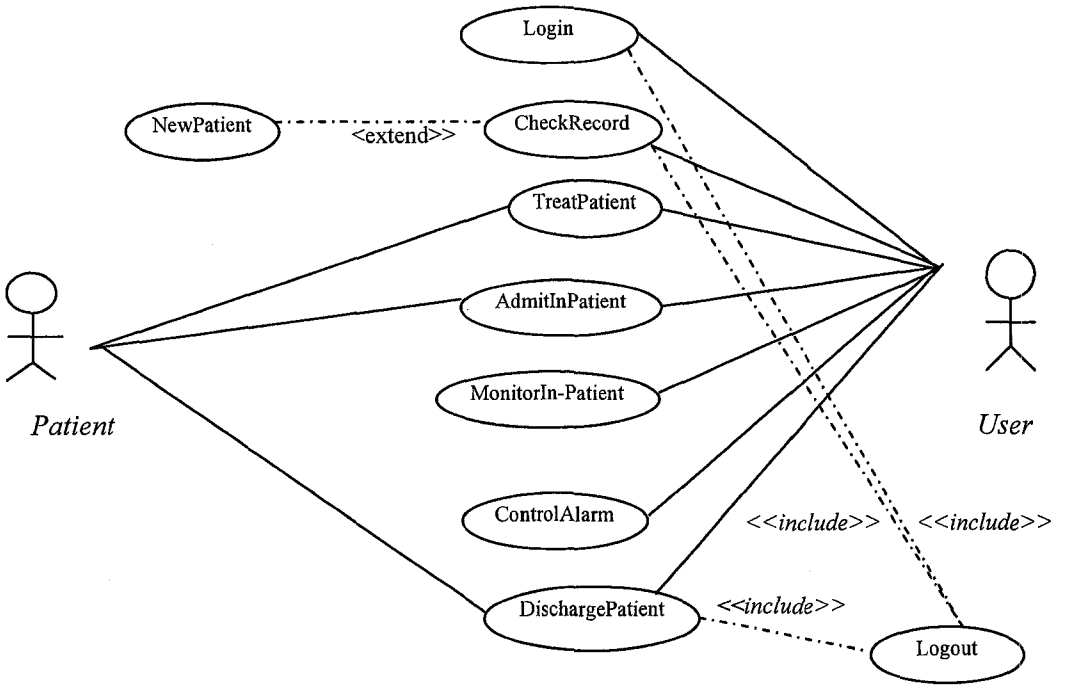


Figure 37: Patient Mangement System Use Case Diagram

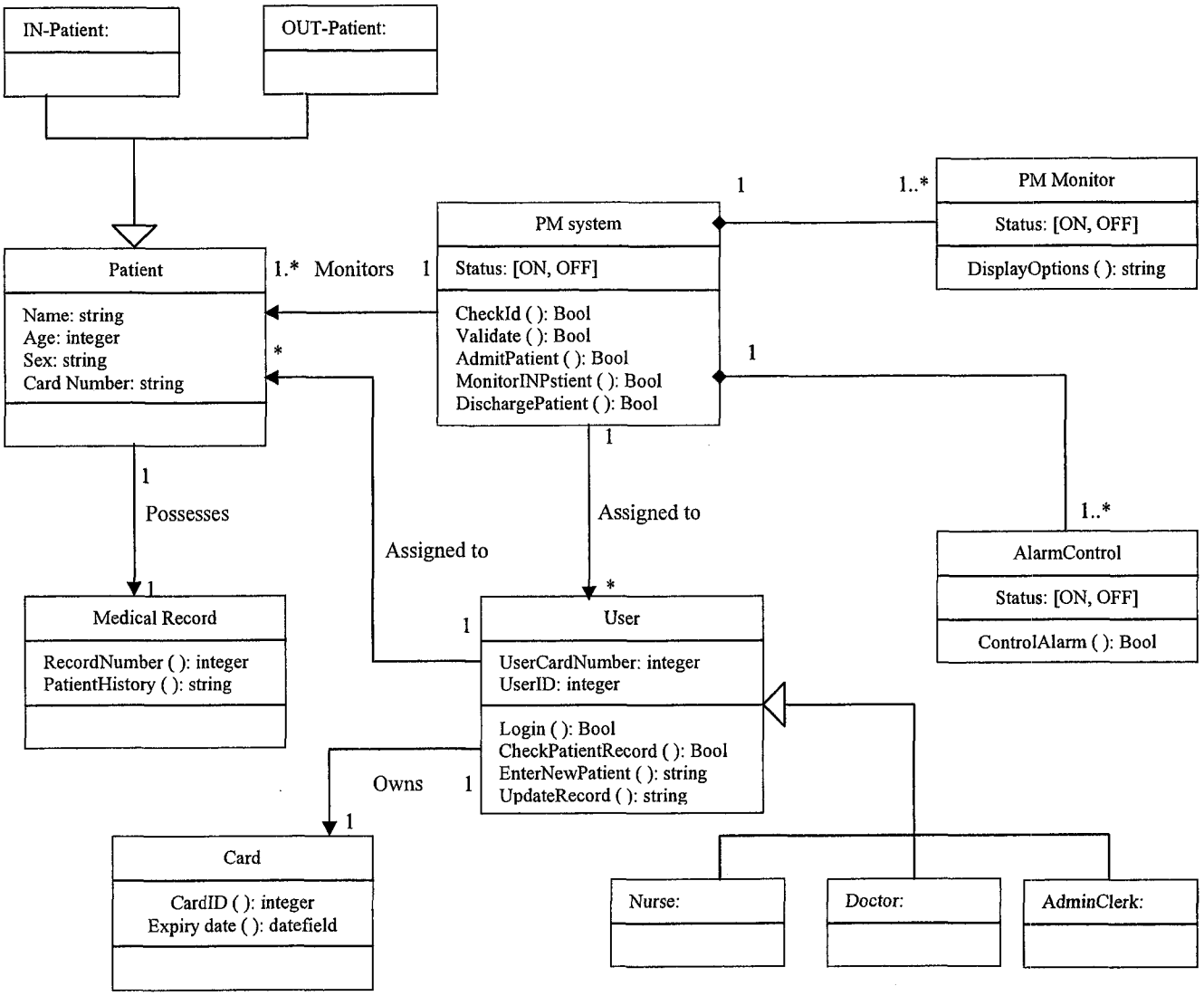


Figure 38: Patient Management System Domain Model

Use Cases:

The Use Cases for the patient management system as per the user requirements are:

- LOGIN
- CHECKRECORD
- TREATPATIENT
- ADMITINPATIENT
- MONITORINPATIENT
- CONTROLALARM
- LOGOUT
- DISCHARGEINPATIENT
- NEWPATIENT

Use Case: Login

Title : LOGIN

Primary Actor : USER

Goal : To grant access to PM system to perform activities like admitting patients

Precondition : PM system is ON, USER is NOT logged in

Postcondition : USER is logged in

Use Cases steps:

2. USER inserts card
3. PM system asks PIN
4. USER enters PIN
5. PM system checks PIN
6. If USER PIN is valid, PM system displays MENU options

Extensions:

1.a. USER Card is invalid

1.a.1. PM system starts system status alarm

1.a.2. After 20 seconds, PM system ejects card

2.a. After 60 seconds

2.a.1. PM system starts system status alarm

2.a.2. After 20 seconds, PM system ejects card

4.a. USER PIN is invalid and USER number of attempts is < 4

4.a.1. Go to step 2

4.b. USER PIN is invalid and USER number of attempts is = 4

4.b.1. PM system starts system status alarm

4.b.2. After 20 seconds, PM system ejects card

Use Case : CheckRecord

Title : CheckRecord

Primary Actor : USER-Admin.Clerk

Goal : Patient record is needed to be checked

Precondition : USER is logged in, PM system is ON, PM monitor displays MENU options

Postcondition : Patient Record is checked

Use Case steps:

- USER chooses CheckPatientRecord function from the MENU options
- PM system prompts Patient Record Number
- USER enters Patient Record Number

  Extension point ---> Create Record for new patients devoid of records: NEWPATIENT Use case
- If Patient Record Number is valid, patient Record is checked

Extensions:

2.a. After 60 seconds

2.a.1. PM system starts system status alarm

2.a.2. After 20 seconds, Go to step 2

4.a. Patient Record Number is NOT valid

4.a.1. PM system starts system status alarm

4.a.2. After 20 seconds, PM system displays MENU options

Use Case : TreatPatient

Title : TreatPatient

Primary Actor : USER-Doctor

Goal : Doctor decides the status of the patient to be in-patient or out-patient

Precondition : Patient Record is checked and doctor is assigned to patient

Postcondition : Patient status is in-patient OR out-patient

Use Case steps:

- Doctor checks patient

- If patient condition is serious, the doctor's decision is to admit patient and the patient status is in-patient

Extensions:

2.a. Patient condition NOT serious

2.a.1. Doctor gives medical prescriptions

2.a.2. Patient record is updated

Use Case : AdmitIn-Patient

Title : AdmitIn-Patient

Primary Actor : USER

Goal : To admit the patient in the hospital through the PM system

Precondition : Patient status is in-patient, USER is logged in, PM system displays MENU options, PM system is ON

Postcondition : Patient status is monitoring, doctor is assigned, Hospital location is assigned

Use Case steps:

- USER chooses Patient Admit function from the PM system monitor
- PM system prompts Hospital location and name of the doctor
- USER enters Hospital location and doctor's name
- PM system prompts vital signs
- USER enters vital signs
- USER connects cables
- PM system starts patient monitoring

Use Case: MonitorIn-Patient

Title : MonitorIn-Patient

Primary Actor : USER

Goal : To monitor the status of the admitted patient

Precondition : Patient status is monitoring, PM system is ON

Postcondition : Patient status is monitoring

Use Case steps:

4. PM system reads vital signs
5. PM system checks vital signs
6. PM system displays vital signs

Extensions:

1.a. Vital signs unreadable

1.a.1. PM system starts system status alarm

2.a. Vital signs in abnormal limits

2.a.1. PM system starts system status alarms

3.a. After 20 seconds,

3.a.1. Go to step 1.

Use Case : ControlAlarm

Title : ControlAlarm

Primary Actor : USER

Goal : To silence the alarm in case the alarm is triggered

Precondition : PM system is ON, PM system alarm status is ON, USER is logged in

Postcondition : Patient status is monitoring

Use Case steps:

- USER chooses AlarmSilence function from MENU options
- PM system suspends alarm
- After 60 seconds, PM system stops alarm
- Patient status is montoring

Extensions:

2.a. Before 60 seconds after suspending alarm

2.a.1. PM Monitor displays vital signs to be unreadable (OR) abnormal vital signs

Use Case: DischargePatient

Title : DischargePatient

Primary Actor : USER

Goal : To discharge the patient after treatment

Precondition : PM system is ON, USER is logged in, Patient status is monitoring, PM monitor displays MENU options

Postcondition : Patient Record is updated, Transaction details are dispatched

Use Case steps:

- USER chooses PatientDischarge function from the MENU
- PM system disables alarms
- USER disconnects cables

- Patient Record is updated and Transaction details are dispatched

Use Case: Logout

Title : Logout

Primary Actor : USER

Goal : An authorized USER wants to logout from the PM system

Precondition : USER is logged in, PM system displays MENU, PM system is ON

Postcondition : USER is logged out, PM system is ON

Use Cases steps:

- USER selects Logout option from the MENU
- PM system displays logout message

Use Case: NewPatient

Title : NewPatient

Primary Actor : USER

Goal : To enter the details of the new patient to the hospital data base

Precondition : USER is logged in, PM system displays MENU, PM system is ON

Postcondition : Patient Record is created, PM system is ON

Use Case steps:

1. USER selects CreateRecord option from MENU
2. PM system displays the input fields for the new patient
3. USER enters the patient details in the respective fields
4. USER clicks the SAVE option

6.1.2 Snapshot of the PMS domain model:

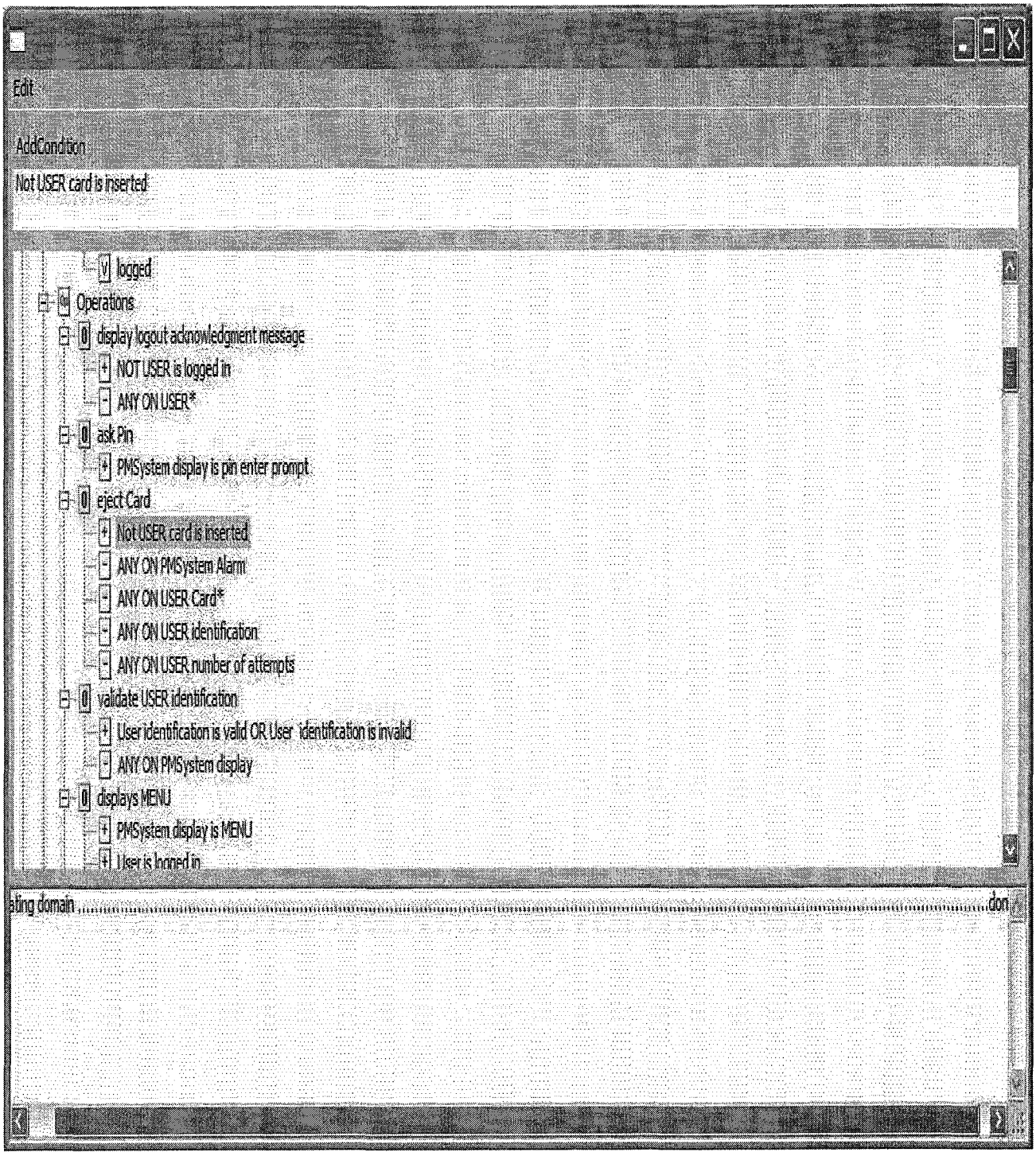


Figure 39: Domain Model Screen Shot of Patient Management System

6.1.3 Snapshot of the Use Cases in UCed

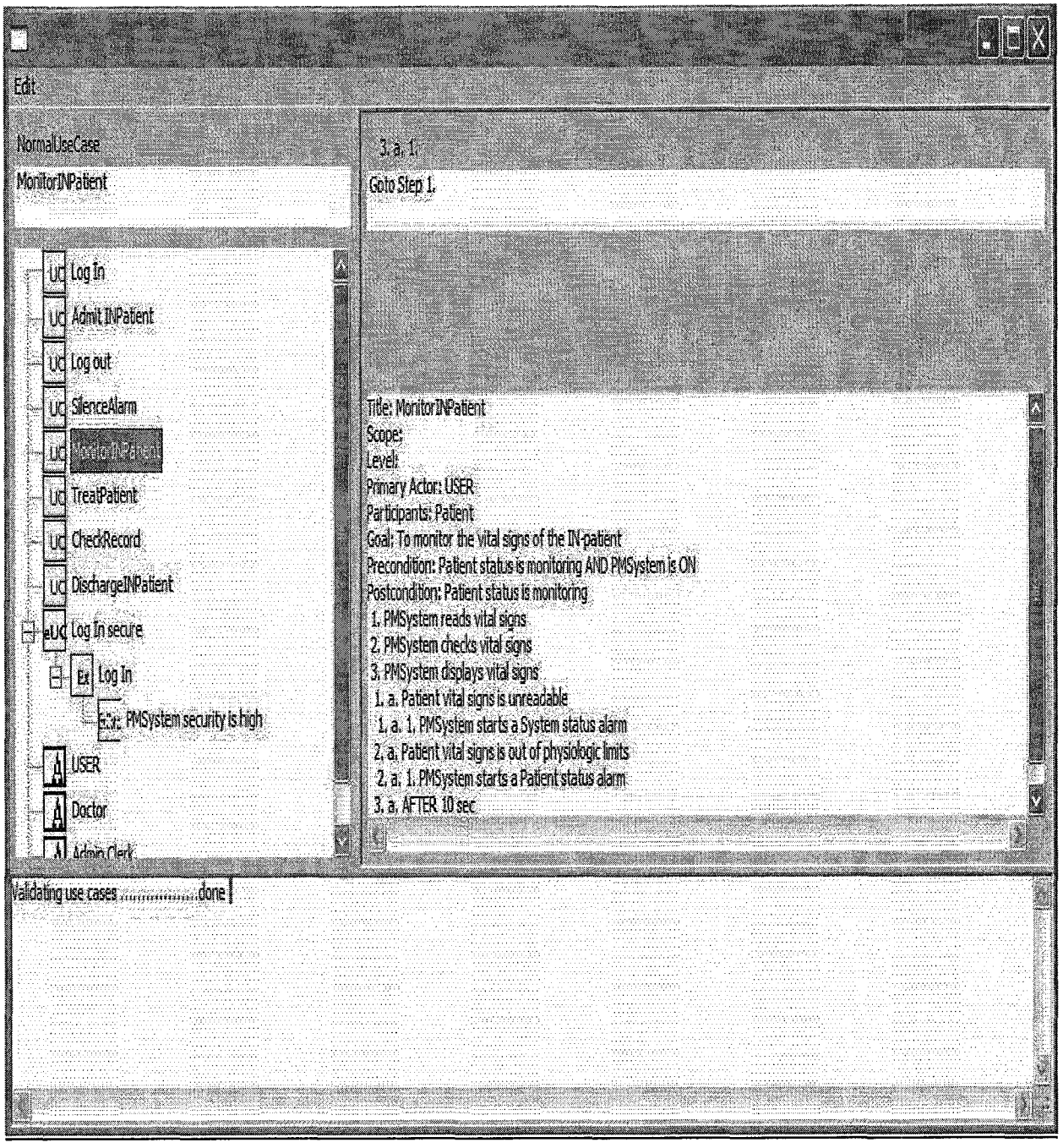


Figure 40: Screen Shot of Use Cases for Patient Management System

6.1.4 PSV verification by Prolog for Use Cases scenarios

Conventions:

All scenario sequence numbers are referred with respect to the above described Use Case. The input files '1' and '2' are inputs to the PSV module by the Prolog parser. All input files '1' referred in the future are the expected precondition and postcondition from the user's requirement document. All input files '2' referred in the future are the predicates corresponding to the precondition of the Use Case and predicates extracted sequentially from the domain model by the Prolog parser with respect to each Use Case step. The outputs are the verification sequence and reports obtained from the PSV module after verification.

Use Case Login (Happy Scenario):

The sequence is 1-2-3-4-5-6.

Input file 1:

Precondition : PMSystem(ON), User(NOT logged in)
Postcondition : User(logged in), PMSystem.Display(MENU)

Input file 2:

Pre : PMSystem(ON), User(NOT logged in)

Use Cases step 1: insertcard
User.Card(inserted)

Use Cases step 2: askPIN
PMSystem.Display(pin enter prompt)

Use Cases step 3: typePIN
User.Identification(entered),PMSystem.Display(ANY), User.NumberOfAttempts(ANY)

Use Cases step 4: validateIdentification
User.Identification(checking), PMSystem.Display(ANY)

Use Cases step 5:validIdentification
User.Identification(valid), PMSystem.Display(MENU)

Output file:

Verification Sequence:

VS line 1:
PMSystem(ON), User(NOT logged in)

VS line 2:
PMSystem(ON), User(NOT logged in), User.Card(inserted)

VS line 3:
PMSystem(ON), User(NOT logged in), User.Card(inserted), PMSystem.Display(pin enter prompt)

VS line 4:
PMSystem(ON), User(NOT logged in), User.Card(inserted), User.Identification(entered),

VS line 5:
PMSystem(ON), User(NOT logged in), User.Card(inserted), User.Identification(checking)

VS line 6:
PMSystem(ON), User(NOT logged in), UserCard(inserted), User.Identification(valid), PMSystem.Display(MENU)

Result of verification:

The actual obtained postcondition is :
PMSystem(ON), User(NOT logged in), User.Card(inserted), User.Identification(valid), PMSystem.Display(MENU)

The desired postcondition is:
User(logged in), PMSystem.Display(MENU)

The verification is not successful because the actual postcondition differs from the desired postcondition.

Report of suggestions:
The domain model needs to be reconsulted on the predicate 'User(logged in')
The reasons may be :

1. Domain model is conceptually contradicting the user requirement

   OR

2. Adequate withdrawn conditions are missing in some operations

Use Cases Login (Unhappy Scenario):

The sequence is 1-1a-1a1-1a2

Input file 1:

Precondition : PMSystem(ON), User(NOT logged in), User.CardStatus(irregular)

Postcondition : Timeout(20 sec), User.Card(NOT inserted), PMSystem(ON), User(NOT logged in)

Input file 2:

Pre : PMSystem(ON), User(NOT logged in)

Use Cases step 1a: insertcard

User.Card(inserted), User.CardStatus(irregular)

Use Cases step 1a1: startSystemStatusAlarm

PMSystem.Alarm(System status), PMSystem.Display(ANY)

Use Cases step 1a2: ejectCardAfter20sec

Timeout(20 sec), User.Card(NOT inserted), User.Identification(ANY) User.NumberOfAttempts(ANY)

Output file:

Verification Sequence:

VS line 1:

PMSystem(ON), User(NOT logged in)

VS line 2:

PMSystem(ON), User(NOT logged in), User.Card(inserted), User.CardStatus(irregular)

VSline 3:

PMSystem(ON), User(NOT logged in), User.Card(inserted), User.CardStatus(irregular), PMSystem.Alarm(System status)

VS line 4:

Timeout(20 sec),PMSystem(ON),User(NOT logged in), User.Card(NOT inserted), User.CardStatus(irregular), PMSystem.Alarm(System status)

Result of verification:

The actual obtained postcondition is :

Timeout (20 sec), PMSystem(ON), User(NOT logged in), User.Card(NOT inserted), User.CardStatus(irregular), PMSystem.Alarm(System status)

The desired postcondition is:
Timeout(20 sec), User.Card(NOT inserted), PMSystem(ON), User(NOT logged in)

The verification is successful because the actual postcondition satisfies the desired postcondition.

Invariant for the Use Cases:

- PMSystem(ON)
- User(NOT logged in)

Report of suggestions:

The domain model is inconsistent with the user requirements because of the additional predicates, UserCardStatus(irregular), PMSystemAlarm(System status)

The reasons may be :

1. Adequate withdrawn conditions are missing from input file operations lines 1a2.

Use Cases AdmitINPatient (Happy Scenario):

The sequence is 1-2-3-4-5-6-7.

Input file 1:

Precondition : PMSystem.Display(MENU), User(logged in), PMSystem(ON), Doctor.Decision(admitPatient)
Postcondition : Patient.Status(monitoring) , Doctor(assigned)

Input file 2:

Pre : PMSystem.Display(MENU), User(logged in), PMSystem(ON), Doctor.Decision(admitPatient)

Use Cases step 1: choosePatientAdmissionfunction
Patient.Status(admission initiated), PMSystem.Display(ANY)

Use Cases step 2: promptsDoctorName
Doctor.Name(asked), PMSystem.Display(doctor name prompt)

Use Cases step 3: enterDoctorName
Doctor.Name(entered), PMSystem.Display(doctor name)

Use Cases step 4: promptsVitalSigns
PMSystem.Display(vital signs prompt)

Use Cases step 5:entersVitalSigns)
Patient.Status(vital signs entered), PMSystem.Display(vital signs)

Use Cases step 6: connectsCables
Patient.Status(INPatient)

Use Cases step 7: startsPatientMonitoring
Patient.Status(monitoring)

Output file:

Verification Sequence:

VS line 1:
PMSystem.Display(MENU), User(logged in), PMSystem(ON), Doctor.Decision(admitPatient)

VS line 2:
User(logged in), PMSystem(ON), Doctor.Decision(admitPatient), Patient.Status(admission initiated)

VS line 3:
User(logged in), PMSystem(ON), Doctor.Decision(admitPatient), Patient.Status(admission initiated), Doctor.Name(asked), PMSystem.Display(doctor name prompt)

VS line 4:
User(logged in), PMSystem(ON), Doctor.Decision(admitPatient), Patient.Status(admission initiated), Doctor.Name(entered), PMSystem.Display(doctor name)

VS line 5:
User(logged in), PMSystem(ON), Doctor.Decision(admitPatient), Patient.Status(admission initiated), Doctor.Name(entered), PMSystem.Display(vital signs prompt)

VS line 6:
User(logged in), PMSystem(ON), Doctor.Decision(admitPatient), Doctor.Name(entered), Patient.Status(vital signs entered), PMSystem.Display(vital signs)

VS line 7:
Patient.Status(INPatient), User(logged in), PMSystem(ON), Doctor.Decision(admitPatient),

PMSystem.Display(vital signs)

VS line 8:
Patient.Status(monitoring), User(logged in), PMSystem(ON), Doctor.Decision(admitPatient), PMSystem.Display(vital signs)

Result of verification:

The actual obtained postcondition is :
Patient.Status(monitoring), User(logged in), PMSystem(ON), Doctor.Decision(admitPatient), PMSystem.Display(vital signs)

The desired postcondition is:
Patient.Status(monitoring) , Doctor(assigned)

The verification is not successful because the actual postcondition differs from the desired postcondition.

Report of suggestions:

The domain model needs to be reconsulted on the predicate 'Doctor(assigned)'
The reasons may be :

1. Adequate added conditions are missing in some operations

Use Cases MonitorINPatient (Happy Scenario):

The sequence is 1-2-3-3a-3a1

Input file 1:

Precondition : Patient.Status(monitoring), PMSystem(ON)
Postcondition : PMSystem.Display(vital signs), Patient.Status(monitoring),PMSystem(ON)

Input file 2:

Pre : Patient.Status(monitoring), PMSystem(ON)

Use Cases step 1: readVitalSigns
Patient.MonitoringStatus(vital signs reading), Patient.VitalSigns(ANY), PMSystem.Display(ANY)

Use Cases step 2: checkVitalSigns
Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(within physiologic limits)

Use Cases step 3: displayVitalSigns
PMSystem.Display(vital signs)

Use Cases step 3a1: after10sec
Timeout(10 sec), Patient.Status(monitoring)

Output file:

Verification Sequence:

VS line 1:
Patient.Status(monitoring), PMSystem(ON)

VS line 2:
Patient.MonitoringStatus(vital signs reading), Patient.Status(monitoring), PMSystem(ON)

VS line 3:
Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(within physiologic limits), Patient.Status(monitoring), PMSystem(ON)

VS line 4:
PMSystem.Display(vital signs), Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(within physiologic limits), Patient.Status(monitoring), PMSystem(ON)

VS line 5:
Timeout(10 sec), Patient.Status(monitoring), PMSystem.Display(vital signs), Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(within physiologic limits), PMSystem(ON)

Result of verification:

The actual obtained postcondition is :
Timeout(10 sec), Patient.Status(monitoring), PMSystem.Display(vital signs), Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(within physiologic limits), PMSystem(ON)

The desired postcondition is:
PMSystem.Display(vital signs), Patient.Status(monitoring), PMSystem(ON)

The verification is successful because the actual postcondition satisfies the desired postcondition.

Invariant for the Use Cases:

1. PMSystem(ON)
2. Patient.Status(monitoring)

Report of suggestions:

The domain model is inconsistent with the user requirements because of the additional predicates, PatientMonitoringStatus(vital signs checking), PatientVitalSigns(within physiologic limits)

The reasons could likely be due to adequate withdrawn conditions missing in input file operations lines 3.

Use Cases MonitorINPatient (Unhappy Scenario):

The sequence is 1-2-3-3a-3a1

Input file 1:

Precondition : Patient.Status(monitoring), PMSystem(ON), Patient.VitalSigns(out of physiologic limits)
Postcondition: PMSystem.Alarm(Patient status), PMSystem.Alarm(ON), PMSystem(ON), Patient.status(monitoring)

Input file 2:

Pre : Patient.Status(monitoring), PMSystem(ON)

Use Cases step 1: readVitalSigns
Patient.MonitoringStatus(vital signs reading), Patient.VitalSigns(ANY), PMSystem.Display(ANY)

Use Cases step 2a: checkVitalSigns
Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns(out of physiologic limits)

Use Cases step 2a1: startPatientStatusAlarm
PMSystem.Alarm(Patient status), PMSystem.Alarm(ON), Patient.VitalSigns(ANY), Patient.MonitoringStatus(ANY)

Output file:

Verification Sequence:

VS line 1:
Patient.Status(monitoring), PMSystem(ON)

VS line 2:

Patient.MonitoringStatus(vital signs reading), Patient.Status(monitoring), PMSystem(ON)

VS line 3:

Patient.MonitoringStatus(vital signs checking), Patient.VitalSigns (out of physiologic limits), Patient.Status(monitoring), PMSystem(ON)

VS line 4:

PMSystem.Alarm(Patient status), PMSystem.Alarm(ON), PMSystem(ON), Patient.Status(monitoring)

Result of verification:

The actual obtained postcondition is :

PMSystem.Alarm(Patient status), PMSystem.Alarm(ON), PMSystem(ON), Patient.Status(monitoring)

The desired postcondition is:

PMSystem.Alarm(Patient status), PMSystem.Alarm(ON), PMSystem(ON), Patient.Status(monitoring)

The verification is successful and the domain model is consistent with requirements because the actual postcondition is exactly the desired postcondition with invariants.

Invariant for the Use Cases:

1. PMSystem(ON)
2. PatientStatus(monitoring)

6.1.4 Result of PSV verification after domain model improvement for Use Case Login (Unhappy Scenario)

Use Cases Login (Unhappy Scenario):

The improvement in the domain model is:

A withdrawnCondition, *User.CardStatus (ANY)* and *PMSystem.Alarm(ANY)* were added to the domain operation, *ejectCardAfter20sec* which corresponds to the Use Cases step 1a2.

The sequence is 1-1a-1a1-1a2

Input file 1:

Precondition : PMSystem(ON), User(NOT logged in), User.CardStatus(irregular)
Postcondition : Timeout (20 sec), User.Card(NOT inserted), User(NOT logged in), PMSystem(ON)

Input file 2:

Pre : PMSystem(ON), User(NOT logged in)

Use Cases step 1a: insertcard
UserCard(inserted), User.CardStatus(irregular)

Use Cases step 1a1: startSystemStatusAlarm
PMSystem.Alarm(System status), PMSystem.Display(ANY), User.CardStatus(irregular)

Use Cases step 1a2: ejectCardAfter20sec
Timeout(20 sec), User.Card(NOT inserted), PMSystem.Alarm(ANY), UserCard.Status(ANY) User.Identification(ANY) , User.NumberOfAttempts(ANY)

Output file:

Verification Sequence:

VS line 1:
PMSystem(ON), User(NOT logged in)

VS line 2:
PMSystem(ON), User(NOT logged in), User.Card(inserted), User.CardStatus(irregular)

VS line 3:

PMSystem.Alarm(System status), User(NOT logged in) PMSystem(ON), User.Card(inserted), User.CardStatus(irregular)

VS line 4:

Timeout(20 sec), PMSystem(ON), User(NOT logged in), User.Card(NOT inserted)

Result of verification:

The actual obtained postcondition is :

Timeout (20 sec), PMSystem(ON), User(NOT logged in), UserCard(NOT inserted)

The desired postcondition is:

Timeout(20 sec), PMSystem(ON), UserCard(NOT inserted), User(NOT logged in)

The verification is successful and the domain model is consistent because the actual postcondition is same as the desired postcondition inspite of the invariants.

Invariant for the Use Cases:

1.PMSystem(ON)

2. User(NOT logged in)

Report of suggestions:

The domain model is consistent with the user requirements

6.1.5 Output snapshot from PSV module

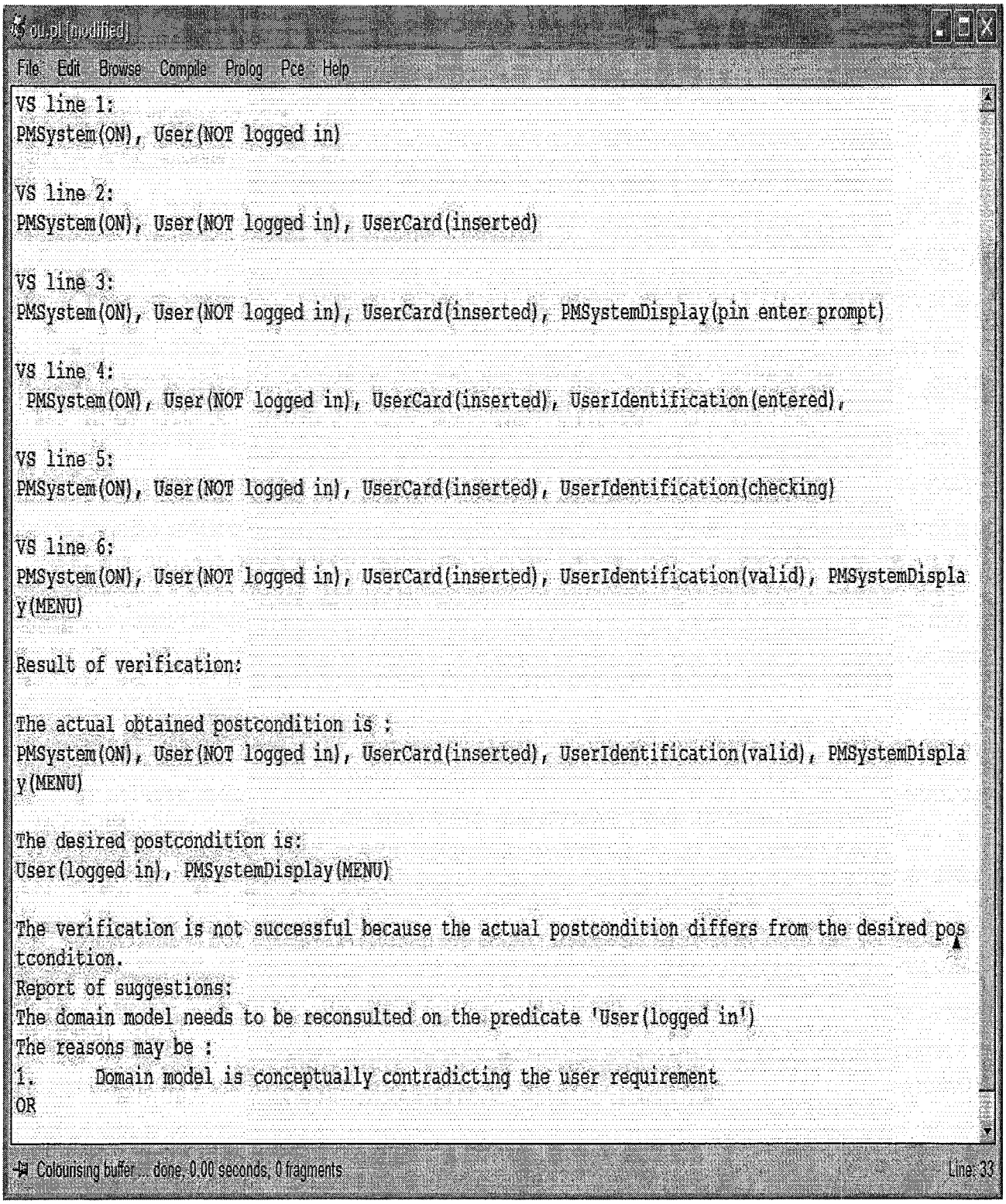


Figure 41: Screen Shot of PSV Output

6.1.6 Complexity of the PSV process for the PM System

Scalability:

The PSV process uses three files for its execution. There are two input files and one output file. The first input file contains the expected pre/postcondition of the use case and the second input file contains the logical predicates corresponding to the operations concerning each use case step sequentially. The output file displays the verification sequence resulting form the PSV process and the result of verification with report on suggestions for domain model improvement. Since we use abstraction using predicates, the evaluation of PSV process reveals that it scales in accordance with the number of predicates used for representing a whole use case.

For a single use case scenario, S:

Input file 1:

Precondition: a set of predicates in conjunction,

Pre = {P1/\ P2/\...../\Pk}

where '*k*' represents the number of predicates for the precondition

Postcondition: a set of predicates in conjunction,

Post = {P1/\ P2/\...../\Pt}

where '*t*' represents the number of predicates for the postcondition

Therefore, input file 1 scales to a total number of **'k+*t*'** predicates.

Input file 2:

Let the total number of steps in a use case scenario be '*n*'.

Each step in the use case scenario corresponds to a domain operation represented by a conjunction of predicates true at the execution of that operation. Let *F(s)* be a function which conveys the total number of predicates representing the use case step '*s*'. A use case is assumed to be comprised of a finite number of steps indicating that the use case will surely be terminated.

Therefore, input file 2 scales to ***F(s1)+F(s2)+...+F(sn)***

So for the single use case scenario, S, the input files scale to:

[k] + [F(s1) + F(s2) + ..+F(sn)] number of predicates.

In the PSV verification process, the '*passing*' mechanism causes predicates corresponding to each use case step to be passed to the next use case step. So a Verification Sequence line (VS line) of a use case step, s, in the output file, in the worst case, will consists of all the logical predicates corresponding to the use case step, s and the use case steps which occurred before s. The output file for scenario, S, in the worst case scales to:

{[k] + [k + F(s1)] + [k + F(s2) + F(s3)] + ...+ [k + F(s1) + F(s2) +...+ F(sn)]} number of predicates.

Time complexity:

For a single use case scenario, S, let 'n' denote the number of use case steps in S.

In the PSV process, each use case step, s is scanned once to retrieve the logical predicates corresponding to the domain operation represented by s.

Let '**Ґ(s)**' be the total time taken for retriving the domain operation postconditions corresponding to the step, s and translating the operation postconditions to logical predicates.

There are finite number of postconditions listed for a domain operation, P.

Let t(c1) represents the total time taken to retrive the first postcondition,c1.

Hence, for a domain operation containing 'k' number of postconditions, the time taken to retrieve all the 'k' postconditions will be T(c1) + T(c2)+...+T(ck) and the time required for predicate translation of these postconditions corresponingly will be T(p1) + T(p2) +...+T(pk)

Therefore, $Ґ(s) = T(c1p1) + T(c2p2) +...+ T(ckpk)$

Hence the total time taken for the execution of the PSV process for one single use case scenario, S:

$Ґ(S) = Ґ(s1) + Ґ(s2) + ...+ Ґ(sn)$

Accuracy of Suggestions:

Suggestions for improving the domain model are provided in domain model inconsistency and unsuccessful cases when the expected postcondition of the use case scenario is not equal to the obtained postcondition from the PSV process. The reasons for domain model inconsistency exist because of unnecessary withdrawnconditions specified for operations by the developer or missing withdrawnconditions for specified operations in the domain model. Generally, for a use case scenario in the PSV process, each use case step corresponds to a specific operation in the domain model. After the execution of this step, the postconditions (AddedConditions and WithdrawnConditions) for the operation are satisfied which means that the logical predicates

corresponding to the AddedConditions become true and the logical predicates corresponding to the WithdrawnConditions become neglected. When the result of this use case step is passed to the next immediate use case step, the AddedConditions for this step will be applied to the result. If there are no WithdrawnConditions for this step, then there may exist some predicates corresponding to conditions unnecessary or contradictory for the current use case step which in turn lead to inconsistency in the domain model.

For example, for the scenario for Login use case, 1-1a-1a1-1a2,

The desired postcondition is:

Timeout(20 sec), User.Card(NOT inserted), User(NOT logged in), PMSystem(ON)

The actual obtained postcondition is:

Timeout (20sec),PMSystem(ON), User(NOT logged in), User.Card(NOT inserted), User.CardStatus(irregular), PMSystem.Alarm(System status)

The report of suggestion is:

The domain model is inconsistent with the user requirements because of the additional predicates, UserCardStatus(irregular), PMSystemAlarm(System status)

The reasons could likely be be due to adequate withdrawn conditions were missing from input file operations lines 1a2.

In this case, the additional predicates, User.CardStatus(irregular) and PMSystemAlarm(System status) caused the inconsistency. Here, after the PM system detects that the User card is invalid, the final output should be the ejection of user card after 20 seconds. But after the execution of the scenario, although the card is ejected after 20 seconds, the status of the card is still set as irregular and the PMsystem alarm is still System status (ON) which is untrue after card ejection. Therefore necessary Withdrawnconditions should be inserted at the particular domain operation at which the conditions stop to be true. In this case, it is the domain operation corresponding to the last use case step, *'After 20 seconds, PM system ejects card'*. Once the Withdrawnconditions 'UserCardStatus(ANY) and PMSystemAlarm(ANY) are included at the above use case step domain operation, the output will be equal to the desired postcondition and the domain operations will become consistent.

6.2 Semi-Automated Validation

6.2.1 Semi-Automated Validation prior to performing PSV verification

The use cases 'Login' and 'MonitorINPatient' are composed to generate the state chart from UCed

Input file 1:

```
UCL-pre: PMSystem (ON) and USER (NOT logged in)
UCL-post: USER (logged in)

UCL-pre: USER. Card (inserted) and USER. card status (irregular)
UCL-post: Timeout(20sec) and USER. Card (NOT inserted)

UCL-pre: PMSystem. Display (pin enter prompt) and Timeout(60sec)
UCL-post: Timeout (20sec) and USER. Card (NOT inserted)

UCL-pre: USER. Attempts (checking) and USER. number of attempts (< 4)
UCL-post: PMSystem. Display (pin enter prompt) and Timeout (60sec) and USER. Card (NOT inserted)

UCL-pre: USER. Attempts (checking) and USER. number of attempts (>= 4)
UCL-post: Timeout (20sec) and USER. Card (NOT inserted)

UCL-pre: Patient. status (monitoring) and PMSystem (ON)
UCL-post: Patient. status (monitoring)

UCL-pre: Patient. Vital status (being checked) and Patient. Vital Signs (unreadable)
UCL-post: PMSystem. Alarm (System Status)

UCL-pre: Patient. Status (monitoring) and Patient. Vital Signs (out of physiologic limits)
UCL-post: PMSystem. Alarm (System Status)

UCL-pre: PMSystem. Display (vital signs) and Timeout (20sec)
UCL-post: Patient. Monitoring status (vital signs reading)
```

Input file 2:

The state chart generated from UCed for PM system

```
SState:1 [USER - NOT logged in, PMSystem - ON]
 SState:2 [USER Card status – checking, PMSystem – ON]
 SState:3 [USER Card - inserted, USER - NOT logged in, PMSystem Alarm - System Status, USER Card status - irregular, PMSystem - ON]
 SState:4 [USER Card - inserted, PMSystem Alarm - System Status, USER - NOT logged in, USER Card status - irregular, Timer4:20.0 second, PMSystem - ON]
 SState:5 [USER - NOT logged in, USER Card - NOT inserted, PMSystem - ON]
 SState:6 [USER Card - inserted, USER - NOT logged in, PMSystem Display - pin enter prompt, PMSystem - ON]
 SState:7 [USER Card - inserted, USER - NOT logged in, PMSystem Display - pin enter prompt, Timer7:60.0 second, PMSystem - ON]
 SState:8 [USER Card - inserted, USER - NOT logged in, PMSystem Alarm - System Status, PMSystem - ON]
 SState:9 [USER identification - entered, USER Card - inserted, USER - NOT logged in, PMSystem - ON]
 SState:10 [USER Validation status – checking, PMSystem – ON]
 SState:11 [USER Card - inserted, PMSystem Alarm - System Status, USER - NOT logged in, Timer9:20.0 second, PMSystem - ON]
 SState:12 [USER Card - inserted, USER - NOT logged in, PMSystem - ON, USER identification - valid]
```

SState:13 [USER Card - inserted, PMSystem Display - MENU, PMSystem - ON]

SState:14 [USER attempts – checking, PMSystem – ON]

SState:15 [USER Card - inserted, USER - NOT logged in, PMSystem Alarm - System Status, USER number of attempts >= 4, PMSystem - ON, USER identification - invalid]

SState:16 [Timer16:20.0 second, USER Card - inserted, PMSystem Alarm - System Status, USER - NOT logged in, USER number of attempts >= 4, PMSystem - ON, USER identification - invalid]

SState:17 [Patient Vital signs –checking, Patient status – monitoring, PMSystem – ON]

SState:18 [Patient Vital Signs - out of physiologic limits, Patient status - monitoring, PMSystem - ON, Patient Monitoring status - vital signs checking]

SState:19 [Patient Vital Signs - out of physiologic limits, Patient status - monitoring, PMSystem Alarm - Patient Status, Patient Monitoring status - vital signs checking, PMSystem - ON]

SState:20 [Patient Vital Signs - within physiologic limits, Patient status - monitoring, PMSystem - ON, Patient Monitoring status - vital signs checking]

SState:21 [Patient Vital Signs - within physiologic limits, Patient status - monitoring, Patient Monitoring status - vital signs checking, PMSystem Display - vital signs, PMSystem - ON]

SState:22 [Patient Vital Signs - within physiologic limits, Timer28:10.0 second, Patient status - monitoring, PMSystem - ON, PMSystem Display - vital signs, Patient Monitoring status - vital signs checking]

SState:23 [Patient status - monitoring, PMSystem - ON]

SState:24 [Patient Vital status –being checked, PMSystem – ON]

SState:25 [USER Card - inserted, USER - NOT logged in, PMSystem - ON, USER Card status - irregular]

SState:26 [USER Card - inserted, USER - NOT logged in, PMSystem - ON]

SState:27 [USER Card - inserted, USER - NOT logged in, PMSystem - ON, USER identification - invalid, USER number of attempts >= 4]

SState:28 [USER Card - inserted, USER - NOT logged in, PMSystem - ON, USER identification - invalid, USER number of attempts < 4]

SState:29 [USER Card - inserted, USER - NOT logged in, PMSystem - ON, USER identification - invalid]

SState:30 [Patient Monitoring status - vital signs reading, Patient status - monitoring, PMSystem - ON]

SState:31 [Patient Monitoring status - vital signs reading, Patient status - monitoring, PMSystem - ON, Patient Vital Signs - unreadable]

SState:32 [PMSystem Alarm - System Status, Patient Monitoring status - vital signs reading, Patient status - monitoring, Patient Vital Signs - unreadable, PMSystem - ON]

SState:33 Patient status – monitoring, PMSystem - ON

**** SCTRANSITIONS ***

1---insert card/-->2

2---[USER Card status - irregular]/start System status alarm-->3

2---[USER Card status - NOT irregular]/ask Pin-->6

3---TIMEOUT(Timer4:20.0 second)/eject Card-->5

6---TIMEOUT(Timer7:60.0 second)/start System status alarm-->8

6---type PIN/validate USER identification-->10

8---TIMEOUT(Timer9:20.0 second)/eject Card-->5

10---[USER identification - valid]/displays MENU-->13

10---[USER identification - invalid]/-->14

14---[USER number of attempts < 4]/-->2

14---[USER number of attempts >= 4]/start System status alarm-->15

15---TIMEOUT(Timer16:20.0 second)/eject Card-->5

17---[Patient Vital Signs - out of physiologic limits]/start Patient status alarm-->19

17---[Patient Vital Signs - within physiologic limits]/display vital signs-->21

21---TIMEOUT(Timer28:10.0 second)/read vital signs-->24

24---[Patient Vital Signs - NOT unreadable]/check vital signs-->17

24---[Patient Vital Signs - unreadable]/start System status alarm-->32

Output file 1 from semi-automated validation:

1. USER(NOT logged in) and PMSystem(ON)
2. USER. Card (inserted), PMSystem. Display (MENU), PMSystem (ON)
3. USER (NOT logged in) and PMSystem (ON) and IF USER.Card(inserted)
4. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (irregular)
5. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)
6. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
7. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (NOT irregular)
8. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF Timeout (60sec)
9. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
10. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
11. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)
12. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
13. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF Timeout (60sec)
14. [USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
15. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
16. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF USER. pin(entered)
17. USER. Validation status (checking) and PMSystem (ON) and IF USER. identification (valid)
18. USER. Card (inserted), PMSystem. Display (MENU), PMSystem (ON)
19. [USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
20. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
21. USER. Validation status (checking) and PMSystem (ON) and IF USER. identification (invalid)
22. USER. Attempts (checking) and PMSystem (ON) and IF USER. numberofattempts (<4)
23. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (NOT irregular)
24. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF Timeout (60sec)
25. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
26. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
27. USER. Attempts (checking) and PMSystem (ON) and IF USER. numberofattempts (<4)

28. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (irregular)
29. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)
30. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
31. USER. Attempts (checking) and PMSystem (ON) and IF USER. numberofattempts (>=4)
32. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)
33. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
34. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)
35. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
36. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON)
37. Patient. Status (monitoring) and PMSystem (ON)
38. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)
39. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
40. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs(within physiologic limits)
41. Patient. Vital Signs (within physiologic limits) and Patient. status (monitoring) and Patient. Monitoring status (vital signs checking) and PMSystem. Display (vital signs) and PMSystem (ON) and IF Timeout (10sec)
42. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
43. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)
44. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
45. Patient. Vital Signs (within physiologic limits) and Patient. status (monitoring) and Patient. Monitoring status (vital signs checking) and PMSystem. Display (vital signs) and PMSystem (ON) and IF Timeout (10sec)
46. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
47. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)
48. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
49. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
50. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs(within physiologic limits)
51. Patient. Vital Signs (within physiologic limits) and Patient. status (monitoring) and Patient. Monitoring status (vital signs checking) and PMSystem. Display (vital signs) and PMSystem (ON) and IF Timeout (10sec)
52. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
53. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)
54. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
55. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (unreadable)
56. PMSystem. Alarm (System Status) and Patient. Monitoring status (vital signs reading) and Patient. Status (monitoring) and Patient. Vital Signs (unreadable) and PMSystem (ON)

Result of verification:

Scenario 1:

Precondition: PMSystem (ON) and USER (NOT logged in)

The desired postcondition is : USER (logged in)

The obtained postcondition is:

2. USER. Card (inserted), PMSystem. Display (MENU), PMSystem (ON)

The verification is unsuccessful because the Use Cases postcondition does not conform to the condition of SCL line 2.

Scenario 2:

Precondition: USER. Card (inserted) and USER. card status (irregular)

The desired postcondition is : Timeout(20sec) and USER. Card (NOT inserted)

The obtained postcondition is:

5. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)

6. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 5 and 6.

Scenario 3:

Precondition: PMSystem. Display (pin enter prompt) and Timeout(60sec)

The desired postcondition is : Timeout (20sec) and USER. Card (NOT inserted)

The obtained postcondition is:

9. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)

10. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 9 and10.

Scenario 4:

Precondition: USER. Attempts (checking) and USER. number of attempts (< 4)

The desired postcondition is : PMSystem. Display (pin enter prompt) and Timeout (60sec) and USER. Card (NOT inserted)

The obtained postcondition is:

24. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (irregular)

25. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) . and PMSystem (ON) and IF Timeout (20sec)

26. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 24,25 and 26.

Scenario 5:

Precondition: Attempts (checking) and USER. number of attempts (>= 4)

The desired postcondition is : Timeout (20sec) and USER. Card (NOT inserted)

The obtained postcondition is:
32. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)
33. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
34. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)
35. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 32, 33,34 and 35.

Scenario 6:

Precondition: Patient. status (monitoring) and PMSystem (ON)

The desired postcondition is : Patient. status (monitoring)

The obtained postcondition is:
37. Patient. status (monitoring) and PMSystem (ON)
The verification is successful because the Use Cases postcondition conforms to the condition of SCL line 37.

Scenario 7:

Precondition: Patient. Vital status (being checked) and Patient. Vital Signs (unreadable)

The desired postcondition is : PMSystem. Alarm (System Status)

The obtained postcondition is:
56. PMSystem. Alarm (System Status) and Patient. Monitoring status (vital signs reading) and Patient. Status (monitoring) and Patient. Vital Signs (unreadable) and PMSystem (ON)
The verification is successful because the Use Cases postcondition conforms to the condition of SCL line 56.

Scenario 8:

Precondition: Patient. Status (monitoring) and Patient. Vital Signs (out of physiologic limits)

The desired postcondition is : PMSystem. Alarm (System Status)

The obtained postcondition is:
39. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
The verification is successful because the Use Cases postcondition conforms to the condition of SCL line 39.

Scenario 9:

Precondition: PMSystem. Display (vital signs) and Timeout (20sec)

The desired postcondition is : Patient. Monitoring status (vital signs reading)

The obtained postcondition is:
"cannot be determined"
The verification is unsuccessful because the Use Case precondition is missing from the state chart.

6.2.2 Output Analysis prior to PSV verification:

It can be seen from the above Semi-Automated Verification (SAV) result that two violations are detected before the PSV verification process.

1. Scenario 1- The predicate, *USER (logged in)* is missing from the obtained SAV postcondition. This is because since the PSV verification is not performed, the domain model is not free from requirement errors and the statechart was generated from the domain model with requirement errors.

2. Scenario 9- The precondition to be checked is absent in the state chart, in other words, a state representing the precondition of scenario 9 is missing in the state chart. This is a serious case of functional requirement violation.

6.2.3 Semi-Automated Validation after performing PSV verification

Output file:

1. USER(NOT logged in) and PMSystem(ON)
2. USER. Card (inserted), USER(logged in), PMSystem. Display (MENU), PMSystem (ON)
3. USER (NOT logged in) and PMSystem (ON) and IF USER.Card(inserted)
4. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (irregular)
5. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)
6. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
7. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (NOT irregular)
8. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF Timeout (60sec)
9. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
10. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
11. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)

12. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
13. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF Timeout (60sec)
14. [USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
15. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
16. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF USER. pin(entered)
17. USER. Validation status (checking) and PMSystem (ON) and IF USER. identification (valid)
18. USER. Card (inserted), PMSystem. Display (MENU), PMSystem (ON)
19. [USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
20. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
21. USER. Validation status (checking) and PMSystem (ON) and IF USER. identification (invalid)
22. USER. Attempts (checking) and PMSystem (ON) and IF USER. numberofattempts (<4)
23. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (NOT irregular)
24. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Display (pin enter prompt) and PMSystem (ON) and IF Timeout (60sec)
25. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)
26. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
27. USER. Attempts (checking) and PMSystem (ON) and IF USER. numberofattempts (<4)
28. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (irregular)
29. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)
30. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
31. USER. Attempts (checking) and PMSystem (ON) and IF USER. numberofattempts (>=4)
32. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)
33. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
34. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)
35. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)
36. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON)
37. Patient. Status (monitoring) and PMSystem (ON)
38. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)
39. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
40. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs(within physiologic limits)
41. Patient. Vital Signs (within physiologic limits) and Patient. status (monitoring) and Patient. Monitoring status (vital signs checking) and PMSystem. Display (vital signs) and PMSystem (ON) and IF Timeout (10sec)
42. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
43. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)

44. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
45. Patient. Vital Signs (within physiologic limits) and Patient. status (monitoring) and Patient. Monitoring status (vital signs checking) and PMSystem. Display (vital signs) and PMSystem (ON) and IF Timeout (10sec)
46. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
47. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)
48. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
49. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
50. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs(within physiologic limits)
51. Patient. Vital Signs (within physiologic limits) and Patient. status (monitoring) and Patient. Monitoring status (vital signs checking) and PMSystem. Display (vital signs) and PMSystem (ON) and IF Timeout (10sec)
52. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (NOT unreadable)
53. Patient. Vital signs (checking) and Patient. status (monitoring) and PMSystem (ON) and IF Patient. Vital Signs (out of physiologic limits)
54. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)
55. Patient. Vital status (being checked) and PMSystem (ON) and IF Patient. Vital Signs (unreadable)
56. PMSystem. Alarm (System Status) and Patient. Monitoring status (vital signs reading) and Patient. Status (monitoring) and Patient. Vital Signs (unreadable) and PMSystem (ON)

Result of verification:

Scenario 1:

Precondition: PMSystem (ON) and USER (NOT logged in)

The desired postcondition is : USER (logged in)

The obtained postcondition is:

2. USER. Card (inserted), USER(logged in), PMSystem. Display (MENU), PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL line 2.

Scenario 2:

Precondition: USER. Card (inserted) and USER. card status (irregular)

The desired postcondition is : Timeout(20sec) and USER. Card (NOT inserted)

The obtained postcondition is:

5. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) and PMSystem (ON) and IF Timeout (20sec)

6. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 5 and 6.

Scenario 3:

Precondition: PMSystem. Display (pin enter prompt) and Timeout(60sec)

The desired postcondition is : Timeout (20sec) and USER. Card (NOT inserted)

The obtained postcondition is:

9. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and PMSystem (ON) and IF Timeout(20sec)

10. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 9 and10.

Scenario 4:

Precondition: USER. Attempts (checking) and USER. number of attempts (< 4)

The desired postcondition is : PMSystem. Display (pin enter prompt) and Timeout (60sec) and USER. Card (NOT inserted)

The obtained postcondition is:

24. USER. Card status (checking) and PMSystem (ON) and IF USER. Card status (irregular)

25. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. Card status (irregular) . and PMSystem (ON) and IF Timeout (20sec)

26. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 24,25 and 26.

Scenario 5:

Precondition: Attempts (checking) and USER. number of attempts (>= 4)

The desired postcondition is : Timeout (20sec) and USER. Card (NOT inserted)

The obtained postcondition is:

32. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)

33. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

34. USER. Card (inserted) and USER (NOT logged in) and PMSystem. Alarm (System Status) and USER. number of attempts (>= 4) and PMSystem (ON) and USER. identification (invalid) and IF Timeout (20sec)

35. USER (NOT logged in) and USER. Card (NOT inserted) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL lines 32, 33, 34 and 35.

Scenario 6:

Precondition: Patient. status (monitoring) and PMSystem (ON)

The desired postcondition is : Patient. status (monitoring)

The obtained postcondition is:

37. Patient. status (monitoring) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL line 37.

Scenario 7:

Precondition: Patient. Vital status (being checked) and Patient. Vital Signs (unreadable)

The desired postcondition is : PMSystem. Alarm (System Status)

The obtained postcondition is:

56. PMSystem. Alarm (System Status) and Patient. Monitoring status (vital signs reading) and Patient. Status (monitoring) and Patient. Vital Signs (unreadable) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL line 56.

Scenario 8:

Precondition: Patient. Status (monitoring) and Patient. Vital Signs (out of physiologic limits)

The desired postcondition is : PMSystem. Alarm (System Status)

The obtained postcondition is:

39. Patient. Vital Signs (out of physiologic limits) and Patient. status (monitoring) and PMSystem. Alarm (Patient Status) and Patient. Monitoring status (vital signs checking) and PMSystem (ON)

The verification is successful because the Use Cases postcondition conforms to the condition of SCL line 39.

Scenario 9:

Precondition: PMSystem. Display (vital signs) and Timeout (20sec)

The desired postcondition is : Patient. Monitoring status (vital signs reading)

The obtained postcondition is:

"cannot be determined"

The verification is unsuccessful because the Use Case precondition is missing from the state chart.

6.2.4 Output analysis after PSV verification

It can be seen from the above SAV output after PSV process that the Scenario 1 violation is eliminated , but the Scenario 9 violation still exists. Scenario 9 violation is caused because of the missing state corresponding to the precondition of the use case scenario. This issue arises beacuse inadequacies in the technical generation of the state chart during integration of various use cases.

6.2.5 Complexity of Semi-Automated Validation

Scalability:

The above state chart generated for the PM System consists of 33 states. There are two input files to Semi-Automated Validation. The first input file contains the desired pre/postcondition sets for the use cases and the second input file contains the original state chart.. The output filedisplays the SCL verification sequence for the generated state chart and describes the result of verification.

For input file 1:

Let there be a total of 'n' scenarios corresponding to the use cases.

Let F(UC1) return all the logical predicates corresponding to all the pre/postconditions of scenarios in use case, uc1. Therefore, the input file 1 will scale to:

F(UC1) + F(UC2) + ...+ F(UCn) number of predicates.

For input file 2:

Let there be 'k' number of states in the state chart and let the function F(s) retrieves the number of logical predicates corresponding to state, s,

Therefore, the input file 2 will scale up to:

F(s1) + ***F(s2)*** +..+ ***F(sk)*** number of predicates.

For output file:

The number of sequences generated will be approximately equal to ***O ((V-l)*** + ***T)*** where *'l'* represents the leaf states representing the states with no transitions, *'V'* denotes the whole list of nodes representing all the states in the state chart and *'T'* represents the whole list of edges representing the transitions in the state chart. In the above PM system with a total of 33 states, the number of sequences generated = 16

If F(s1) retrieves the total number of predicates for sequence, s1, then for a total of 'n'

sequences for a state chart, the output file 1 scales to:

F(s1) + F(s2) +...+F(sn) number of predicates.

Time complexity:

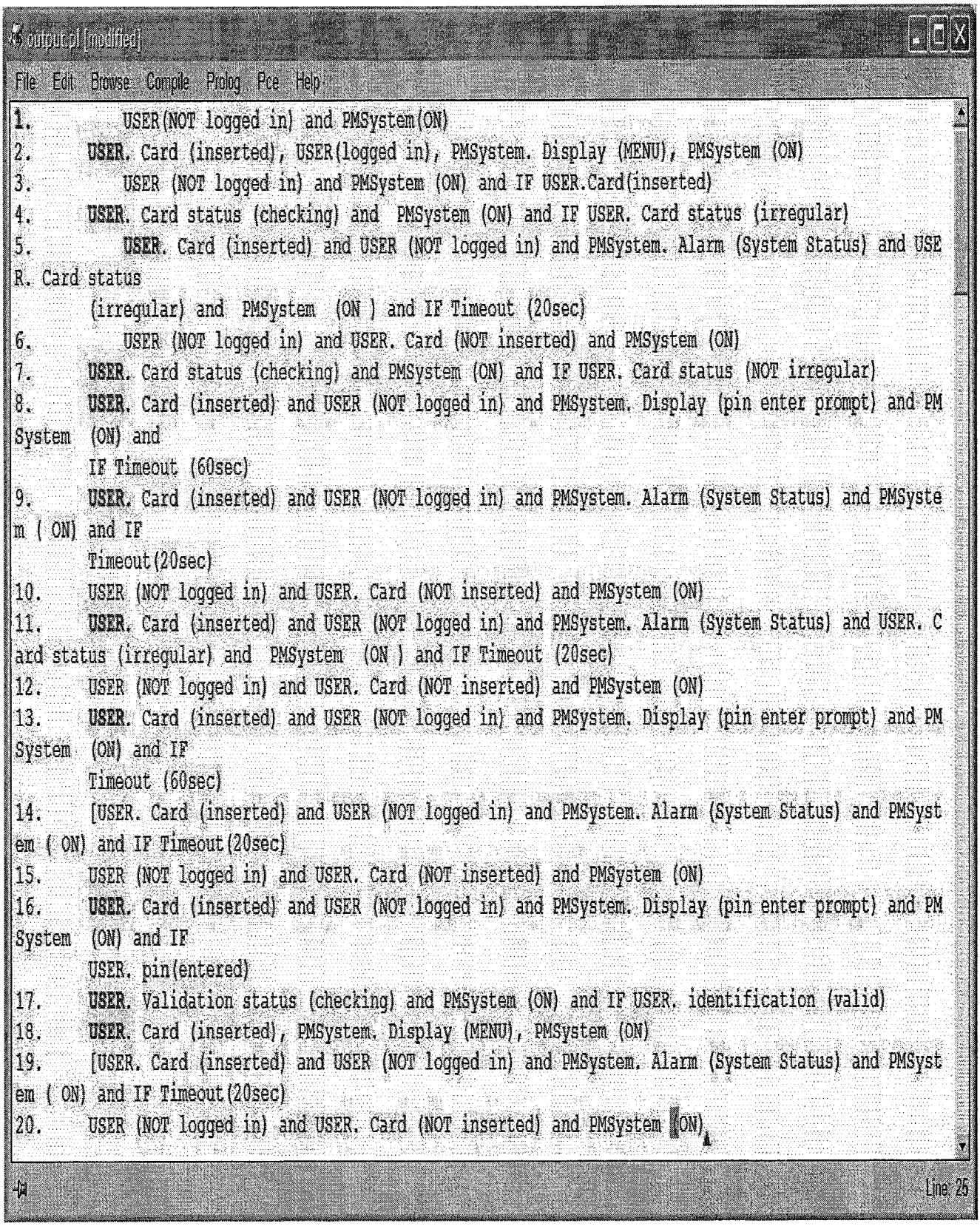
Time complexity for the SAV state chart verification yields:

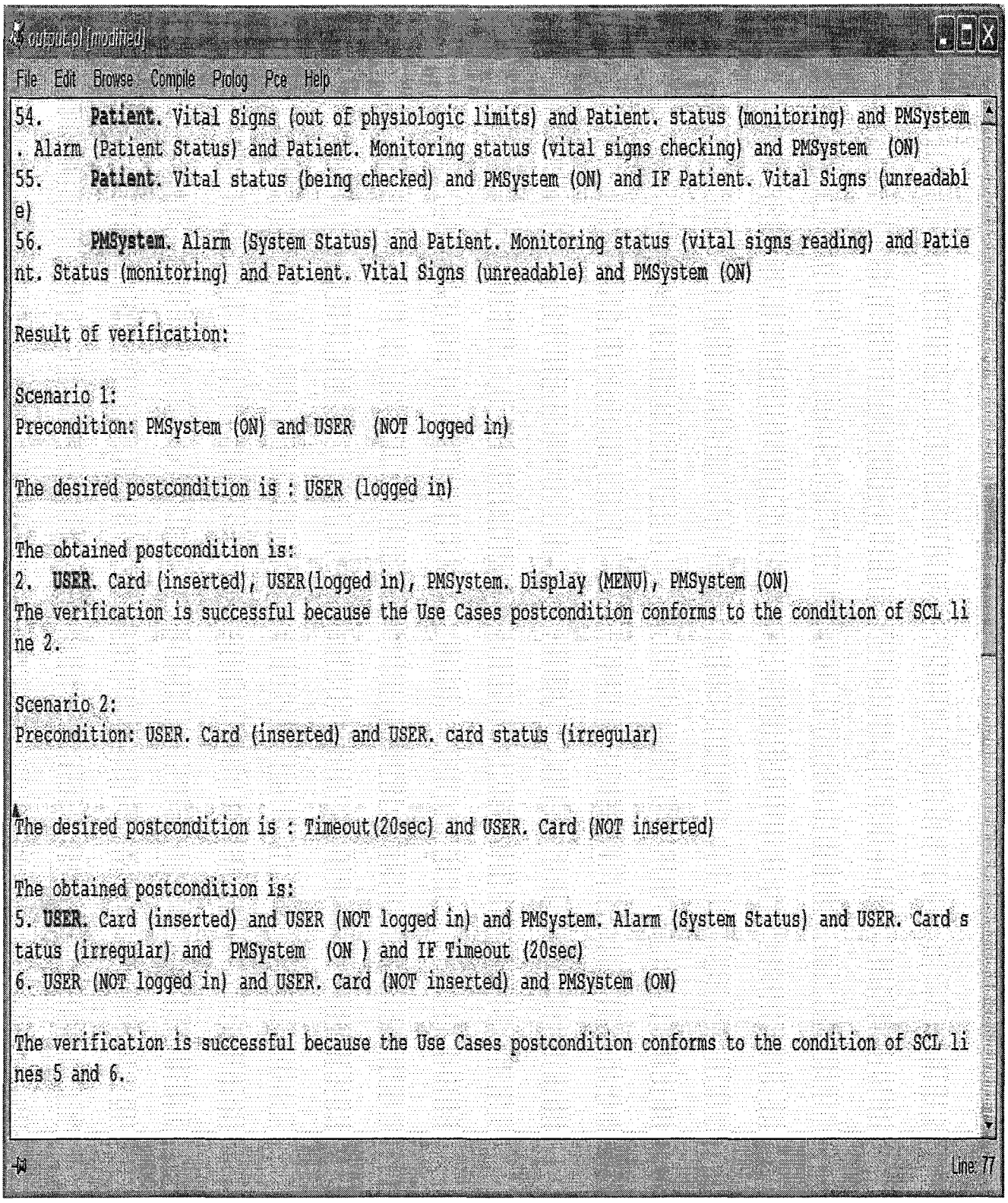
n (T(generate_SCL) + T(generate_SCL_trans) + T(generate_SCL_resultant)) + O(2n)

where 'n' represents the node list denoting states, T(generate_SCL) represents the time needed for the execution of the procedure generate_SCL, T(generate_SCL_trans) represents the time needed for the execution of generate_SCL_trans, T(generate_SCL_resultant) represents the time needed for the execution of generate_SCL_resultant. O(2n) is obtained by O(n) time needed for visiting all the states in the SCL conversion algorithm along with O(n) time needed for searching the SCL statement corresponding to the UCL-pre in the matching algorithm.

For the above described PM system with 33 states, the total execution time ~ 3 seconds.

6.2.6 Screenshot for Semi-Automated Validation





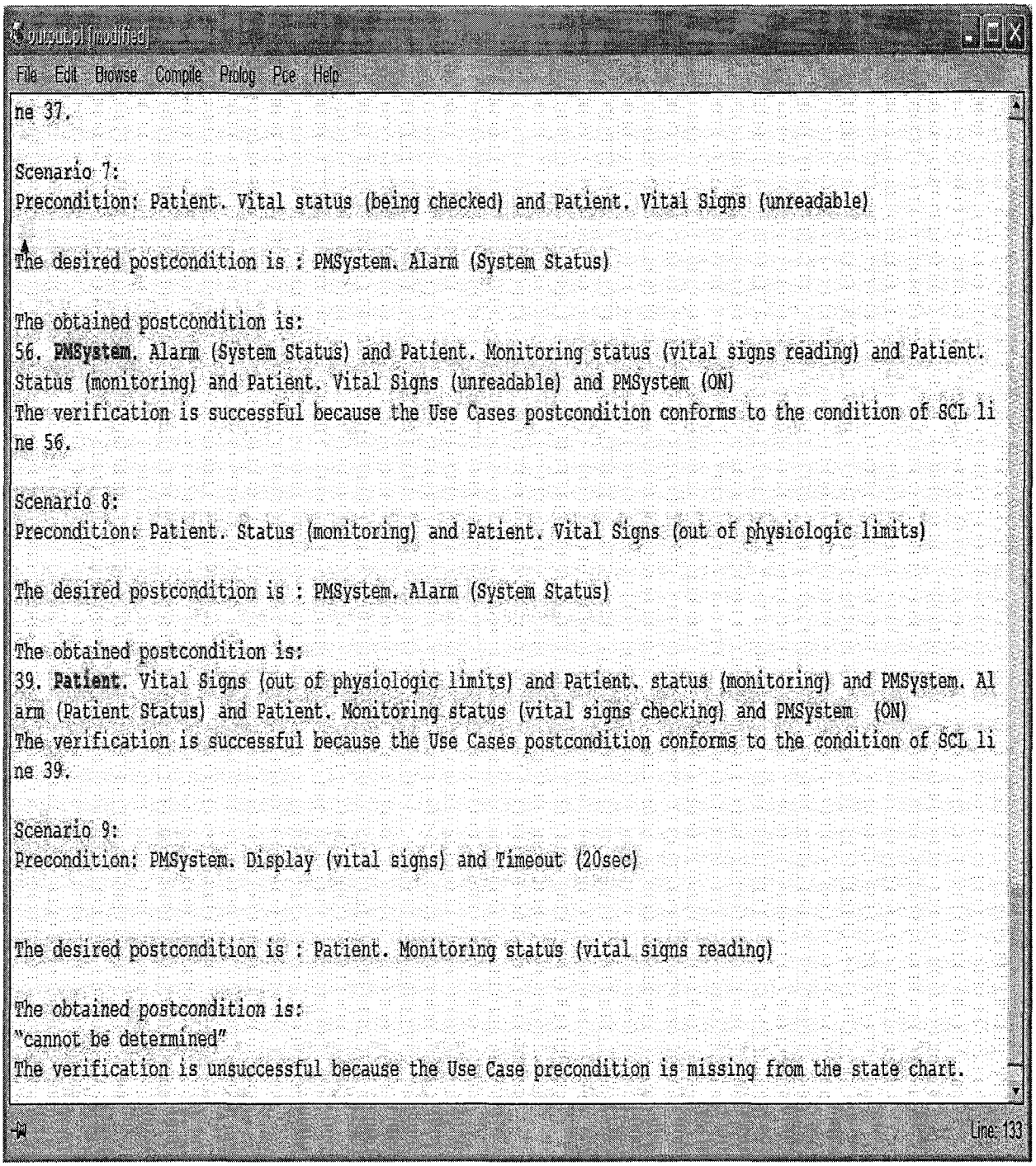


Figure 42: Screen Shot of SAV Output after PSV

Chapter 7. Conclusion

7.1. Summary and Advantages of Proposed Approaches

This research has focused on the importance of Requirements Verification and Validation during early requirement and design stages of software development. Performing requirements verification at early requirement and design stages secures the future of software development. Software systems benefit from verifying requirements at the early stages of software development. Despite the difficulties of using Natural Language informal requirements for requirement verification, a semi-formal representation which preserves NL behavior is used in this research for representing the informal Natural Language requirements as semi-formal Use Cases. Maintaining this semi-formal nature for Use Cases requirements and imparting a logical representation to these semi-formal NL requirements, facilitated in bridging the gap between formal and informal requirements. Most of the existing model checking approaches [5], [6] for requirement verification is very formalized and semi-automated and consume much time and effort. The proposed approaches in this research are automated and rely on a semi-formal NL representation for informal requirements and therefore eliminate most of the disadvantages in existing model checking approaches.

Two approaches are presented in this research for requirement verification and validation. A *Predicate based Sequential Verification (PSV)*, and a *Semi-automated validation* approach. The two approaches are model based approaches and they concentrate on performing early requirement and design verification and validation. The *PSV* approach has verified the domain model against Use Case requirements, detected errors and inconsistencies in the domain model and proposed necessary corrections and improvements to the domain model. The *semi-automated validation* approach has verified the formal design model (state chart diagrams) against the Use Case requirements and detected requirement violations in the state chart model. A proof based strategy, *'Scenario Sequential Verification Strategy'* was described as an alternative method for domain model verification.

For *PSV* and *Semi-automated validation* approaches, the informal NL requirements were converted to a *semi-formal* NL representation based on *DCG.* The domain model used in both of the approaches is based on the 'contract' notion described in section *3.4.2* of Chapter 3. The domain model is formalized based on an extended UML domain model description and *DCG.* A formalized representation for state charts, which considers a state as a characteristic set of *predicates and transitions*, is described for *semi-automated validation.* In order to make the verification process automated and technically feasible, a more formal representation for Use Cases and state charts is proposed for *PSV* verification and *Semi-automated validation.* The formal representation used in PSV verification and Semi-automated validation is termed *'logical predicate'* representation.

The PSV verification approach takes *expected* pre/postconditions of Use Case scenarios, domain model and the Use Case scenarios as input and verifies the domain model against Use Case scenarios' expected postconditions. The output highlights the result of verification and report on improving the domain model in case of any domain model inconsistencies. The scenarios are extracted from Use Cases based on rules and conventions described in Chapter 4. These Use Case scenarios and the expected pre/postconditions of Use Case scenarios are converted to *logical predicates* based on the conversion rules described in Chapter 4. These *logical predicates* (representing *expected* pre/postconditions of scenarios and Use Case scenarios) are verified by the *PSV* module by extracting postconditions of domain operations from the domain model corresponding to the Use Case scenario *steps*. The final Use Case scenario postcondition from the *PSV* process will be verified against the *expected* postconditions to check whether the domain model meets the expected Use Case requirements. The *PSV* process is performed by the *PSV* algorithm described in Chapter 4. The relevance of *PSV* verification is emphasized by the need for checking domain model for errors and inconsistencies as described in Chapter 4. The analysis of various outputs from *PSV* verification and the alternative method which achieves the objective of *PSV* process are also discussed in Chapter 4. A case study on *Patient Management System* is proposed in *Chapter 6* to demonstrate the *PSV* process and the output snapshots from *PSV* process.

The *semi-automated* validation approach is performed for verification of formal design models especially the state chart diagrams. Although the formal design model is derived from the improved domain model obtained from *PSV* process, there may still exist some requirement errors or inconsistencies in the formal model. The inadequacies of formal model generation methods call for the necessity for formal model verification. The other reasons for requirement errors and inconsistencies in formal design model are discussed in Chapter 5. The *semi-automated* validation approach takes semi-formal Use Cases and state chart model as input and verifies the state chart against pre/postconditions of Use Case scenarios derived from the Use Cases. The pre/postconditions and the state chart are converted to *'logical predicate'* representations before performing the semi-automated validation using the conversion rules of Chapter 5. The state chart conversion and verification is performed by the algorithms discussed in Chapter 5. The output of semi-automated validation reveals whether the state chart model conforms to the pre/postconditions of Use Case scenarios.

The advantages of proposed approaches can be summarized to:

- Verification of domain model against the Use Case requirements
- Reporting errors or inconsistencies present in the domain model
- Suggesting improvements in the domain model to tackle the errors and inconsistencies.
- Detecting the invariants for the Use Case scenarios and verifying that the domain model meets the invariants
- Verification of state chart model against Use Case requirements
- The *semi-automated* validation method can be applied to any state chart models, since most of the state chart models use the formalized representation described in Chapter 5.
- The *PSV* and *semi-automated* validation methods can be used independently or in integration with other software development tools provided the assumptions described for *PSV* process and semi-automated validation are satisfied.

Limitations of the proposed Approaches

The proposed approaches use a very strict terminology and representation for the domain model description. Unlike the domain description format adopted by Larman, the domain model in the proposed approaches considers the model to be comprising of actors, attributes and operations related to entities. In other words, the proposed version of domain model serves as a dictionary to the problem domain which defines the entities, the attributes associated with actors and operation related to the entities. The relationships among entities are not explicitly specified and therefore the model lacks the details regarding the internal flow of data. It is assumed that the Use Case description and the domain model possess naming consistencies and operational similarities, hence, a one-one mapping of the Use case decription to domain model description is not taken into consideration to reduce the complexity of the verification approaches. Although the PSV process suggests improvements to the domain model by identifying the wrong predicates in case of inconsistencies, it is advised that the developer should also rely on his judgement in identifying and correcting the exact operations which caused the inconsistencies. The reasons for the inconsistencies detected by the PSV process may depend on the correctness aspects of both Use Case requirements and domain model description. Therefore it is expected that the corrections to the domain model is partially dependant on the mutual agreement between the developer and the stakeholder.

The SAV process makes the assumption that the generated state chart depicts the flow of activities between the various scenarios of a use case and between use cases from the viewpoint of Requirements Engineering.

7.2. Future Work and Open Issues

Although advancement is made in imparting technicality and automation to requirements verification and validation in the early requirement and design stages, there is still some open issues present pertaining to the proposed approaches and Use Case based requirements engineering.

The *PSV* verification and *Semi-automated validation* are model checking requirement

verification approaches and therefore possess the disadvantages of model checking. Since the *semi-automated validation (SAV)* generates all available sequences from state chart for verification, SAV suffers from the state explosion problem. The working of these approaches is based on specific assumptions described in Chapter 4 and Chapter 5. The logical conversions described for Use Cases and state machines rely on specific conversion rules described in Chapter 4 and Chapter 5. One of the main open issues related to the proposed model checking approaches is that the semi-formal representation of Use Cases is based on a restricted Natural Language based on the DCG described in Chapter 3. Complex constructs in informal Natural Language like iterations are not taken into consideration for the semi-formal representation of Use Case requirements. In the case of *PSV* verification, the expected postconditions of Use Case scenarios are manually recorded in the semi-formal nature by mutual agreement between the designer/developer and stakeholders. As a future enhancement, the extraction of expected postconditions of Use Case scenarios can be automated in *PSV* verification. The semantics of semi-formal Natural Language description of Use Cases can be improved and more complex English constructs can be added to the DCG Natural Language description. The proposed domain model description used for both of the approaches uses a restricted verbal format for *AddedConditions* and *WithdrawnConditions*. The *PSV* verification and *semi-automated validation* concentrate only on verifying models against Use Case scenarios and do not focus on verifying properties and constraints of Use Cases. This issue is open and can be integrated to the proposed approaches to obtain a complete coverage of requirement verification in the early requirement and design stages. The *semi-automated validation* restricts the formal representation of states in state charts to be represented in the form of predicates. The use of other logic based formal representations for state charts in *semi-automated validation* is left open for future considerations. The formal state chart representation used in *semi-automated validation* allows only simple states, normal states and compound states and does not consider a situation when a state can have concurrent transitions. The inclusion of states with concurrent transitions in *semi-automated validation* is also open to future considerations. The *semi-automated validation* deals with Use Cases and state charts individually and does not take care of Use Case and state chart integrations. Although, it is possible to adapt the *semi-automated validation* for state chart

integration (global state chart), all the Use Cases have to be composed manually to obtain a global Use Case model for performing the semi-automated validation of the global state chart. This issue is discussed towards the end of Chapter 5. Finally, the proposed approaches do not deal with the various levels of requirements and non functional requirements.

The SAV process proposed in this thesis does not consider iterations for Use Case requirements. The SAV process can be adapted to include iterations by selecting the frequently visited loops and treating them as individual scenarios. The SAV verification strategy can then be easily performed on these individual scenarios.

Although the SSV startgey is based on Hoare logic calculus, the style of logical proving differs from the conventional Hoare proving style. Therefore the SSV strategy can be viewd as an independent proving startgey used explicitly for proving scenarios. As a futue enhancement, this proving strategy can be simulated on Hoare logic based theorem proving tools like Krakatoa with an adapted proving style involving propositional calculus.

An interesting approach in the area of formal verification is to integrate theorem proving with model checking, this can take care of most of the individual disadvantages of theorem proving and model checking up to a certain point. The basis for this idea lies on abstraction, which bridges the gap between the two formal verification methods. An example of this approach is discussed in [20] where a model checker is used to build a simple model and the theorem proving approach is used to ensure that the model preserves every single property of the original system.

In a nutshell, although the proposed approaches have presented a promising future in requirement verification and validation, there is still a sea of problems to be resolved in the area of Use Case based requirements engineering and formal requirement verification methods.

Bibliography

[1] A. Davis. *Software Requirements: Object, Function, and States*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[2] Ahmed Seffah, Jan Gulliksen and Michel C. Desmarais. *Linking User Needs and Use Case-Driven Requirements Engineering,* Human-Computer Interaction series, Springer Publications, 2006

[3] AIPS-98 Planning Competition Committee. PDDL - *The Planning Domain DefinitionLanguage,* Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[4] Alan W. Brown, David J. Carney. *Principles of CASE Tool Integration,* Oxford University Press, SEI Publication, 1994

[5] Arie Gurfinkel. *Model-Checking Software Using Precise Abstractions*, IFIP WC on Verified Software: Tools, Techniques, and Experiments, October 2005.

[6] Ariel Fuxman, John Mylopoulos, Marco Pistore, Paolo Traverso. *Model Checking Early Requirements Specifications in Tropos*, Electronic Edition , IEEE Computer Society DL,pages:174-181,2001

[7] A. van Schouwen. *The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems,* Technical Report 90-276, Queens University, Hamilton, Ontario, 1990.

[8] B. Benyó. *Verification of complex object oriented systems*, 4th IEEE International Workshop on Design and diagonostics of electronic circuits and systems 2001, IEEE DDECS, Hungary, pages: 289-290, 2001

[9] Bernd B Schmidt, T.Clark. *Object Modeling with OCL*, Springer-Verlag/Sci-Tech/Trade, first edition, 2002

[10] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. *Translating the Object Constraint Language into First-order Predicate Logic*, Proceedings of VERIFY, Workshop at Federated Logic Conferences (FLoC), 2002

[11] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[12] Bill Venners, Artima. *Design by Contract; A Conversation with Bertrand Meyer, Part II,* http://www.artima.com/intv/contracts.html, December 2003

[13] Björn Regnell, Kristofer Kimbler. *Improving the Use Cases Driven Approach to Requirements Engineering,* RE'95: Proceedings of the second IEEE International Symposium on Requirements Engineering, Page: 40, 1995

[14] B. Boehm. *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[15] Boehm, Barry, and Wilfred J. Hansen. *The Spiral Model as a Tool for Evolutionary Acquisition.* CrossTalk May 2001.

[16] Booch, G., Jacobson, I., and Rumbaugh, J. *Unified Modeling Language Users Guide,* Addison Wesley Longman, 1999

[17] Booch, Rumbaugh, Jacobson. *Unified Modeling Language User Guide,* Addison-Wesley, 1998

[18] C.A.R.Hoare. *An Axiomatic Basis for Computer Programming*, Communications of the ACM, Volume 12, October 1969

[19] C. A. R. Hoare. *Communicating Sequential Processes,* Prentice Hall International Publication, June 2004

[20] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. *Property preserving abstractions for the verification of concurrent systems.* Formal Methods in System Design, 6: pages: 11–44, 1995

[21] C. Ramamoorthy, A. Prakesh, W. Tsai, and Y. Usuda. *Software engineering: Problems and perspectives,* IEEE Computer, pages 191-209, October 1984.

[22] C.W. Johnson. *The Natural History of Bugs: Using Formal Methods to Analyse Software Related Failures in Space Missions*, Springer Berlin / Heidelberg publications, Volume 3582/2005, pages 9-25, August, 2005

[23] Charles Gregory Nelson. *Techniques for Program Verification.* Doctoral Thesis. UMI Order Number: AAI8011683, 1980

[24] CMU, SEI. *Domain Engineering: A Model-Based Approach.* http://www.sei.cmu.edu/domain-engineering/domain_engineering.html, 2004

[25] Cockburn, Alistair. *Writing Effective Use Cases*, Addison-Wesley, 2001

[26] Cockburn, Alistair. *Use Cases: Ten Years Later*, STQE, SQE, Vol. 4, No. 2, March/April 2002

[27] Colin Snook, Michael Butler. *UML-B: Formal modeling and design aided by UML*, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 15, Pages: 92 – 122, January 2006

[28] D.W. Lovelan. *Automated theorem proving: mapping logic into AI*, International Symposium on Methodologies for Intelligent Systems, Proceedings of the ACM SIGART international symposium on Methodologies for intelligent systems, Pages: 214 – 229, 1986

[29] Divya K Nair, Stephane S Some. *Use Case based Requirements Verification: Verifying the consistency between use cases and assertions*, paper accepted for 9th International Conference on Enterprise Information Systems, June 2007

[30] Divya K.Nair, Stéphane S. Somé. *A Formal Approach to Requirement Verification*, SEDE 2006, pages: 148-153, 2006

[31] Doron Peled, *Software Reliability Methods*, Bell Labs/Lucent Technologies, Springer-Verlag Publications, June 2001

[32] Edmund M. Clarke, Orna Grumberg, Doron A. Peled. *Model Checking*, MIT Press, January 2000

[33] Eelco Visser. *An introduction to Program Transformation*, Software Technology Colloquium, October 2003.

[34] F.C.N Pereira, D.H.D Warren. *Definite clause grammars for language analysis- a survey of the formalism and comparison with augmented transition networks*, Artificial Intelligence, 13, pages: 231-278, 1980

[35] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*, Wiley Publications, page 10, 1998

[36] Giuseppe Lami. *QuARS: A Tool for Analyzing Requirements*, TECHNICAL REPORT, CMU/SEI-2005-TR-014, September 2005

[37] Harel D, *State charts: An Z formalism for complex systems*, Science of Computer Programming. Vol. 8 (1987), p. 231

[38] H. M. Munoz-Avila, D. W. Aha, D. Nau, R. Weber, L. Breslow, and F. Yaman. *SiN: Integrating case-based reasoning with task decomposition.* In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, pages: 999-1004, 2001.

[39] Howard Pospesel. *Introduction to Propositional Logic*, Prentice Hall, edition 3, 1997.

[40] Howard Pospesel. *Introduction to Logic: Propositional Logic,* 3rd Edition, Prentice Hall, October 1999

[41] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide,* Wiley Publication, 1997.

[42] Ivy Hooks. *Writing Good Requirements,* Proceedings of the Fourth Annual Symposium, INCOSE working Group Papers, Volume II, 1994

[43] Jacobson, I. *Object Oriented Software Engineering, A Use Cases Driven Approach*, Addison-Wesley, 1992

[44] Jacobson I, Booch G, Rumbaugh J. *The Unified Software Development Process.* Addison-Wesley., 1999

[45] Jacobson. *Object-Oriented Software Engineering*, Addison Wesley, 1992

[46] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*, Cambridge Tracts In Theoretical Computer Science; Vol. 3, Pages: 131, Cambridge University Press, 1988

[47] J. Penix, C. Pecheur, and K. Havelund. *Using Model Checking to Validate AI Planner Domain Models,* In Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard, 1998.

[48] John D. McGregor. *Domain *,* Journal of Object Technology, Vol. 3, No. 7, July-August 2004

[49] Karl W. E. *Software Requirements 2: Practical techniques for gathering and managing requirements throughout the product development cycle*, 2nd ed., Redmond: Microsoft Press, 2003

[50] Krzysztof Apt. *Verification of Sequential and Concurrent Programs*, Springer publications, edition 2, 1997.

[51] Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design. Prentice Hall, 1998

[52] Lester O. Lobo and James D. Arthur. *Effective Requirements Generation:Synchronizing Early Verification & Validation, Methods and Method Selection Criteria,* ACM-class: D.2.1; D.2.9, 2005

[53] Linda Rosenberg. *Methodology for Writing High Quality Requirement Specifications and for Evaluating Existing Ones,* Presented at Software Assurance Technology Center: NASA Goddard Space Flight Center Greenbelt, MD Software September 24, 1998

[54] M.M. West, D.E. Kitchin and T.L. McCluskey. *Validating Planning Domain Models Using B-AMN*, PlanSIG, Netherlands, 2002

[55] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993

[46] Martin Fowler. *Patterns of Enterprise Application Architecture,* Addison Wesley, November 2002

[57] Martin Giese and Rogardt Heldal. *From Informal to Formal Specifications in UML,* Proceedings of UML2004, Seventh International Conference on the Unified Modeling Language, October 2004

[58] Martin Glinz. *Improving the Quality of Requirements with Scenarios*, Proceedings of the Second World Congress for Software Quality (2WCSQ), pages: 55-60, September 2005

[59] Martin, J., and Odell, J. *Object-Oriented Methods: A Foundation.* Englewood Cliffs, NJ.: Prentice-Hall, 1995

[60] Matt Kaufmann and J Strother Moore. *Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification*, Spanish Royal Academy of Science (RACSAM), 98(1), pp. 181-196, 2004.

[61] Michael Boggs, Wendy Boggs. *Mastering UML with Rational Rose*, Sybex, Bk&CD Rom edition, August 1999

[62] Michael Collins. *Formal Method: Report on Dependable Embedded Systems,* Carnegie Mellon University 18-849b Spring 1998

[63] Michael Fisher and Richard Owens. *Executable Modal and Temporal Logics*, Lecture Notes in Computer Science, Springer publications, February 1995

[64] Michael G, Christel Kyo C. Kang. *Issues in Requirements Elicitation*, Technical Report, CMU/SEI-92-TR-012, September 1992

[65] N. Leveson. *Safeware: System Safety and Computer*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[66] Nilesh Parekh. *Software Verification and Validation Model*, Buzzle Web Article, 2005

[67] Ohnishi, A., Potts, C. *Grounding Scenarios in Frame-Based Action Semantics*, Proc. of 7th International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ'01), pages: 177-182, 2001

[68] Ohnishi, A., Zhang, H., Fujimoto, H. *Transformation and Integration Method of Scenarios*, Proc. of 26th IEEE International Computer Software and Applications Conference (COMPSAC'02), pages: 224-229, 2002

[69] Ohnishi, A. *Software Requirements Specification Database Based on Requirements Frame Model*, Proc. of the IEEE second International Conference on Requirements Engineering (ICRE'96, pages:221-228, 1996

[70] OMG. *OMG Unified Modeling Language Specification,* version 1.3, 2000

[71] Pamela Zave and Michael Jackson. *Four dark corners of requirements engineering*, ACM Transactions on Software Engineering and Methodology VI (1), pages: 1-30, January 1997

[72] Paulson, Lawrence C. *Isabelle: A Generic Theorem Prover,* Lecture Notes in Computer Science, volume 828, May 2005

[73] Pradip Kar. *Characteristics Of Good Requirements, 6th* INCOSE *Symposium, 1996*

[74] R. Lutz. *An overview of REFINE 2.0,* Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1993.

[75] Raymond M. Smullyan, *First-Order Logic*, Dover Publications, January 1995

[76] Reiner Hähnle, Kristofer Johannisson, Aarne Ranta. *An Authoring Tool for Informal and Formal Requirements Specifications*, Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, Lecture Notes In Computer Science, Vol. 2306 , Pages: 233 – 248, 2002

[77] Richard Thayer, Merlin Dorfman. *"Software Requirements Engineering"*, 2nd edition, IEEE Computer Society, 1997

[78] Ricky W. Butler, Victor A, Ben L. Di Vito, Kelly J. Hayhurst, C. Michael Holloway, Jeffrey M. Maddalon, Paul S. Miner. *NASA Langley's Research and Technology-Transfer Program in Formal Methods,* Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS95), Gaithersburg, MD, June 1995.

[79] Robert S. Boyer, Matt Kaufmann. *The Boyer-Moore Theorem Prover and Its Interactive Enhancement*, Computers and Mathematics with Applications, 29(2), pages: 27-62, 1995

[80] Robinson, J.A. *A machine-oriented logic based on the resolution principle.* Jour. Assoc. for Comput. Mach, 1965, 23-41.

[81] Ruth Malan, Hewlett-Packard Company, and Dana Bredemeyer. *Functional Requirements and Use Cases*, Bredmeyer Consulting, 1999

[82] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*, Computer Science Laboratory, SRI International, MenloPark, CA, December 2001

[83] Ste´phane S. Some. *Supporting Use Cases based requirements engineering,* Information and Software Technology, 48, pages: 43–58, 2005

[84] St´ephane S. Some. *UCEd: a tool for Use Cases based requirements acquisition*, Automated Software Engineering Conference, 2003.

[85] Stephane S Some. *UCED Use Cases Development Approach*, Twiki-SEG 3601 Web, http://cserg0.site.uottawa.ca/seg/pub/SEG3601/LaboratoireUCEd/ucedGuide.pdf, 2003

[86] Stephane S Some. *An approach for the synthesis of state transition graphs from Use Cases*, Proceedings of the International Conference on Software Engineering Research and Practice (SERP'03), volume I, pages 456–462, June 2003.

[87] Stefania Gnesi. *Model Checking of Embedded Systems*, ERCIM News No. 52, January 2003

[88] T. L. McCluskey and D. E. Kitchin. *A Tool-Supported Approach to Engineering HTN Planning Models*, Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence, 1998.

[89] Tatsuya Toyama, Atsushi Ohnishi. *Rule-based Verification of Scenarios with Pre-conditions and Post-conditions,* Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05) - Volume 00, Pages: 319 - 328, 2005

[90] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science, Springer publications, May 2002

[91] United States Department of Energy, Albuquerque Operations Office. *Guidelines for Requirements Management*, Department of Energy Quality Managers Software Quality Assurance Subcommittee, Reference Document SQAS19.01.00, 2000

[92] W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, Braunschweig, 2. edition, 1987

[93] Weinstock, Charles B, Gluch, David P. *A Perspective on the State of Research in Fault-Tolerant Systems*, Final Report, Carnegie-Mellon University, Pittsburgh, SEI, 1997

[94] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese and Reiner. *The KeY Tool*, Software and System Modeling Journal, vol.4, pages: 32-54, 2005

[95] Wolfgang Grieskamp, Yuri Gurevich. *Generating Finite State Machines from Abstract State Machines*, International Symposium on Software Testing and Analysis, Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pages: 112 – 122, 2002

[96] Wos L. *Automated reasoning: basic research problems*. Report ANL/MCSTM-67, Argonne National Lab, Argonne, IL, March 1986.

[97] Wouter Geurts, Klaas Wijbrans, CMG Den Haag B.V. *Testing and Formal Methods,* BOS Project Publication, May 1999

[98] Zave, P., Jackson M. *Four dark corns of requirements engineering,* ACM Transaction on Software Engineering and Methodology, 6 (1), pages 1-30, 1997